DETECTING FRAGILE COMMENTS

by

Inderjot Kaur Ratol

School of Computer Science

McGill University, Montreal

August 2017

A THESIS SUBMITTED TO MCGILL UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF

MASTER OF SCIENCE

ABSTRACT

The development lifecycle of a software system demands incessant improvements in the source code of a system to maintain its high quality with improved performance and code readability. Refactoring is a common software development practice that reshapes the internal structure and non-functional properties of a system without modifying its core functionality. Many simple refactorings like renaming code elements, extracting a snippet from large method to form new method etc. can be performed with the help of automatic tools. Renaming code elements like classes, interfaces or methods is a widely used refactoring activity. With tool support, rename refactorings can rely on the program structure to ensure correctness of the code transformation. Unfortunately, the textual references to the renamed identifier present in unstructured comment text cannot be formally detected through the syntax of the programming language. These textual references to the previous version of a renamed identifier pose threats to the consistency between code and comments, which leads to poor program comprehensibility. The comments containing such textual references become fragile with respect to the renamed program element and are referred to as fragile comments.

This thesis proposes a new rule-based approach to detect and fix the fragile comments that result from renaming the identifiers. We implemented this approach for the Java programming language in the form of an Eclipse plug-in called Fraco. Fraco takes into account the type of an identifier, its morphology i.e. the part-of-speech tag and its inflectional form, its scope that defines its visibility in the source code and the location of comments in the source code with respect to the identifier.

We evaluated the performance of our technique, as implemented for Java in Fraco, by comparing its precision and recall against hand-annotated benchmarks created for both development and test sets each containing six target Java systems, and also compared the results against the performance of Eclipse's automated in-comment identifier replacement feature. Fraco performed with an average of 99% precision and recall on most components of both development and test data sets, and generally outperformed the baseline Eclipse feature. An average percentage of 25% of the total identifiers of category `type` and `method` in the data sets had fragile comments after renaming, which further motivates the need for research on automatic comment refactoring.

RÉSUMÉ

Le cycle de développement d'un système logiciel exige des améliorations incessantes dans le code source d'un système afin de maintenir élevée sa qualité en termes de performance et de lisibilité du code. Le réusinage de code est une pratique courante dans le développement logiciel qui remodèle la structure interne et les propriétés non fonctionnelles d'un système sans modifier ses fonctionnalités principales. Plusieurs transformations simples peuvent être effectuées à l'aide d'outils automatiques. Renommer des éléments du code comme une classe, une interface, une méthode, etc. est une tâche qui revient souvent lors d'un réusinage. Avec le support d'outils, le renommage peut se baser sur la structure d'un programme pour s'assurer de l'exactitude de la transformation du code. Malheureusement, les références textuelles aux identifiants renommés présentes dans les commentaires non-structurés ne peuvent être détectées formellement à travers la syntaxe du langage. Ces références textuelles aux identifiants renommés sont des obstacles à la synchronisation entre le code et les commentaires, ce qui détériore la compréhensibilité du programme. Les commentaires incohérents peuvent donc devenir une source d'introduction de bogue ou induire en erreur les développeurs.

Cette thèse propose une idée nouvelle combinant le renommage à la détection des commentaires fragiles en introduisant une nouvelle approche basée sur un ensemble de règles pour détecter et corriger les commentaires fragiles produits par un renommage d'identifiant, implémentée sous forme d'une extension de la plateforme Eclipse. L'outil, nommé Fraco, considère le type d'identifiant, sa morphologie, c'est-à-dire l'étiquette partielle et sa forme inflexionnelle, sa portée qui définit sa visibilité dans le code source et le lieu des commentaires par rapport à l'identifiant.

La précision et le rappel de l'outil proposé sont évalués à l'aide de systèmes de référence

annotés manuellement pour les ensembles de développement et d'évaluation, chaque ensemble contenant six systèmes en Java. Les résultats sont comparés à la performance de la fonctionnalité de remplacement d'identifiants à l'intérieur de commentaires intégrée à Eclipse. Fraco a performé une précision moyenne de 99% et un rappel presque optimaux sur la plupart des composantes des ensembles de développement et d'évaluation et performe en général mieux que la fonctionnalité de base d'Eclipse. Un pourcentage moyen de 25 % des identifiants totaux de catégorie `type` et `méthode` dans les ensembles de données présentait des commentaires fragiles après le changement de nom, ce qui motive davantage la recherche sur le refactoring automatique des commentaires.

ACKNOWLEDGMENTS

The completion of this thesis has been a great learning journey which has left me indebted to many people who have supported, contributed and guided me, many a times, by going out of their way in doing so.

First and foremost, I would like to thank my supervisor Prof. Martin P. Robillard for his unconditional support, methodical guidance and utmost care towards the ideation, development and completion of this thesis. Being confidently able to finish this thesis, I respect and admire his patience in successfully training and repeatedly encouraging a novice like myself who knew little about research and academic writing. I feel blessed to have found a mentor in him, as he has not only been an excellent academic guide but has nurtured great ethical and professional values in me, which surely will help me achieve professional success and a respectable life ahead. For all his support and teachings, I am and shall always be grateful.

I am indebted and sincerely grateful for the funding endowed upon me by my supervisor through the Natural Sciences and Engineering Research Council of Canada (NSERC), without which this thesis, my sustenance, and this world class education with phenomenal exposure would not have been possible. My deepfelt gratitude also vests with all the professors who provided me with the opportunity to be their Teaching or Research Assistant.

I am grateful to Mr. Jaspal Singh for accepting to annotate the data for evaluation, giving more credibility to the work presented in this thesis and also, to Mr. Mathieu Nassif & Mr. Alan Do-Omri for translation of the abstract of this thesis into French. Further, my sincere thanks to my fellow researchers in Software Evolution and Research Group (SWEVO) and Computation and

Logic Group for their valuable feedback, inputs and for maintaining a great learning ambiance at the lab which has advanced my knowledge and social skills greatly.

Furthermore, a big shout out for my friends here at McGill who have made my stay in Montreal pleasant, fun and once in a lifetime experience. My heartfelt thanks to my friend Bayar Goswami for his overall support and especially for taking time out to help me proof read some of my writings and this thesis. I, also, am thankful to him for his encouraging words that helped me through my low times. I would also like to thank my friend Shivang Vij for supporting me and encouraging me to pursue higher education. Without his help, I would never have gotten this far.

Finally and most importantly, I express my profound gratitude towards my parents and my siblings for their unconditional support, sacrifices and for always encouraging every endeavour of mine without a speck of judgment or doubt. I hope this makes them proud and expresses my loads of love.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the evolution and growth of a software system, there often is a need for improvements to its internal structure and organization, known as *refactoring* [15]. Refactoring a system helps to maintain the quality of the code and increases its comprehensibility. The changes performed during the refactoring process are known as functionality-preserving changes i.e., the changes that affect the internal structure or non-functional attributes of a system without affecting its external functionality. Individual refactorings can take many forms, including renaming code elements, extracting statements into a method, changing a method's signature etc. [15]. Renaming code elements is, in particular, a very common type of refactoring performed to maintain a set of names that reveal the purpose of code elements to facilitate code comprehension [27]. Also, the identifiers composed of full words prove to be more descriptive than the identifiers made up of abbreviations or single words. The full-word identifiers result in better comprehension and precisely capture the computational intent of their related code elements [28].

Many refactoring activities can be fully or partly automated by tools [36]. Examples include JetBrains Resharper [2] for C# and Eclipse's built-in refactoring tool [1] for Java. Such tools support code transformations by automatically changing a system's source code based on a selection from a catalog of refactorings and, when applicable, the parameterization of the refactoring. Studies show that, despite the prevailing criticism about automatic refactoring tools' adoptability and minimalistic use by developers in practical scenarios, renaming code elements is one of the most

popular refactoring activities performed using automated refactoring tools [35, 37, 54].

Automating laborious refactoring tasks, such as renaming identifiers, relies heavily on encoded knowledge of the rules of a programming language to perform the correct code transformations. Unfortunately, references to a renamed identifier in unstructured comment text cannot be formally detected through the syntax of the programming language, and are thus *fragile* with respect to identifier renaming. We introduce the term *fragile comments* to refer to comments which, upon a given type of modification to the source code, become inconsistent leading to confusion and bug introduction [50]. For example, in the context of identifier renaming, a comment is considered fragile if it is likely to become inconsistent when the identifier is renamed. In one study of three different projects, the authors observed that 97% of the source code changes made while refactoring also needed to change the comments for maintaining coherence between comments and identifiers [14].

Inconsistencies between code and comments are a problem because programmers rely on comments to understand the code and relationships between the different parts of code, its usage and to communicate amongst each other [39, 48, 60]. Comments present in the source code of a system aid the developer in program comprehension and succinctly showcase a coder's intentions behind writing a piece of code. To avoid introducing inconsistencies between comments and code during refactoring, automatic refactoring tools need additional support to analyze and detect the fragile comments followed by their potential modification to resolve the detected fragility.

Existing techniques for comment synchronization fall into two camps. The first camp consists of simple approaches based on exact lexical search and replacement. For example, the refactoring support in Eclipse's Java development tools component [12] provides an option to search-and-replace the occurrences of a renamed identifier in text strings including comments. Pure lexical approaches can be helpful in some cases, but their precision is too low to be useful in the general case. In the case of semi-structured comments (e.g., when combined with the use of in-comment

tags such as `@param` in Javadocs), text-replacement based approaches typically work well, but these constitute a small subset of all possible comments that forms the easiest subset of identifier references to detect. This subset comprises of only the exact matches of an identifier that can be easily detected with simple string comparison. The second camp consists of specialized but *domain-specific* approaches that can detect inconsistencies between comments and code for a subset of programming concepts like synchronization, locking and memory allocation [44, 50, 51]. Though, domain-specific techniques can achieve impressive precision, they are limited to a specialized subset of all possible types of comments.

This thesis advances the state-of-the-art refactoring techniques by introducing Fraco, a *general-purpose* tool-supported approach for detecting and fixing fragile comments when renaming identifiers in Java source code [42]. Fraco relies on a new rule-based algorithm that takes into account the type of an identifier, its morphology i.e. the part-of-speech tag and inflectional form, the scope of the identifier defining its visibility in the source code, and the location of comments while detecting fragility with respect to a single identifier. The proposed approach leverages Natural Language Processing (NLP) techniques to apply morphological analysis on the code comments in combination with the information extracted from language conventions to apply fragility detection rules. The approach successfully avoids the limitation of naive text-replacement approaches that generate large amounts of false positives, while not relying on any domain-specific rules that would limit the approach to a subset of comment types.

1.1 Motivation

This section illustrates the challenge of detecting fragile comments when renaming identifiers with three cases taken from the source code of the Checkstyle project version 7.2 [6]. The discussion is further enhanced with descriptions of the behavior of Eclipse's in-comment identifier replacement feature, hereafter, referred to as *Eccore* (Eclipse Comment Refactoring).

Checkstyle defines a class `Check`, which is the base class for various checking rules. Here the issue is that "check" is also a very commonly used word when documenting methods, such as the one illustrated in Figure 1.1. If one wishes to rename class `Check` to `Rule` for example, a naïve text replacement feature, such as *Eccore*, will erroneously replace all comments that simply mention "check" as an action verb indicating that a method "checks" something, thus generating a large number of false positives.

```
/**
 * Check whether a class may be considered as
 * a checkstyle module. Checkstyle's modules are
 * non-abstract classes [...]
 */
private static boolean isCheckstyleModule(Class<?> loadedClass) {
```

Figure 1.1 – Example of false positive when renaming the class named `Check`.

Another challenge is to determine where to be permissive or strict with case and word morphology. For example, Checkstyle defines a public inner class `Listener`. If one wishes to rename `Listener` to `Observer`, a general case-insensitive matching strategy would generate many false positives, while a case-sensitive matching strategy (such as Eccore) would miss important comments such as the use of the keyword `listener` in Figure 1.2. Therefore, the fragile comment detection technique needs to be sensitive to the scope of the comments while allowing for case-insensitive matches.

As a final example, it is worth noting that some comments can come very close to referring to an identifier without mentioning the exact identifier. Figure 1.3 shows a typical case of identifier re-statement in plain language. In this situation, flipping the polarity of the boolean field "ignore" to "use" would require renaming the identifier to something like `setUseInlineTags`, which would silently render the comment inconsistent with the code. This case is also not detected by *Eccore*.

```
/** Represents a custom listener. */
public static class Listener {

  private String className; /* Name of the listener class */

  /** @return the class name of listener. */
  public String getClassname() {return className; }
}
```

Figure 1.2 – Example of false negatives when using case-sensitive matching.

```
/**
* Sets whether inline tags must be ignored.
* @param ignoreInlineTags whether inline tags
* must be ignored.
*/
public void setIgnoreInlineTags(boolean ignoreInlineTags){
```

Figure 1.3 – Example of indirect mentions of identifiers.

These examples only illustrate a small subset of the situations where it is non-trivial to accurately detect fragile comments. In general, the richness and variety of commenting practices means that simple text-replacement algorithms cannot adequately cope with the problem of detecting fragile comments.

1.2 Contributions

The contributions of this thesis include: a) A general and language-independent formulation of the problem of *fragile comment detection*; b) A tool-supported algorithm for the automatic detection of renaming-induced fragile comments in Java source code; c) A publicly-available benchmark of fragile comments that can be used for independent research[1]; d) Empirical data evaluating both the proposed algorithm and a publicly-accessible tool available as part of the Eclipse IDE; and e)

[1] http://cs.mcgill.ca/~swevo/inderjotmsc/

The specification and implementation of a technique to resolve detected fragile phrases in code comments.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 provides the required background details and review of the related work. The background details section provides basic information about different types of inconsistencies between comment and code, the set of Natural Language tools and the elements of a project that are the core structural elements in our solution designed for the problem of fragile comments, e.g., types of comments, identifiers. Also, it discusses various challenges involved in analyzing code and comments to detect fragile comments.

Chapter 3 presents a precise formulation of the fragile comment detection problem that can be instantiated for different programming languages.

Chapter 4 presents the rules used to detect and link the appropriate comments with code elements.

Chapter 5 introduces the algorithm used for detection of fragile comments and its implementation as an Eclipse plug-in for the Java language. It also illustrates the different types of fragility resolution methods offered by the tool.

Chapters 6 and 7 present the design of the evaluation study and the obtained results. Finally, the conclusion is presented in Chapter 8.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents basic information about the concepts and elements required for designing a solution to the problem of fragile comments. Also, it presents review of the past research work in the field of inconsistency detection, comment analysis and identifier renaming.

2.1 Background

This section provides an introduction to the different types of inconsistencies between code and comments, and the Natural Language Processing techniques used to design a solution to the problem of fragile comment detection. Further, in this section, we list the elements of the source code of a software system that are important for the detection of fragile comments like the different types of comments and identifier types present in a system. Additionally, we discuss the various challenges faced during comment analysis as it involves the understanding of natural language in the context of software engineering.

Types of Inconsistencies: In the context of programming languages, inconsistencies between code and comments can be described as the discrepancies between the code element and the description of its functionality written in the form of comments. Inconsistencies are bilateral i.e. changes in the code can render a comment inconsistent and vice-versa. Due to these inconsistencies, two types of problems arise [49]:

1. **Invalid comment** - When the code is changed correctly but the comment remains outdated with respect to the code. This often leads to confusion and bug introduction in the subsequent versions of the software [50].

2. **Invalid code** - When the comment is up-to-date but the developer does not follow the instructions written in a comment and introduces bugs by writing incoherent code [50].

The designed solution for fragile comments detection deals with only first type of inconsistencies, i.e. *invalid comments*. Because the tool works by analyzing the code when it is being changed, instead of analyzing the history of changes made in a system, the latter type of inconsistencies become irrelevant and are out of scope.

Natural Language Processing: Natural Language Processing refers to a field of research that studies how a computer can understand and generate language automatically [30]. A plethora of research exists on understanding natural language, beginning with the lowest-level analysis technique called morphotactics (analyzing and understanding the smallest unit of language called morphemes) to higher-level techniques used for sentiment analysis of sentences or documents. Our solution to the fragile comments problem uses two basic NLP techniques called *POS tagging* and *lemmatization* to analyze the code comments.

POS tagging

In the context of natural language, part-of-speech (POS) tagging can be described as the syntactic parsing of the words to label them with their syntactic roles [7]. Examples of POS tags include noun (NN), plural noun (NNS), verb (VBZ) etc. The labels like "NN" are standard labels used by the common POS taggers available for parsing natural language text [3, 40]. Figure 2.1 shows an example of a sentence with POS tags labeled by StanfordCoreNLP [31].

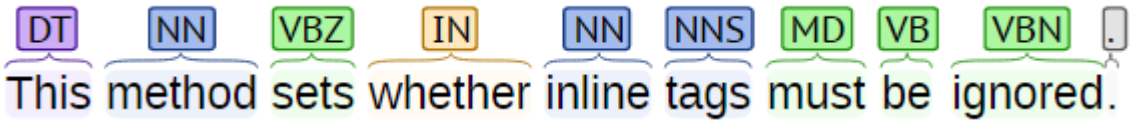

Figure 2.1 – Example of POS tagging for a complete sentence.

These tags demonstrate how the words in a sentence are related to each other. In our solution to the problem of fragile comment detection, we use the StanfordCoreNLP library [31] to tag the words with their parts-of-speech labels. There are many ways of tagging the words with POS labels i.e. feeding the words to a POS tagger as a single unit, a phrase or a full sentence. The details of how POS tagging is used are provided in Chapter 4.

Inflectional Morphology (Lemmatization)

Inflectional morphology can be defined as the study of different forms of a word that can change its grammatical function e.g., the words *walk* or *walked* have the same root *walk* but differ in their grammatical function due to different suffixes.

Lemmatization is the process of removing the inflectional forms of a word to retrieve the complete root i.e., the dictionary form of a word [41]. Our solution to the problem of fragile comment detection uses StanfordCoreNLP lemmatizer to perform lemmatization of words appearing in both comments and identifiers, details of which are given in Chapter 4.

Source Code Elements:

This section defines the important source code elements involved in the detection of fragile comments.

Types of Comments

This thesis divides the comments into three different categories based on their structure and writing specifications. The three categories are block comments, single-line comments, and Javadoc

Table 2.1 – Types of comments

Type of comments	**Format**	**Description**

Javadoc Comment	`/** text...`
	`@tag ...`
	`@tag ...*/`
		These are multi-line comments with special-purpose tags known as Javadoc tags [22] like @deprecated, @params, @return, @author, @see, @link etc.

Block Comment	`/* text...*/`	Block comments are multi-line free text comments i.e., these comments do not contain any tags like Javadoc.
Single-line Comment	`//`	Single-line comments are written on the same line or a line above any variable or decision statement.

comments. The format used for writing each type of comment is shown in Table 2.1.

A clear distinction between these three types of comments helps in understanding their different roles in the source code e.g. *single-line* comments are mostly used inside a method declaration's body in contrast to the *Javadoc* comments that almost never appear inside a method declaration's body. This distinction also helps in designing separate analysis techniques based on the comment categories. The analysis of block and line comments is straightforward because of their simple writing format that does not involve any specific-purpose tags, whereas Javadoc documentation comments need to be analyzed with a special focus on the tags contained in these comments.

Javadoc documentation comments encompass numerous types of special-purpose tags, such as - `@param`,`@return`, `@link`,`@see`, `@value`. Each tag performs a special function which helps in creating easily manageable and presentable documentation. This thesis discusses the details about

Table 2.2 – Identifier Categories

Identifier Category	**Description**
Types	includes the identifiers of classes, interfaces, annotated types and enums.
Methods	includes the identifiers of methods and constructors.
Fields	includes the identifiers of field variables and enum variable declarations.
Local Variables	includes the identifiers of locally defined variables inside methods and formal parameters.

different uses of these tags in the following Sections - Preprocessing §4.3 and Fragility detection §5.1.

Identifiers and their categories

An identifier is the name given to any code element present in the source code and acts as a reference to the code element in a program. The most commonly adopted convention for writing identifiers in Java is *camelCase* [5]. In spite of the wide popularity of this convention, the camelCase format introduces challenges in terms of identifier spitting and its analysis. The challenges of splitting the identifiers written in camelCase and the corresponding solutions designed are described in the later Section §4.3.

To solve the problem of fragile comments, we divide the identifiers into four broad categories based on the type of code elements i.e. *Types*, *Methods*, *Fields* and *Local Variables*. Table 2.2 illustrates the different types of code elements covered under these four categories. This categorical division of identifiers is the underlying basis of our proposed approach as these categories play an important role in designing the fragility detection rules specific to a single category.

Challenges in Comment Analysis

Comments are comprised of text written in natural language, but the analysis of these code comments is not as straightforward as analyzing plain *English* text [60] because English text follows certain grammatical rules, whereas comments are generally written in an unstructured manner [25]. Developers do not always prioritize the grammar and completeness of comments, since their main goal is to explain the intent of a code element by writing non-verbose and concise comments.

Language models used in various NLP components are trained using English language text i.e., grammatically correct and complete sentences. This does not guarantee good performance on source code comments because comments are usually composed of incomplete sentences i.e. short phrases and, at many times, are often grammatically incorrect [25]. For example, a POS tagger trained on natural language text expects the presence of a "noun", i.e. "subject" in the sentence, followed by a "verb" as shown in Figure 2.1. A comparative review of various POS taggers' performance, with respect to code comments, ascertains that these POS taggers do not generalize well on the code comment's text [38]. Figure 2.2 illustrates the how the POS tagger trained on natural language text incorrectly labels "sets" as a plural noun (NNS) in the absence of the word "method" i.e. subject of the sentence.

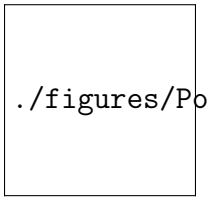

Figure 2.2 – Example of POS tagging a comment.

Additionally, the meanings of English words in the domain of programming languages can differ from those used in natural language. For example, words like "node", "link" and "buffer" have different meanings in the context of programming languages, which makes it difficult to use the existing NLP lexical database called *WordNet* [32]. *WordNet* offers numerous functionalities

for text analysis by providing lists of word synonyms, antonymns, hyponyms etc. These functionalities are essential to capture the semantics of words used in the comments for analysis tasks and, unfortunately, are not suitable for identifying semantically similar words in the domain of software engineering [46]. A few software-specific lexical databases exist, however, they are not accessible in the public domain, which limits the scope of text analysis approaches considerably [21, 53, 59].

2.2 Related Work

The work done on mitigating the problem of inconsistencies between comments and code can be split into three categories. One category discusses the preliminary attempts made at *comment-aware refactoring*. Another category presents the research work that has specifically targeted the detection of inconsistencies between code and comments. The last category includes approaches to obviate the need for consistency maintenance by *generating comments automatically*.

2.2.1 Comment-Aware Refactoring

A number of early proposals to deal with comments during refactoring have focused on the problem of retaining the comments at their proper location in a declaration element's abstract syntax tree (AST), and to preserve their indentation [16, 43]. In particular, Sommerlad et al. [43] built a comment mapper to keep the comments linked with code elements in ASTs because the ASTs generated by a language's parser do not contain specific nodes to represent comments. Existing refactoring tools, like Eclipse's, use similar technique to keep the comments linked to the appropriate code elements. However, these approaches neglect the possible inconsistencies introduced between the modified source code and existing comments. Instead, the main objective of these approaches is to preserve indentation and location of existing comments.

Eclipse [11] also comes with an in-comment text replacement feature, which we call *Eccore*

and describe in the later chapters of this thesis. *Eccore* supports *comment refactoring* to a certain extent, however, *Eccore* only detects and replaces exact matches of identifiers for only two categories of code-elements, i.e. *types* and *fields*, and therefore, does not support name replacement for the *methods* and *local variables*. Our solution to the problem of fragile comments is designed to detect fragile comments for all types of code elements. In addition to the exact matches detected by Eccore, our solution is designed to detect phrases that involve multiple non-contiguous tokens in comments.

2.2.2 Inconsistency Detection

Tan et al. proposed a technique to automatically extract program rules and apply them to detect inconsistencies related to locking mechanisms in the source code [50]. They also devised an approach to extract information from comments to detect inconsistencies in source code related to the specific sets of programming concepts like memory allocation and synchronization [51]. Apart from detecting inconsistencies related to specifically targeted programming concepts, various approaches have been devised to keep the source code of methods consistent with comments. *@TComment* is a technique that detects inconsistencies between a method's parameters' tolerance of null values and its related Javadoc comments [52]. This approach is however constrained to Javadoc comments containing information about a method's parameters. Zhou et al. devised an approach, similar to *@TComment* in terms of its application to method's parameter constraints and exception throwing declarations, that detects inconsistencies between API documentation and its source code by extracting documentation from Javadoc comments, analyzing documentation directives and performing a static analysis of the code of methods [61].

Corazza et al. investigated several projects and devised an approach to detect the coherence between comments and a method's implementation using the Vector Space Model with *tf-idf* term weighting [8]. There are also proposals that focus on specific types of comments for detecting

inconsistencies. For example, Sridhara has developed a tool to detect the fragility of "TODO" comments [44]. All of these techniques focus on a subset of either comment types or programming concepts. In contrast, our solution focuses on detecting the possible inconsistencies produced for all possible types of comments upon renaming an identifier of any type of program element.

2.2.3 Automatic Comment Generation

Automatic comment generation tools offer a different solution to the problem of code-comment consistency maintenance by relieving some of the manual work involved in the creation (and thus maintenance) of comments. *JSummarizer* generates comments for Java classes by using the stereotypes of classes and methods present in the class [33, 34]. Sridhara et al. developed a tool for automatically generating comments for methods based on the rules extracted by code analysis of the method statements [45]. They also proposed a tool to generate parameter comments automatically by using the tool SWUM i.e., Software Word Usage Model built by Emily Hill, to extract the structure and linguistic information about parameters and integrating the information with their previous work on method comments by determining the computational intent of the parameters and its context [20, 47].

Autocomment automatically generates comments for methods by retrieving the information mined from QA websites for code fragments similar to those in the method [57]. Guo et al. propose an approach to automatically generate comments for design patterns by analyzing and predicting the expected usage of design patterns in the source code. [19]. Their work focuses on changes related to design patterns and detects inconsistencies produced due to such changes. Although they share our goal of providing high-quality comments to developers, these approaches apply a different strategy in that they do not take into consideration the pre-existing relation between code and comments. Instead, they automatically generate new comments based on existing artifacts and can presumably replace old comments when applicable. The developer needs to make a conscious

effort in using these tools by devoting extra time, whereas our solution is designed to integrate seamlessly with Eclipse's rename refactoring and produces a list of *fragile comments* when an identifier is renamed.

CHAPTER 3

PROBLEM FORMULATION

The problem of fragile comment detection is cast in the context of a *software project* which comprises a number of *program elements* and a number of *comment units*. This chapter describes the concepts involved in problem formulation, provides a systematic and organized formulation of the problem, and sets the scope of the fragile comment detection approach.

3.1 Concept Description

The problem definition is comprised of five main components - program element, declaration, comment unit, phrase and fragile comment.

Program Element: A *program element* is any element that can be defined in a software program. In this thesis, the approach only takes into account the program elements that can be explicitly named. Consequently, in the problem formulation a program element has a corresponding *declaration* and *identifier*.

Declaration: The *declaration* is the program text that defines the program element, whereas the *identifier* is the part of the declaration that names the element. For example, the declaration shown below is of type method and its identifier is `getListeners`.

```
public Iterable<FileAlterationListener> getListeners();
```

Declarations fall into different categories based on the type of element being defined (e.g., class, method, local variable). These categories are same as the categories of identifiers presented in Table 2.2.

Comment Unit: A *comment unit* is any bounded unit of text considered to be comments according to the syntax of a programming language. Javadoc, block and line comments are the most common types of comment units. A comment unit needs to be contiguous if it is a block comment. For example, as shown in Table 2.1, a Javadoc comment starts with the symbol– $/**$ and ends with the symbol– $*/$. All the sentences in one comment unit needs to be contiguous and should be contained inside the start and end symbols. In the remainder of this thesis, the comment units are simply referred to as *comments*.

Phrase: Given a comment unit, a *phrase* is a subset of the comment unit comprising any coherent set of characters. A phrase can consist of a single word or multiple words taken as a subset of the comment. Note that, when necessary, a distinction will be made between a phrase and a phrase's text. This distinction is necessary when comments have multiple occurrences of some text of interest.

Fragile Comment: If a phrase refers to an element, it is considered that the phrase is at risk of being invalidated if the identifier is renamed, and it is deemed *fragile* with respect to this identifier. By extension, a comment is considered *fragile* with respect to an identifier if it contains at least one fragile phrase. Figure 3.1 showing a comment unit containing the phrase "Gets listeners" which will become fragile on renaming the declaration `getListeners`.

It is important to note that, in practice, the decision of whether a phrase refers to an element can require human interpretation. For example, in a system that comprises the declaration of class `UniqueBuffer`, a comment such as "the UniqueBuffer class" can be directly linked to the `UniqueBuffer` program class if there is only one such declaration. However, a comment such

```
/**
 * Gets the listeners registered in file system.
 * @return The file system listeners
*/
public Iterable<FileAlterationListener> getListeners() {
```

Figure 3.1 – Example of a fragile comment.

as “the buffer is unique...” may refer to a `buffer` identifier of type field or class `UniqueBuffer` depending on the context.

3.2 Problem Formulation

The problem of fragile comment detection is formulated with the help of two main relations that are described below.

refersTo(element) Relation: A *phrase* in a comment **refers to** a program element if an informed developer can determine that the phrase purposefully and specifically refers to the element. This concept is formalized as the refersTo relation.

```
private Buffer buffer;

/** Reloads the existing buffer. */
public void reload() {
   if(buffer != null) { buffer.load(); }}
```

Figure 3.2 – Examples of the refersTo relation

For example, in Figure 3.2, the phrase “buffer” in the comment block refers to the field `buffer` of the same class. In this case, the phrase “buffer” in the comment can be said to *refer to* the field `buffer` where `buffer` is the program element in context.

Conceptually, tuples $\langle \text{phrase}, \text{element} \rangle$ that are members of the *refersTo* relation, fall into the following three categories - lexical, fuzzy and semantic match. In this section, these categories are introduced in terms of the match between a phrase, a phrase's text and an element. A detailed explanation of these categories is provided later in the Section §5.2.

a) Lexical match

A match is considered to be lexical when the phrase is the same as the text of a declaration's identifier, with some tolerance for minor variations (e.g., case sensitivity and plurality). In the case of lexical matches, the phrase is generally a compound unit (without spaces) appearing in the comments written in the camelCase format. The only exception is the cases with phrases composed of single term (word), explained in the later section §5.2. It is important to note that lexical matches do not necessarily imply a refersTo relation because of synonymy; it is possible that a comment mentions an identifier that is shared by multiple program elements, or simply refers to a general concept after which an identifier is named (e.g., "file").

Fuzzy lexical match

A match is fuzzy if the phrase is the same as the declaration's identifier, but with a tolerance factor for small differences owing to misspelled words and typographical errors. It is important to note that fuzzy lexical matches do not necessarily imply a refersTo relation because of both synonymy and approximation (the case where a phrase and an identifier are erroneously determined to be "similar enough").

b) Semantic match

The phrase, consisting of two or more words, semantically matches a program element if the most likely interpretation of the phrase by an expert is that it refers to the element. For pragmatic reasons, the semantic matches are defined as the class of matches that are semantic *without* being

also lexical or fuzzy.

matches(element) Relation: The refersTo relation requires human judgment to select a set of true fragile phrases from the pool of possibly fragile phrases. The task is accomplished by using an automatic approach that approximates the output closer to true fragile comments. To distinguish between phrases that truly refer to an element and phrases estimated to refer to an element, a new relation called **matches** is defined which contains a tuple $\langle \text{phrase}, \text{element} \rangle$ if the corresponding algorithm estimates that the phrase refers to the element.

3.3 Problem Definition

The proposed approach is therefore an implementation of the **matches** relation, and its performance can be measured on a per element basis. Given an element e, let **refersTo**(**e**) be the set of all phrases that refer to this element, and let **matches**(**e**) be the set of all phrases estimated to refer to this element. The true set of fragile phrases for e is thus refersTo(e) and a solution instance given by an algorithm is matches(e). With these definitions, the standard performance measures of precision $P(e)$ and recall $R(e)$ for an implementation of matches can be easily derived as below:

$$P(e) \equiv \frac{|\text{matches(e)} \cap \text{refersTo(e)}|}{|\text{matches(e)}|}$$

$$R(e) \equiv \frac{|\text{matches(e)} \cap \text{refersTo(e)}|}{|\text{refersTo(e)}|}$$

Although precision can be measured accurately, a major obstacle to computing recall is that in the general case the extent of refersTo(e) is not known and can only be approximated.

3.4 Scope

The scope of our solution to the fragile comments problem includes the source code changes made by using an automatic refactoring tool only. Furthermore, the scope is limited to rename refactoring activity because automatic refactoring tools are seldom used by developers except for *Rename and Extract method* refactorings which constitute the maximum proportion of the usage of automatic refactoring tools [35, 36, 55]. Currently, we implemented the approach for the Java programming language, but in light of the fundamental similarities between all object-oriented programming languages, we suspect that adaptation to other similar programming languages would be straightforward.

To facilitate future adaptations, the approach is described in a language-independent manner to the extent possible, and the Java-specific implementation details are furnished whenever applicable. However, this thesis does not make any formal claim about the potential ease with which the approach can be adapted to other programming languages. We have built a tool to support fragile comment detection in Java in the form of an Eclipse plug-in named Fraco. Note that, this thesis is an extension of our work that was accepted at the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).

CHAPTER 4
LINKING AND PREPROCESSING

The proposed approach is defined in terms of a given program element *declaration* i.e., all the rules and pre-processing methods are explained by taking a single program element into consideration.

The process of detecting fragile comments is divided into four conceptual phases. The first is to *detect* all the comments and *link* them to the input declaration (§4.1). We then *filter out* the inapplicable comments based on the program's scoping rules (§4.2). The remaining comments and the identifier of the input program element are then *preprocessed* for textual analysis (§4.3). In the final phase, various *matching rules* are applied to the resulting data obtained at the end of pre-processing phase (§5.1–§5.3). This chapter presents the first three phases of the approach and the matching rules phase is explained in the following chapter.

4.1 Linking Code with Comments

Intuitively, comments that are located meaningfully "close" to a declaration should be treated differently from general comments in the program. This intuition is captured with the concept of comment *locality*. For a given declaration, the comments in a system's source code can thus be divided into two categories: *local* and *global*.

Local Comments: A comment is local if it is found in the proximity of the declaration of a program element. The local comments are further divided into two categories based on their specific location in the code - *header* and *inner* comment. To qualify as a local comment for a declaration, a comment must either be a *header* for the declaration, or be lexically located within the declaration (which is referred to as an **inner** comment).

Header Comment

A comment is considered a declaration's header if it is located immediately above the declaration (without any consideration of white space in between). A declaration can have zero or one header comment. Note that this definition of header comment is different from the pre-existing definition of header comments used in the past research by Steidl et al. [48]. They associate the *header* comment to only class level program elements whereas in this thesis we describe it as a header comment for all types of program elements.

Inner Comment

Inner comments are the comments found inside the body of a program element. Only declarations that have a lexical *body* can have inner comments which includes *classes* and *methods*, but excludes *fields* and *local variables*. The relation between an inner comment and its corresponding declaration is transitive i.e., the inner comments for a method or inner class are also considered inner comments for the declaring class.

Global Comments: For a given declaration, all comments that do not qualify as local comments automatically fall into the *global* category. For example, given a class declaration, all the comments that appear within or above other classes are considered global with respect to the input declaration.

4.2 Scope Based Filtering of Comments

In practice, the problem-space of searching for fragile comments can be narrowed down if one observes that the scoping rules of the programming language greatly affect the likelihood that a given comment may or may not contain a phrase that refers to a given declaration.

For example, in a realistic Java code base, it would be surprising to see an in-line comment inside a method refer to a local variable defined in a different method. So technically, a local variable inside a method is out of scope for all the other program elements declared outside of the method body. This source-code level concept of *scope*, supported by all the existing compilers, is captured by defining the function **applies** which takes as input a declaration and all the comments for a program, and returns the subset of the comments where the declaration can be expected to be visible according to the rules of the language.

The implementation of the **applies** function can be reduced to a lookup in Table 4.1. For an element declaration, the applies function can be computed by inspecting the access modifier of the declared element and its parent type (when applicable), looking up the corresponding scope, and then returning all comments linked to a declaration within the same scope. For example, a private type with no parent type maps to the *class* scope, so applies would return all the comments in the same class. The scope for all local variables (including formal parameters) is the *method* scope, which includes only comments linked to the variable's declaring method.

The precise definition of the applies function is language-dependent and must take into account both the scoping rules of the language and practical knowledge of common commenting practices for this language. The function is implemented for Java based on the *Java Language Specifications, Java SE 8 Edition* [18].

Table 4.1 – Scoping rules for the applies function

Element	Parent Type	Scope
Types		
public	public	global
public	default or protected	package
public	private	parent class
private	any type	parent class
protected	private	parent class
protected	public or default or protected	package
public	None	global
private	None	class
protected	None	package
default	None	package
Methods		
public	public	global
public	private	parent class
public	default or protected	package
protected	public or default or protected	package
protected	private	parent class
default	public or default or protected	package
default	private	class
private	any type	class
Fields		
public	private	parent class

Element	Parent Type	Scope
public	default or protected	package
public	public	global
private	any type	class
protected	private	parent class
protected	default or protected or public	package

4.3 Preprocessing

To analyze the natural language text, basic preprocessing is required to apply any matching rules for fragility detection. Therefore, both identifiers and comments must be preprocessed before applying the matching rules. The preprocessing steps are explained in text as well as presented in the form of an algorithm (Algorithm 1). The preprocessing of identifiers and comments is similar with minor differences and for clarity purpose, this section details them separately.

Preprocessing of Identifiers This section describes the step-by-step preprocessing of an identifier. Conceptually, an identifier consists either of a single term, or of multiple terms that can be distinguished through typographical conventions.

Splitting the identifier

The first step in preprocessing identifiers is to split them into terms. In our Java implementation, an identifier is split using camel casing rules, with an additional rule to preserve acronyms. For example, identifier "ASTParser" will be split into two terms, i.e. "AST" and "Parser". The splitting is achieved with the help of custom designed regular expressions shown below:

```
(?<=[a-z|$|\_])(?=[A-Z|0-9|$|\_])|(?<=[A-Z])((?=[A-Z][a-z])|(?=[/_])|(?=[0-9]))|(?<=[0-9|$|\
    _])(?=[A-Z|a-z|$|\_])
```

POS tagging and Lemmatization

After an identifier is split into two or more terms, each term is tagged with its part-of-speech (POS) tag and its lemma (word root left after removing inflectional suffixes) is identified. POS tagging and lemmatization are two common Natural Language Processing techniques used for text analysis and can assist with text searching tasks.

A POS tag is a label that is associated with a word to indicate its syntactic function (generally in a sentence, but also in a sentence fragment, such as an identifier). For example, in the identifier `addListener`, tagging a single word at a time, the term `add` would be tagged as a verb and the term `listener` as a noun. Fraco uses the POS tagger of Stanford Core NLP library [31] to perform POS-tagging.

Comparing lemmas (inflectional forms) instead of original words can help pave over non-essential differences such as use of the singular or plural form of a word, or different conjugations of a verb. For lemmatization, Fraco uses Stanford Core NLP library's Lemmatizer. The preprocessing phase generates two dictionaries as output containing ⟨term,POS-tag⟩ tuples and ⟨term,lemma⟩ tuples respectively for each term found in the identifier. The whole procedure of preprocessing identifiers is shown in Algorithm 1.

Preprocessing Comments In the preprocessing phase, first, a comment is split into sentences and then into individual units called tokens. We then remove the stopwords i.e. the words carrying no relevant information for text analysis task from the set of tokens obtained after tokenization. The stopwords-filtered tokens are then lemmatized to obtain their inflectional forms.

Table 4.2 – Regular Expressions used in preprocessing of comments and identifiers

Purpose	**Regular Expression**
Split a comment to sentences	`([;|!|?|[<>]|[/*]])|(?<=[a-z])(?:[.][ ])`
Split sentences to tokens	`((?<=%1$s)|(?=%1$s))` and the delimiters are `[â-zA-Z_0-9$]+`
Detect compound term identifiers	`(([_$a-z])([A-Za-z0-9_$]+)|` `([A-Za-z]+))([A-Z][a-z0-9_$]+)|([A-Z]+)`

Splitting the comments

The first step in preprocessing comments is splitting them into sentences using rules based on the regular-expressions designed as a part of the approach shown in Table 4.2. Standard punctuation-based algorithms do not work well for comments because of the common presence of source-code elements that include the punctuation. For the purpose of sentence-splitting, block and in-line comments are treated differently. We split the Javadoc block comments into sentences based on the list of custom devised delimiters that includes period, comma, semi-colon and angle brackets, as shown in row 1 of Table 4.2. Our sentence splitting algorithm avoids using characters that are legal to use for Java identifiers (e.g., underscore) to preserve any identifiers present in the comments and split in-line comments based on *periods* and *commas*.

Further, we split the sentences into *tokens* using the regular expression shown in Table 4.2. Finally, tokens that are detected (through the regular expression shown in Table 4.2) to be compound code terms, i.e. the tokens following camelCase conventions (such as `addFigure`), and are further split by applying the same preprocessing rules used for identifiers. However, both the split and unsplit version of the token is kept because some matching rules work with the original compound term (see section §5.1).

a) Stopword removal and Lemmatization

After tokenization, we remove the stop words (e.g., “the”, “an”) from the list of tokens obtained. The default stopwords list used by NLP tools contains a lot of words like “before”,“after” etc. that are relevant in the context of software engineering. Therefore, we devised a customized list of stopwords by working backwards on the conventional stopwords list using by natural language processing tools and is used while preprocessing the comments. Below are the words treated as irrelevant for the comment analysis task:

```
is, a, this, that, on, in, the, of, are, and, or
```

In the final step, lemmatization and POS-tagging is applied to the tokens in the list. Contrary to the identifier POS-tagging, the comments are tagged with the part-of-speech label on a sentence basis, i.e. a whole sentence is assigned the part-of-speech tags at once.

Algorithm 1 Preprocessing comment and identifier

Input: Comment (C)

Output: $Lmap \leftarrow \{\}$ // a list of $\langle \text{token}, \text{lemma} \rangle$ values

Output: $Pmap \leftarrow \{\}$ // a list of $\langle \text{token}, \text{POStag} \rangle$ values

```
1:  procedure PREPROCESSCOMMENT(C)
2:      Split C into sentences
3:      for each sentence S do
4:          Split S into tokens
5:          Remove stop words from S.
6:          for each token T in S do
7:              if T == (non − alphabetical) then
8:                  continue
9:              else
10:                 Lemmatize T and add to Lmap
11:                 Tag T with POS and add to Pmap
12:             end if
13:         end for
14:     end for
15: end procedure
```

Input: Identifier(I)

Output: $Lmap \leftarrow \{\}$ // a list of $\langle \text{token}, \text{lemma} \rangle$ values

Output: $Pmap \leftarrow \{\}$ // a list of $\langle \text{token}, \text{POStag} \rangle$ values

```
1:  procedure PREPROCESSIDENTIFIER(I)
2:      Split I into terms
3:      for each term T in I do
4:          Lemmatize T and add to Lmap
5:          Tag T with POS and add to Pmap
6:      end for
7:  end procedure
```

CHAPTER 5

DETECTING AND RESOLVING FRAGILITY

This chapter outlines the various fragility detection rules designed to match fragile phrases in comments. It describes the implementation details of all the rules and finally, presents the techniques designed for the automatic resolution of fragile phrases.

5.1 Overview of Matching Rules

The matches(e) function is implemented through a number of *matching rules*. The matching rules can be roughly organized into two categories: (mostly) *lexical* rules that target the text of phrases and identifiers, (more) *semantic* rules that seek to match comments and identifiers that refer to the same thing despite having different spelling or writing format. The rules are organized in these categories to facilitate the presentation, but it should be noted that most matching rules are neither purely lexical nor semantic, but constitute a combination of features. Given an element *e*, the approach returns the union of the results obtained by applying the different matching rules.

5.2 Lexical Matching Rules

The assumption behind lexical matching is that if a phrase has the same text as an element's identifier, the phrase may refer to identifier. In practice, however, returning all the instances of phrases whose text matches an identifier under consideration produces a deluge of false positives due to

synonymy. Additionally, limiting the search to exact lexical matches misses cases where the name of some identifiers is transformed morphologically (e.g., used in the plural form, such as "receives the `Events`" which refers to the class `Event`). Therefore, we devised a new algorithm for lexical matching of program identifiers that takes into account the type of the identifier, its morphology, and the location of the comment containing the phrase under consideration.

Table 5.1 and 5.2 provide a case-based specification of the algorithm. Each cell in these tables presents the matching variant for one of 14 possible cases determined by the type of identifier, whether the identifier is a single or compound term, and whether the phrase to match is in a local or global comment. The matching rules are expressed as predicates using the binary operators and functions described in Table 5.3.

Table 5.1 – Lexical Matching Rules for Types and Methods.
p refers to the input and i to the identifier of the element under processing. Each non-header cell in the last two rows is referred as $Cell_{r,c}$ where r is the row number with value either 1 or 2 and c is the column number ranging from 1 to 4. The operators are defined in Table 5.3.

Comment Type	**Types**		**Methods**	
	One term	Multiple terms	One term	Multiple terms
Global Comment	$(p \tilde{=} i) \wedge$ $(\text{noun}(p) \vee$ $\text{paren}(p))$	$p \tilde{=} i$	$(p \approx i) \wedge$ $(\text{decl}(i,p) \vee$ $\text{paren}(p))$	$p = i$
Local Comment	$(p \tilde{\approx} i) \wedge$ $(\text{noun}(p))$	$p \tilde{=} i$	$p \approx i$	$p = i$

As it can be observed, more complex rules are necessary to determine the correct matches for

Table 5.2 – Lexical Matching Rules for Fields and Local Variables.
p refers to the input and i to the identifier of the element under processing. Each non-header cell in the last two rows is referred as $Cell_{r,c}$ where r is the row number with value either 1 or 2 and c is the column number ranging from 1 to 3. The operators are defined in Table 5.3

Comment Type	**Fields**		**Locals**
	One term	Multiple terms	
Global Comment	$(p = i) \wedge (\text{upper}(i) \vee \text{decl}(i,p))$	$p = i$	$p = i$
Local Comment	$p \approx i$	$p = i$	$p = i$

single term identifiers (e.g., `add`, `copy`) due to their common use in program text which creates a massive amount of ambiguity. We provide an example to illustrate the use of these matching rules: assuming the problem case is to determine the comments that are fragile with respect to the declaration of a method `copy` declared in class `Interval` of project `JFlex` and that the comments in Figure 5.1 and Figure 5.2 are under consideration. Based on the design of the approach, with respect to the declaration shown in Figure 5.1, Fraco considers two cases: first, if the comment is a local comment for the method (i.e., its header block) as shown in Figure 5.1, and another case if the comment is a global comment i.e., not directly associated with the method `copy` declared in class `Interval`.

Considering the local comment shown in Figure 5.1, we select the $Cell_{2,3}$ of Table 5.1 because the element is of type `method` and the identifier is composed of a single term. The rule in $Cell_{2,3}$ specifies that a phrase matches the identifier if $p \approx i$. The only phrase in the comment that validates this predicate is `copy` because `copy` $\approx$ `copy`, so the rule returns the only instance of the string "copy" in the comment.

Table 5.3 – Description of the operators used in the rules of Table 5.1 and 5.2 with positive examples. *When a phrase p is used as input, it is assumed that its comment-context (the rest of the text in the comment) is also available to the operator. This context is represented as C in the examples. When an identifier i is used as input, it is assumed that its declaring element is also available to the operator. This context is represented as d in the examples.*

Operator	**Description**	**Positive Example**
$=$	Case-sensitive match	`Tag` $=$ `Tag`
$\approx$	Case-insensitive match	`Tag` $\approx$ `tag`
$\tilde{=}$	Case sensitive match that tolerates the plural form	`Tag` $\tilde{=}$ `Tags`
$\approx$	Case-insensitive match that tolerates the plural form	`tag` $\approx$ `Tags`
noun(p)	True if the POS tag of p is a noun or proper noun (sensitive to the language model used in the POS tagger)	noun(`tag`)
paren(p)	True if p is immediately followed by the opening and closing round parentheses having the number of intermediate tokens equal to the count of declaration's arguments, if any.	paren(`read`) where $C =$ `...read(File) ...` and $d =$`copy(String file)`
decl(i, p)	True 1) If p is present in the i's declaring class. OR 2) If first condition is false, check if the simple name of i's declaring class can be found in the same comment as p, without considering case.	decl(`copy`,`copy`) where $C =$ `...copy this interval...` and $d =$ `Interval`
upper(i)	True if i is all in upper case characters	upper(`SORTED`)

```
/**
 * Make a copy of this interval.
 * @return the copy
 */
 public Interval copy() {
```

Figure 5.1 – Example from the project JFlex's source code showing the method "copy" from class Interval.java.

However, in the case where the comment is global with respect to the element `copy` of class `Interval`, the rule of $Cell_{1,3}$ of Table 5.1 applies. An example of global comment is shown in Figure 5.2. The first part of the rule states that any matched phrase must be the same as the identifier (insensitive of case), whereas the second part offers two options to detect a match. The first options checks the string that corresponds to the method's declaring class's identifier (i.e., `Interval`) must also be in the comment unit given the matched phrase and program element declaration are not located in the same class. Second option checks if the token immediately following the phrase is an opening round parenthesis followed by a closing round parenthesis having the number of intermediate tokens equal to the count of declaration's arguments, if any. Thus, the rule matches the instance of string "copy" because of its verbatim similarity with the declaration in question, i.e., the method `copy` of class `Interval`. Because the comment neither mentions the declaring class name i.e., "interval" nor has any instance of string "copy" followed by empty pair of round parentheses such as `copy()`, the rule returns an empty set for the comment.

```
/**
 * Return a (deep) copy of this char set
 * @return the copy
*/
 public IntCharSet copy() {
```

Figure 5.2 – Example from the project JFlex's source code showing the method "copy" from class IntCharSet.java.

5.3 Semantic Matching Rules

The lexical rules match identical or near-identical terms pairwise. In many situations, a set of keywords in comment's text can refer, as a whole, to an identifier. We have designed a new matching rule to capture this situation called the *semantic* matching rule. Because this rule is intended to match "units of meaning" in comments that are likely to refer to an identifier, it is referred as "semantic" to distinguish them from the lower-level text matching rules described in the previous section. In practice, the developers tend to describe a program element's identifier using full sentences or phrases appearing only in the *header* comments. Thus, the semantic matching rule is applicable only to the **local comments**.

We apply the semantic rule on the comments taking one sentence at a time. Given all the (non-stopword) lemmas of an identifier obtained as described in §4.3, our approach looks for identical lemmas in the sentence of the comment unit. If all lemmas of an identifier are found in the sentence, the corresponding lemmas in the comment are returned as the fragile phrase. Note that a comment may contain multiple fragile phrases if it is composed of more than one sentence.

For instance, as shown in Figure 5.3, after preprocessing the comment and identifier which includes the removal of stop words like "and", the sentence contains 4 token-lemmas matching the 4 term-lemmas of the identifier i.e., "parse", "javadoc", "print" and "tree".

```
/**
 * Parses block comment as javadoc and prints its tree.
 * @param node block comment begin
 * @return string javadoc tree
*/
private String parseAndPrintJavadocTree(DetailAST node)
```

Figure 5.3 – Example from Checkstyle's source code.

The implementation of this semantic matching heuristic must take into account some of the idiosyncrasies of common commenting practices that depend on the type of identifier being matched. *Methods:* There are two special cases for method identifiers:

1. If a method's identifier has the word *"get"* as its first term, the term can be matched with both the word *get* and the tag *@return* or lemma *return* in a comment. The *@return* tag signifies the return of a value which aligns with the main functionality of the getter methods and therefore justifies the use of `@return` in place of the word `get`. For example, in the code below the phrase ⟨`last`,`node`⟩ would be returned as fragile:

```
/**
 * @return the last node that was selected,
 * or null if there are no Nodes selected.
 */
public Node getLastNode()
```

2. If a method's identifier starts with *"is"* and has a local comment of type Javadoc, for instance `isSelected`, the word *"is"* is not matched and instead the rule verifies the presence of the word "true" or "false" immediately following the `@return` tag.[1] For example, in the code below the phrases -⟨`member`,`Enum`⟩ and ⟨`enum`,`member`⟩ on line 2 and 4 respectively would be returned as fragile:

```
/**
 * Checks if current AST node is member of Enum.
 * @param ast AST node
 * @return true if it is an enum member
 */
private static boolean isEnumMember(DetailAST ast)
```

Local Variables: In one special case for matching formal parameters, if the comment is a *Javadoc* comment, the parameter's name only is matched against the tokens present in the text related to its `@param` tag, leaving out the rest of the comment.

[1] A performance-motivated proxy for verifying that the method returns a boolean.

5.4 Language Specific Rules

Apart from the matching rules discussed in the above sections, there are some common rules applied for detecting fragile phrases based on the language specifications (Java) or *Javadoc documentation* specifications that are described below.

Renaming Overridden Methods: In this section, we provide the description of two different categories of overridden methods and how Eclipse's refactoring tool handles renaming of such methods. The declarations of type `method` can be overridden in the Java language. These overridden methods are further divided into two categories:

1. **Binary type methods** - The methods that belong to a ".class" file which does not exist in the source code as a part of the Java model are called binary methods. These type of declarations are not allowed to be renamed by Eclipse's refactoring tool as renaming such methods can break compatibility with the corresponding API in which they are initially declared. Therefore, these are out of the scope for our approach.

2. **Non-binary type methods** - The original declaration of these methods can be found in the source code of a system being refactored using rename refactoring. However, renaming any such overridden method will automatically rename all the overridden declarations of that method present in the source code. Therefore, our approach detects fragile phrases in all the comments containing the matching phrases related to one such method irrespective of the method declaration in context.

Since Fraco is built on top of the Eclipse's refactoring tool, it is vital to understand the restrictions imposed by Eclipse's tool and the services it offers with respect to every type of program element. Moreover, the clarity about how non-binary type of methods are renamed is crucial for evaluating the approach because there exist many instances of identifier references in comment's

text that can cause ambiguity due to the cross-referenced fully qualified names of overridden methods. Figure 5.4 shows an example of a non-binary overridden method. Upon renaming the method, Fraco will detect a fragile phrase in the corresponding comment unit shown in the example but the fully qualified name used in the comment creates ambiguity i.e. should we consider this reference as fragile with respect to the method in context or not. Considering the behavior of Eclipse's refactoring tool, all the non-binary methods will be automatically renamed which makes this phrase fragile even when it is referencing some other declaration of this overridden method.

```
/*
 * (non-Javadoc)
 * @see org.springframework.data.convert.EntityConverter#getMappingContext()
*/
@Override
public RedisMappingContext getMappingContext() {
  return this.mappingContext;
}
```

Figure 5.4 – Example from the project Spring-Redis-Data's source code showing the method "getMappingContext" from class MappingRedisConverter.java.

Javadoc Tag References: Javadoc comments are composed of multiple special-purpose tags. There are some tags that can be used to reference program elements, which are updated automatically when the corresponding identifiers are renamed. These tags are: "@link" and "@linkplain". Their main purpose is to provide the cross-references inside comments that can be automatically updated on renaming the program elements. This automatic update of cross-references is a function of Javadoc comments. Therefore, Fraco ignores the identifier references found in the text related to such tags.

5.5 Tool Support

The complete approach is developed as an Eclipse plug-in called Fraco. The plug-in seamlessly integrates with the normal Eclipse-based workflow. The program routine to detect fragile matches is triggered whenever the developer renames an identifier using Eclipse's usual *Rename refactoring* feature. The results i.e., the fragile phrases with respect to the renamed identifier, are reported as Eclipse's *warning markers*, which by default appear in the *Problem View* and as annotations in the gutter (sidebar) of Eclipse's Java editor. From the Problem View, a developer can, as usual, click on a *fragile comment warning* to immediately access the location of the fragile comment detected by the approach. The developer can further click on the warnings both in the gutter (sidebar) or problems view to get resolution options. The resolutions are provided using the quick fix feature of the Eclipse editor.

5.6 Resolving Fragility

Fraco detects the fragile phrases and reports them as warnings in Eclipse's editor. It also provides an option to automatically resolve the detected fragile phrases by replacing them with the new name of the renamed program element. Note that, it does not resolve the fragility automatically during detection process, leaving the ultimate decision of fragility resolution with the developer. This section introduces the different resolution methods offered by Fraco and the resolution techniques designed based on the type of fragile phrases i.e., lexical and semantic.

Resolution Methods: There are two different methods of resolutions offered by Fraco:

1. **Single warning resolution** - The tool resolves only one warning at a time. Here, one warning represents one fragile phrase. Therefore, resolving a single warning related to a comment does not resolve all of the fragile phrases present in that comment if there are more than one.

2. **Multiple warnings resolution**- Fraco gathers information about the warnings related to the specific renaming and resolves all of them at once. Note that this option only resolves the warnings related to a single identifier and does not modify any existing warnings in Eclipse.

Replacing lexical fragile phrases: Fraco can resolve lexically fragile phrases, which includes case-sensitive, case-insensitive and nearly-identical matches (i.e., the matches found by applying the plurality variant of the lexical matches mentioned in Tables 5.1 and 5.2).

Replacing near-lexical fragile phrases: The phrases detected using case-insensitive variant of lexical matching rules are simply replaced with the new identifier without preserving their case-insensitivity attribute. For example, if a class's identifier is renamed from `Property` to `KeyValue`, our approach replaces the case-insensitive fragile phrases such as "property" with the raw form of the new identifier i.e. "KeyValue". The implementation of case-insensitive replacement adds unnecessary complexity considering the fact that case-ignorant replacement does not have any impact on the comprehensibility of a comment. However, the replacement of the phrases detected with the plurality variant of lexical matches is a bit complex in nature.

The plurality attribute of the fragile phrases detected through plurality rule is kept intact during the resolution process. Considering the same example of class `Property`, our approach replaces any fragile instance of this class occurring in plural form, such as "properties", with the plural form of new identifier i.e. "KeyValues".

The plural form of the new identifier is obtained by using a rule-based plurality-conversion algorithm designed as a part of our approach. It is based on the basic English grammar rules to identify the plural form of an English language word e.g., words ending with "sh","ch" etc. will have the suffix "es" in their plural form. This algorithm is then used to replace the detected fragile phrases with an appropriate plural form of the new identifier.

5.7 Replacing semantically fragile phrases:

Resolving the fragile phrases found through semantic rules is a challenging problem because it cannot be resolved by simply replacing the fragile phrases with a new identifier. As semantic phrases are composed of multiple words appearing in a comment's sentence, replacing all of the words in fragile phrases would require a technique to generate correct phrases based on the new identifier. Currently, the text generation for resolving inconsistencies is not in the scope of this thesis. Additionally, it is tricky to design text templates for replacing the old text based on the new identifier alone. The template designing needs to be contextual and should factor in the overall source code of a program element while generating the replacement text. There exists a lot of research on the generation of comments by analyzing source code of an element, but none of them can be directly integrated with the comment refactoring approach presented in this thesis because they all work on the final version of the source code obtained after modifications [33, 34, 45, 47].

5.8 Fuzzy Matching Rules: Discarded

This section describes the implementation of *fuzzy* matching rules and the reason for omitting the *fuzzy* rules from the final version of our approach.

In the case of lexical matching, spelling and typographical errors are a potential cause of false negatives. This issue is mitigated by including a rule that implements *fuzzy* lexical matching. The fuzzy matching applies information retrieval technique called *n-grams* to detect the similarity between two misspelled words [29]. Fraco uses bi-grams i.e., pair-wise comparison of letters in both the identifier and matching phrase. The fuzzy matching algorithm puts following constraints on comparison between two words:

1. The length of both the identifier and the phrase can vary by only one unit i.e., phrase having one character more or less than the identifier is a valid target for comparison.

2. The number of pairs allowed to be different/misspelled are limited by a threshold calculated based upon the length of the identifier e.g., above 80% of the pairs should be same in the matching phrase and identifier having a character length of 10, therefore it can have a maximum of 2 misspelled characters.

3. Both the character pairs and the indexes retrieved from the respective phrase and the identifier strings should match. The fuzzy match algorithm does not pair the first and last letter in a word to avoid detecting two anagrams, i.e., the words generated by rearranging the sequence of characters, as similar. For instance, "slap" and "laps" will have same number of bi-grams if the constraint on index matching is removed.

The phrase under consideration needs to satisfy all the three constraints to be detected as a fuzzy match. We finalized these constraint values after experimenting with various different values of constraint 1 and 2. Decreasing the percentage value of constraint 2 resulted in a large number of false positives whereas increasing the value essentially transformed the fuzzy rule into lexical rule i.e. leaving no room for detection of misspelled words.

However, none of the systems, in the development set of Fraco, contained a single instance of a textual reference to an identifier that was found through fuzzy match alone. For this reason, fuzzy matching rule is not considered to be necessary and has been removed from the final version of the approach.

5.9 Development of the Approach

This section illustrates the step-by-step progression that we followed in designing the final approach. It explains how the approach is improved by starting from the heuristics and finalizing the different scoping and matching rules. Also, it presents the rationale behind various approach-refinement decisions made during the development process.

Designing Heuristics: The author of this thesis manually studied the relationship between the identifiers and comments to create a collection of heuristics needed to detect the fragile comments. Initially, lexical matching was tried without the additional constraints shown in Table 5.1 and without categorizing the comments into *local* and *global*. As one would expect, it created more false positives than correct matches. Then, the concept of proximity between the identifier and a comment was integrated and the distinction is introduced as *local* and *global* comments. This strategy curbed the number of false positives to a great extent but not enough to achieve practical usefulness.

Later, the introduction of new identifier type-sensitive variants using POS tags, case-sensitivity and inclusion of the *parent* identifier in certain cases helped to achieve performance levels that aligned with practical usefulness. In the case of the semantic matching rule, the use of lemmas yielded the desired results (no false positives) on the development set.

Extension using POS-tag variants: Initially, both *lexical* and *semantic* rules were designed to extract more information from POS tags and use that information in comment parsing. For example, the general observation of identifiers indicated that most of the `class` type identifiers are made up of noun terms and most of the `method` type identifiers start with a verb or an adjective. This information was included in the semantic rules by matching both lemmas and POS tags based on these assumptions which were validated by observing the identifiers in the development set used for this approach. Noticing that almost 95% of the time the POS tag of the identifier term was same as the POS tag of the token matched using lemma's rule, this POS tags extension resulted in decreased performance of the tool, in terms of execution time, without a counterbalancing increase in precision or recall. In fact, it was observed that the apriori assumption about the first term of a method identifier of being mostly a verb or an adjective reduced the recall due to numerous instances of invalidation of this assumption in methods such as `yTransform` where `y` is not a verb. Therefore, the final version of this approach incorporates POS tagging in the *lexical* rules only.

Language Features: We designed some new rules, based on the results of ASE study, that are tailored towards the use of some specific Java language features in the problem of fragility detection. These language features, described in detail in §5.4, include the ensured similar treatment of all the comments related to non-binary methods irrespective of the method declaration in context and excluding the binary methods from scope. We added these features to remove the false positive cases registered in an earlier version of our approach due to overridden methods.

CHAPTER 6

EVALUATION STUDY DESIGN

The performance of the proposed tool "Fraco" can be evaluated in terms of the metrics of *precision* and *recall* (see Section 3.3) for a sample of input *identifiers*. The design of the evaluation study comprises five main components: *a)* The selection of projects for the development and test sets; *b)* The *sampling* of identifiers for which to detect fragile phrases; *c)* The creation of a general *benchmark* for these identifiers; *d)* The computation of *baseline results* to help in the interpretation of the results; *e)* The computation of *metrics* for evaluating the performance of the approach.

6.1 Data Sets

We evaluated our approach in two phases: *i)* initial development and evaluation; *ii)* approach refinement and final evaluation. Accordingly, we used two different sets of projects for each evaluation phase. The *development set* is used for the initial evaluation, whereas the *test set* is used to evaluate the final version of the approach. Table 6.1 presents the version and a brief description about the different types of projects used for the development and testing of the approach. We carefully selected these sets of projects to represent the diverse open-source community of software systems.

Development Set: We developed the approach iteratively using six code bases as a preliminary development set, shown in the Table 6.1. These six Java systems are used to evaluate the pre-final version of the approach. The six target systems are moderately-sized and well-commented

(see Table 6.1). While these systems offer a diversity of application domain and open-source communities, their medium size and general-purpose application domain makes it reasonable for investigators to inspect.

Table 6.1 – Description of projects in Development and Evaluation sets of projects

Development Set			**Test Set**		
Project	**Version**	**Description**	**Project**	**Version**	**Description**
Log4j	1.2.17	Logging Utility	Commons-IO	2.5	Utilities library for IO functionality
JUnit	4.12	Unit testing Framework	Mockito	2.8.48	Mocking Framework
Joda time	2.9.6	Date and time library	JFreechart	1.0.18	Chart library
JFlex	1.6.1	Lexer/Scanner Generator	Findbugs	3.1.0	Static code analyzer to detect bugs
Chronicle Map	3.11.0	In-memory key-value store	Apache JMeter	3.2	Load testing tool
Spring Data Redis	1.7.8	Redis data connection configurator	HazelCast	3.7.8	In-memory data grid

Test Set: For the *Test set*, we selected the systems based on various attributes of a project like the project size, the total number of program elements, total number of comments and the density

of comments. The project size refers to the lines of code (LOC), whereas the comment density is measured by dividing the total number of comment units by total number of program elements present in a system. Tables 6.2 and 6.3 show the different metrics calculated for these projects and demonstrates the categorization of these six projects into different categories based on their size and comment density. This selection criteria allows us to obtain a set of projects to cover a variety of projects with different sizes and helps to analyze the impact of comment density on our approach.

Table 6.2 – Selection metrics of evaluation set projects

Project Name	**LOC**	**Comments (C)**	**Identifiers (I)**	**Comment Density (C/I)**
Commons-IO	63410	1718	1136	1.5
Mockito	84134	1152	2026	0.56
JFreechart	302388	12765	7416	1.72
Findbugs	422509	9181	13751	0.67
Apache JMeter	565510	9766	11844	0.82
HazelCast	1133442	7947	15832	0.50

6.2 Sampling

We evaluated the approach using various projects to cover a variety of potential identifier renaming situations. Because of the underlying structure of programs and commenting practices, a naive

Table 6.3 – Categorization of evaluation set projects

Project size (LOC)	**Small (50K-150K)**	**Medium (150K-500K)**	**Large (>500K)**
Comment Density			
Well commented (C/I $\geq$0.8)	Commons-io	JFreeChart	Apache Jmeter
Sparsely Commented (C/I $<$ 0.8)	Mockito	Findbugs	HazelCast

random sampling approach is not appropriate to fairly evaluate Fraco. First, in most of the software projects, only a fraction of identifiers is ever mentioned in the comments. By sampling randomly, any aggregated result would be heavily biased by the underlying prior distribution of identifiers in the comments. Second, the proportion of different identifiers types (e.g., local variables vs. classes) is not uniform and so drawing from the general population of identifiers is likely to lead to a glut of local variables, thus degrading the ability to evaluate the performance of the approach for other identifier types. A final constraint on the sampling is scalability and understandability of the underlying software, given that the resulting benchmark must be created through manual code inspection (see §6.3).

Target Population: The target population of program elements is defined as only the elements that have at least one fragile phrase. In principle, this means all elements $\{e \mid \texttt{refersTo}(e) \neq \emptyset\}$. However, as described in §3.3, `refersTo` can only be estimated with `matches`, which means that the target population is partially defined in terms of the approach itself. In practice, this imperfect and unavoidable situation is mitigated by the high overlap between the output of `refersTo` and `matches`, as discussed in §6.6.

Stratified Random Sampling: We used a stratified random sampling strategy to achieve a diversity of program element types while keeping the size of the data set to a manageable level. Stratified random sampling protects against the selection bias while ensuring that all the classes of interest are covered in a sampled population.

Development Set Sample: To obtain samples from the development set, we randomly selected 100 identifiers from each of the six development set systems, in proportion to the number of elements of each type in the target population (of elements with at least one fragile comment).

Tables 6.4 and 6.5 show the number of identifiers in the sample for each program element type. For each target system (row), the tables indicates the total number of program elements of a given type, followed by the number (in parentheses) of program elements of this type for which at least one fragile phrase was detected. The right column for each element type indicates the number of program elements of that type in the sample. For example, in case of Log4j, 244 classes are detected, of which 116 had a non-empty result for matches, which is 33% of all identifiers across all types $(116 + 139 + 73 + 20)$. For this reason, 33 identifiers for elements of class types are randomly selected to be part of the sample.

When piloting the evaluation on the development set, we discovered that the performance of the semantic rule contribution to the matches relation had very low coverage. In other words, there were relatively few cases where the semantic rule yielded results. For this reason, the semantic rule is evaluated separately using a sample that consists of all different types of elements in each system (e.g. 244 classes of Log4j).

Test Set Sample: The *development set* comprised only 0.05% of the total number of identifiers of type *local* in all the six systems because the local variables were rarely mentioned in the comments leading to an extremely low number of fragile comments related to local variables. In

Table 6.4 – Composition of the **Development Set** Sample - Types and Methods

Project Name	**Class Type**		**Method Type**	
	Total (Pop.)	*Sample*	*Total (Pop.)*	*Sample*
Log4j	244 (116)	33	853 (139)	40
JUnit	218 (96)	54	587 (72)	40
Joda time	227 (118)	32	933 (223)	54
JFlex	71 (36)	28	297 (66)	40
Chronicle Map	265 (68)	42	784 (67)	4
Spring Data Redis	426 (194)	54	1199 (119)	33

order to obtain the sample capturing the substantial number of *local* type elements, we increased the total number of elements selected randomly in the *test set* sample from 100 to 200. Similar to the development set, the *semantic* rule is evaluated using a sample comprising all program elements in each system of the *test set*.

We selected a subset of the test set sample containing 50 elements per system. We refer to the super set with 200 elements as **Test Set A** and the subset containing 50 elements per system as **Test Set B**. The identifiers in *Set B* are selected by re-sampling the already sampled *Set A*. The purpose of *Set B* is to measure the correctness of the benchmark by measuring inter-annotator agreement. Since the available resources did not allow us to engage multiple annotators for the full test set A, we decided to create the subset B that is of manageable size and which helps to mitigate accidental bias that would be introduced by having a single annotator for the other two sets i.e., *development* set and *test set A*.

Similar to the development set, the Tables 6.6 and 6.7 show the distribution of different types

Table 6.5 – Composition of the **Development Set** Sample - Fields and Local Variables.

Project Name	**Field Type**		**Local Variable Type**	
	Total (Pop.)	*Sample*	*Total (Pop.)*	*Sample*
Log4j	640 (73)	21	776 (20)	6
JUnit	144(11)	6	377 (0)	0
Joda time	500 (52)	12	706 (18)	2
JFlex	250 (34)	20	307 (26)	12
Chronicle Map	342 (23)	14	473 (10)	3
Spring Data Redis	624 (50)	13	689 (3)	0

of identifiers in Set A and Set B. The columns *Set A* and *Set B* represent the number of identifiers of a given type selected in the respective sample. Every identifier selected in the samples has at least one fragile comment. The sum of all types of identifiers in Set A is 200 for each system. Similarly, the sum of different types of identifiers in Set B is 50 for each system. For example, the sample selected for *Commons-IO* in Set A has $(67+90+52+1) = 200$ elements where the sample selected in Set B has $(17+23+16+0) = 50$.

6.3 Benchmarks

As a necessary component of the evaluation, benchmarks are created for fragile comments that constitutes a general contribution of this work. We created three different benchmarks for three sets of data - one *development set* and two test sets *Set A and Set B*.

Development Set Benchmark: The benchmark includes, for each element e as reported in Tables 6.4 and 6.5, the full set refersTo(e). The author of this thesis created the benchmark

Table 6.6 – Composition of the **Test Set** Sample - Types and Methods

Project Name	**Class Type**			**Method Type**		
	Total (Pop.)	*Set A*	*Set B*	*Total (Pop.)*	*Set A*	*Set B*
Commons-IO	120 (81)	67	17	373 (110)	90	23
Mockito	445(120)	86	22	817 (133)	95	24
JFreechart	639(283)	29	8	2967(1372)	140	35
Findbugs	1483 (472)	75	19	4797 (496)	79	20
Apache JMeter	1116 (425)	80	20	3854 (333)	63	16
HazelCast	3062 (810)	108	27	5768 (432)	57	14

by manually inspecting the detected fragile comments. For each phrase returned as the results of the matches(e) relation for an element *e*, the researcher made a binary decision as to whether the phrase referred to the element's identifier or not. The validation or invalidation of phrases as fragile is a low subjectivity task given that comments are intended for human consumption and therefore generally not ambiguous *to a human reader*. In addition, the false negatives discovered through the computation of the approximations of recall (R^* & $R^{\cup}$) as described in §6.5 are also included in the benchmark.

Test Set Benchmarks: We created two different benchmarks for the test sets A and B—*Benchmark A* and *Benchmark B* respectively. Similar to the development set benchmark, these two benchmarks include all the elements reported in Tables 6.6 and 6.7. The author of this thesis created **Benchmark A** by manually inspecting each fragile instance reported for every element *e* present in the samples.

On the other hand, the creation of **Benchmark B** involved an external annotator. A first year

Table 6.7 – Composition of the **Test Set** Sample - Fields and Local Variables.

Project Name	**Field Type**			**Local Variable Type**		
	Total (Pop.)	*Set A*	*Set B*	*Total (Pop.)*	*Set A*	*Set B*
Commons-IO	251 (63)	52	16	286 (2)	1	0
Mockito	188(18)	13	3	584 (9)	6	1
JFreechart	1483 (280)	28	7	2280 (31)	3	0
Findbugs	3138 (209)	33	8	4309 (85)	13	3
Apache JMeter	3343 (268)	15	13	3527 (36)	7	2
HazelCast	3558 (242)	32	8	3376 (25)	3	0

Masters student in the school of Computer Science at McGill University helped in the annotation of benchmark B. The external and lead annotator i.e., the author of this thesis, annotated Set B samples separately. Note that the working and implementation details of the approach were not provided to the external annotator, who was only provided with general instructions to understand the task of annotation. Similar to the development set benchmarks, both the annotators made the binary decision about the phrases being fragile or not and also computed the approximations of recall (R^* & $R^{\cup}$) described in §6.5.

6.4 Baseline

Although the performance of the approach can be interpreted in absolute terms, it is interesting to compare it with an existing baseline. Basic lexical matching is technically an option, but as described in §5.9, it performs so poorly that it cannot reasonably qualify as a baseline for this type of work. A more appropriate domain-specific baseline is offered by Eclipse's refactoring tool. Along with refactoring the code, Eclipse's refactoring tool allows the user to select an option to replace

the textual matches present in comments when renaming an identifier. As mentioned earlier, this feature is referred to as Eccore ("Eclipse comment refactoring"). Eccore is only available for *type* and *field* code elements (so not available for *methods* and *local variables*). To use Eccore as a baseline when conducting the evaluation, all of the applicable declarations (types and fields) are programmatically renamed and all textual replacements are considered to be the *fragile phrases*.

6.5 Metrics

We evaluated the approach using the standard metrics of precision and recall. For a given program element e, precision is computed exactly as defined in §3.3. In general, precision measures the degree of absence of false positives, which in the case of Fraco are the phrases falsely reported as fragile. In contrast, recall measures the degree of absence of false negatives. In the case of Fraco, false negatives correspond to the fragile phrases that remain undetected. Recall is not generally possible to compute, since to compute $\text{refersTo}(e)$ one would need to manually inspect the entire source code of a system. For this reason, approximations must be considered as the potential method of measuring the performance of the tool.

The *Eccore* feature is only applicable to class and field identifiers and therefore, the union of the sets of true positives is used for both approaches as the approximation of $\text{refersTo}(e)$ in the denominator of the recall equation in §3.3. This version of recall is denoted as $\textbf{Recall}^{\cup}$. As *Eccore* is not applicable to methods and local variables, an alternative strategy is required to estimate the recall, which, herein is done with a liberal equivalent defined as follows. For a given program element e, we perform a textual search for all instances of the element's identifier in comments in the *same file* as e, and identify any fragile phrase in the set. Then, the union of the set of these fragile phrases and $\text{matches}(e)$ as the equivalent of $\text{refersTo}(e)$ is used as the measure of recall. This version of recall is referred to as $\textbf{Recall}^{*}$. In general, both approximations of recall can be expected to be an *upper bound* approximation of the true recall of the approach.

6.6 Threats and Limitations

The threats to validity and limitations of the experimental design are as follows. First, the population from which the sample of program identifiers is drawn, is defined as a function of the approach, as described in §6.2. However, this will only introduce bias as a function of the number of identifiers for which the approach generates *a)* only false positives, or *b)* no positives in the presence of false negatives. Case a) can be precisely controlled and it was verified that across all of the target projects, the sampling error is between 0 and 3 elements for all projects. Case b) is impossible to determine reliably, but can be estimated to be very low given the high recall reported in the next Chapter. The threat of investigator bias when deciding whether a match is a true positive or not is mitigated by the cross-annotation performed by the second annotator. It is further reduced by the fact that the task is of low subjectivity, and that we have released our benchmarks publicly[1]. Second, the level of the knowledge of both the annotators about programming language as well as English language can pose a threat to the validity of results. Finally, as mentioned above, the computation of recall designed in this thesis is approximation of a theoretical value that is not feasible to compute precisely. In consequence of these experimental conditions, the proper way to interpret the results presented in the next chapter is as an illustration of the potential of the approach in twelve distinct contexts, as opposed to a general prediction of the operational performance of the tool.

[1] http://www.cs.mcgill.ca/~swevo/inderjotmsc

CHAPTER 7

RESULTS AND DISCUSSIONS

In this chapter, we present the results of evaluation study performed on the approach using two different datasets - *development set* and *test set A*. The *development set* and *test set A* benchmarks are annotated by the author of this thesis and the results obtained comparing the performance of Fraco and Eccore against these benchmarks are presented in § 7.1 and § 7.2.

To assess the reliability of the benchmark, we asked a second, external evaluator to identify true positives, false positives, and false negatives for a subset of the *Test set A's* benchmark. We then measured agreements and disagreements between the two annotators. We present this analysis in Section §7.3. We refer to the subset of identifiers annotated by the external annotator as Test Set B. Note that, the test set B is essentially the same sample as test set A except that it contains only one fourth of the elements present in test set A and is annotated by multiple annotators.

7.1 Development Set

We report the results of our evaluation in three parts organized to facilitate the interpretation of the data collected. First, we present the results of the evaluation of the lexical matching rules for the methods and local variables. These results must be interpreted in absolute terms because Eccore does not support replacement for the identifiers of such types. We then report the results of the lexical matching rules for types and fields, which we compare against Eccore's output. The results

of the first two sections are based on the same sample. Finally, we report on the results of the semantic matching rule which is based on all of the identifiers in the target systems. Note that, the evaluation results for development set were obtained before including the language-specific features related to overridden methods.

Lexical Matching of Methods and Local Variables: Table 7.1 shows the results of the evaluation of the lexical matching rules for the methods and local variables. We note that JUnit and Spring Data Redis do not have mentions of local variable identifiers in the comments for the given sample, so performance results in these cases are not available. For methods, the precision is above 95% for all the systems except Spring Data Redis. However, all of the false positives in Spring Data Redis are caused by the artificial case that the comment refers to a method that is being overridden, which is not a likely scenario in practice given that renaming an overriding method changes the behavior of the code.

The recall* is generally very good, with only Chronicle Map registering eight false negatives. Seven of these cases are caused by overloading or other types of ambiguity related to arguments. For example, a method named `of(first,second)` is not matched to an in-comment reference `of(...)`. The eighth case was one of the confusion between a field and method name. Local variables are seldom referred to in comments. In our sample, we observe perfect recall* but equivocal precision for both JFlex and Chronicle Map. For these systems, the precision is lower due to the use of common English words like *move* as a local variable identifier, which generates natural ambiguity in the refersTo relation.

Lexical Matching of Types and Fields: Tables 7.2 and 7.3 show the results of the evaluation of lexical matching rules for types and fields. The results can be interpreted in the same way as those in Table 7.1 except that in this case we use recall$^{\cup}$ as defined in §3.3. The main observation for types is that it is a relatively simple problem to solve. Eccore shows perfect

Table 7.1 – Results of the evaluation for identifiers of local variables and methods in **Development set**. The columns indicate the number of identifiers searched (Ids), the number of true positives (TP), the number of false positives (FN) the number of file-relative false negatives (FN), the precision (P), and file-relative recall (R*, described in §6.5)

Project Name	**Method Type**						**Local Variable Type**					
	Ids	*TP*	*FP*	*FN*	*P*	*R**	*Ids*	*TP*	*FP*	*FN*	*P*	*R**
Log4j	40	110	2	3	98	97	6	6	0	0	100	100
JUnit	40	102	0	1	100	99	-	-	-	-	NA	NA
Joda time	54	214	0	1	100	99	2	2	0	0	100	100
JFlex	40	60	0	0	100	100	12	10	2	0	83	100
Chronicle Map	41	139	7	8	95	94	3	2	1	0	67	100
Spring Data Redis	33	57	16	0	78	100	0	0	0	0	NA	NA

precision across all projects and Fraco only generates one false positive. On the other hand, the recall of Fraco is superior for five out of the six projects because of the additional context-sensitive tolerance for plurals and case-insensitive matching. Fraco performed on par with Eccore in the case of case-sensitive matches whereas the case-insensitive and plurals matching rule increased the total number of true fragile comments detection by Fraco, which resulted in higher recall. For Spring Data Redis, the recall$^{\cup}$ is lower due to some unexpected uses of fully-qualified names.

In the case of fields, Fraco clearly dominates with perfect precision and very high recall for all but one system. In contrast, Eccore flounders in many situations. For example, if both a field and a method's parameter have the same identifier, Eccore replaces the references of the formal parameter as well. In a more egregious case, when we rename a field named `it` in Chronicle Map, Eccore generates 244 false positives in various comments including the copyright block of files.

Table 7.2 – Results of the evaluation for identifiers of category- Types in **Development set**. The columns indicate the number of identifiers evaluated (Ids), the number of true positives (TP), the number of false positives (FN) the number of false negatives (FN), the precision (P), and recall ($R^{\cup}$, described in §6.5)

Project Name	**Types**										
	IDs	**Eccore**					**Fraco**				
		TP	*FP*	*FN*	*P*	$R^{\cup}$	*TP*	*FP*	*FN*	*P*	$R^{\cup}$
Log4j	33	161	0	29	100	84	174	0	17	100	91
JUnit	54	372	0	65	100	85	407	1	46	99	90
Joda time	32	272	0	49	100	84	323	0	7	100	97
JFlex	28	138	0	103	100	57	223	0	29	100	88
Chronicle Map	42	297	0	51	100	85	350	0	27	100	93
Spring Data Redis	54	225	0	17	100	93	223	0	52	100	81

Semantic Matching: Tables 7.4 and 7.5 show the precision of the semantic matching rule for all identifiers that produced at least one match. We do not attempt to compute the recall for semantic matching because it is not possible to reliably determine the extension of the set of true positives. The main observation that we can draw from these results is that although coverage is relatively low, precision is again very high. The few false positives for methods were generated by the first special case for the return values of method identifiers (see §5.3). However, the handful of false positives generated by this rule are largely offset by the true positives it properly captures. Finally, when we project the evaluation of the semantic matching rules back onto the sample used for lexical rules, we conclude that the number of fragile phrases detected increases by ratios between 4.4% (JFlex) and 51.8% (Chronicle Map).

Table 7.3 – Results of the evaluation for identifiers of category- Fields in **Development Set**. The columns indicate the number of identifiers evaluated (Ids), the number of true positives (TP), the number of false positives (FN) the number of false negatives (FN), the precision (P), and recall ($R^{\cup}$, described in §6.5).

Project Name	**Fields**										
	IDs	**Eccore**					**Fraco**				
		TP	*FP*	*FN*	*P*	$R^{\cup}$	*TP*	*FP*	*FN*	*P*	$R^{\cup}$
Log4j	21	10	15	11	40	47	87	0	3	100	96
JUnit	6	4	0	3	100	57	7	0	0	100	100
Joda time	12	4	0	23	100	15	28	0	0	100	100
JFlex	20	28	107	0	20	100	26	0	2	100	93
Chronicle Map	14	9	3	12	75	42	21	0	0	100	100
Spring Data Redis	13	11	49	6	18	65	17	4	0	80	100

7.2 Test Set A

Following the same pattern used to present the results in §7.1, we organized the evaluation results in three parts. The first part presents the evaluation results of lexical matching rules for the methods and local variables. We, then, report the results of comparison against Eccore's output for types and fields obtained on applying the lexical matching rules. The results presented in these two parts are based on the same sample i.e., 200 elements taken from each test set system. Finally, we report the results of semantic matching rules for all of the identifiers present in the six test set systems.

Lexical Matching of Methods and Local Variables: Table 7.6 shows the evaluation results for methods and local variables obtained on apply lexical matching rules. The systems Hazelcast, Commons-IO and JFreechart have relatively low number of local variables than the

Table 7.4 – Results of the semantic matching for Types and Methods in the *Development Set*.

Project Name	**Class Type**				**Method Type**			
	Ids	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*
Log4j	12	15	0	100	80	94	4	96
JUnit	26	39	0	100	73	85	1	98
Joda time	10	11	0	100	90	157	0	100
JFlex	9	9	0	100	41	54	1	98
Chronicle Map	17	27	1	96	9	11	0	100
Spring Data Redis	37	40	0	100	61	71	1	98

other three systems because these systems have an overall lower percentage of local variables, having at least one fragile comment, compared to other element types. Since, the underlying logic of stratified random sampling is based on the ratios of different stratas in the overall population, in our case stratas are the types of identifiers, it is counter-intuitive to add more number of local variables into the sample. For the small number of local variables selected in the sample, our evaluation results show 100% precision and recall. The results for elements of type method almost achieved a perfect precision score for most of the test set systems except for Commons-IO and Mockito.

Commons-IO used a specific naming pattern for singleton classes i.e., the method returning the class instance had same identifier as the class except the first letter was in lower case. Our lexical matching rules allow for case-insensitive matching in the *local comments*, which in this case are the comments inside the class declaration, to capture the various fragile phrases like the ones described in §5.2. We noted that all cases of false positives are caused by this naming pattern. However, the number of true positives captured by allowing for case-insensitive matches in the

Table 7.5 – Results of the semantic matching for Fields and Local Variables in the **Development set**.

Project Name	**Field Type**				**Local Variable Type**			
	Ids	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*
Log4j	5	6	0	100	2	2	0	100
JUnit	-	-	-	NA	-	-	-	NA
Joda time	-	-	-	NA	-	-	-	NA
JFlex	2	2	0	100	2	2	0	100
Chronicle Map	1	1	0	100	2	2	0	100
Spring Data Redis	-	-	-	NA	0-	-	-	NA

local comments sidetracks the few false positives generated.

In case of Mockito, all the cases of false positives registered are due to the use of one word identifiers for methods that are common in English language such as `only`, `then`, `calls` etc. Our lexical rule, shown in $Cell_{3,1}$ of Table 5.1, searches for the identifier of the declaring class when detecting fragile phrases referring to the method. Nonetheless, this heuristic failed to prevent these false positives generated in system Mockito. Although, the fragile phrases detected correctly identified the relation between the given method and the parent class e.g., "only (method's name) in Mockito class" except here "only" was not a method's name, but the unusual naming pattern used in Mockito resulted in a bunch of false positives.

Most of the systems had a near-perfect recall* except Mockito and JFreechart. JFreechart had a total of 4 cases of false negatives caused by the typographical errors where a "space" character was missing between the identifier under processing and an adjacent token. In case of Mockito, a total of 76 false negatives are produced due to the presence of source-code elements in comments

as shown below:

```
when(mock.someMethod("arg1", "arg2")).thenReturn("one", "two");
```

Our lexical rule in $Cell_{3,1}$ of Table 5.1 applies a condition to match the number of parameters of a method when detecting fragile phrases. Consider the method under processing is `when` declared with only one formal parameter and after tokenization, the above mentioned source-code line present in a comment is incorrectly marked as not fragile by our lexical rule. This happens due to incorrect parsing of code snippets by Fraco as parsing the code snippets present in comments is a very difficult problem and requires special techniques like island parsing. In this example, based on the number of commas appearing inside round parentheses, Fraco counts two parameters instead of one and hence, marks it as false positive.

Also, the *header* comments of class type identifiers in Mockito are extremely long and span over 1300 to 1500 lines. These 76 cases of false negatives were found in only two comments for method `when`. However, keeping track of the commas with respect to the pair of open and close parentheses can help avoid these kind of false negatives and can be fixed in the future versions of the approach.

Lexical Matching of Types and Fields: Tables 7.7 and 7.8 show the evaluation results for types and fields. In case of types and fields, the recall$^{\cup}$ calculated, as described in §3.3, is based on the baseline results obtained from Eccore.

Types: For types, Eclipse achieved 100% precision whereas Fraco had a few cases of false positives. All of these false positives are generated due to common single word identifiers. Fraco allows for case-sensitive matching in *global* comments that generated these few cases of false positives. The figure 7.1 shows an example of a comment that refers to the field declaration shown but Fraco incorrectly matches this comment to the class's declaration named `Day`.

The recall of Eccore is extremely low as compared to Fraco. Since Eccore strictly matches

Table 7.6 – Results of the evaluation for identifiers of local variables and methods in the **Test Set A**. The columns indicate the number of identifiers searched (Ids), the number of true positives (TP), the number of false positives (FN) the number of file-relative false negatives (FN), the precision (P), and file-relative recall (R*, described in §6.5)

Project Name	**Method Type**						**Local Variable Type**					
	Ids	*TP*	*FP*	*FN*	*P*	*R**	*Ids*	*TP*	*FP*	*FN*	*P*	*R**
Commons-IO	90	247	12	0	95	100	1	1	0	0	100	100
Mockito	95	584	76	76	88	88	6	13	0	0	100	100
JFreeChart	140	270	0	4	100	99	3	3	0	0	100	100
Findbugs	79	136	0	0	100	100	13	13	0	0	100	100
JMeter	63	108	0	0	100	100	7	10	0	0	100	100
Hazelcast	57	110	0	0	100	100	3	3	0	0	100	100

```
/** Day. */
public static final DateTickUnitType DAY
 = new DateTickUnitType("DateTickUnitType.DAY", Calendar.DATE);
```

Figure 7.1 – Example of a false positive case resulted due to same identifiers used for both field and Type.

the case of an identifier while looking for the matching references in comments, it misses the true references of the identifier that are detected by Fraco using case-insensitive lexical matching rule. Fraco achieved a higher recall but also registered some cases of false negatives. These cases are generated due to two reasons: a) First, the mention of source-code elements in copyright comments. b) The occurrence of case-insensitive matches outside of local comments. JFreechart has unexpected mention of the source-code elements inside copyright comments. Fraco does not consider copyright comments as a target while detecting fragile comments. The purpose of copyright

comments is not to contain any references to the source code elements and in eleven out of twelve systems, no references to source-code elements were found in copyright comments which justifies our decision to ignore such comments while detecting fragile phrases. Also, in case of one word identifiers, a number of cases are detected where identifier is mentioned in lower case in the comments that are *global* with respect to the identifier. Fraco identifies such case-insensitive cases in *local* comments only and therefore, failed to capture the reference in *global* comments. For example, the fragile phrase located in line 1 of the example shown in Figure 7.2 is not registered by Fraco because it is located outside the class `Packet`.

```
/**
 * Transmits a packet to a certain connection.
 * If this method is called with a null connection, the call returns false
 * @param packet     The Packet to transmit.
 * @param connection The connection to where the Packet should be transmitted.
 * @return true if the transmit was a success, false if a failure.
 */
boolean transmit(Packet packet, Connection connection);
```

Figure 7.2 – Example showing a fragile comment related to class “Packet” of project “Hazelcast” present in class file “ConnectionManager.java”

Fields: In case of fields, Fraco performed extremely well as compared to Eccore. Both precision and recall of Eccore are terribly low and all the cases of false positives are caused by incorrectly matching field identifiers with identifiers referring to formal parameters of a method. Also, Eccore left a number of references undetected resulting in lower recall. These false negatives resulted from the strict use of case-sensitive matching.

Fraco also registered a few cases of false positives owing to the lexical matching rule that considers a phrase fragile with respect to a field if the whole identifier is in upper case irrespective of the location of comments i.e., it does not differentiate between *local* and *global* comments for

Table 7.7 – Results of the evaluation for identifiers of category- Types in **Test Set A**. The columns indicate the number of identifiers searched (Ids), the number of true positives (TP), the number of false positives (FN) the number of false negatives (FN), the precision (P), and recall ($R^{\cup}$, described in §6.5)

Project Name	**Types**										
	IDs	**Eccore**					**Fraco**				
		TP	*FP*	*FN*	*P*	$R^{\cup}$	*TP*	*FP*	*FN*	*P*	$R^{\cup}$
Commons-IO	66	416	0	135	100	76	533	0	0	100	100
Mockito	86	650	0	419	100	60	974	11	0	99	100
JFreechart	29	325	0	262	100	55	451	2	62	99	88
Findbugs	75	390	0	459	100	46	940	11	28	99	97
JMeter	80	347	0	304	100	53	648	0	14	100	98
Hazelcast	108	303	0	219	100	58	634	0	70	100	90

such field identifiers. In most of the cases, this rule captures a lot of true positive references of an identifier mentioned in *global* comments which compensates for a few false positives detected. The recall results for Fraco surpass the Eccore results by a huge margin. However, there are a few cases of false negatives detected in Commons-IO and JFreechart. Again, JFreechart has source-code elements mentioned in copyright comments which is solely the reason for its lower recall value. Commons-IO registered seven cases of false negatives due to the use of same identifier for a field and the formal parameter of a method creating ambiguity. For example, consider a field named `filter` declared in class `FileFilter`. Figure 7.3 shows a method from the same class having a formal parameter, which in our case belongs to the *local variables* category, with same identifier as the field in consideration. In this example, the line 8 of the comment can be related to the field with certainty based on our lexical matching rule that also looks for the mention of its declaring class while searching in *global* comments. But, there are certain ambiguous references

to the field such as in line 1 and 3 that cannot be subjectively related to either the field or formal parameters.

```
/**
 * Construct an instance with a filter and limit the <i>depth</i> navigated to.
 * <p>
 * The filter controls which files and directories will be navigated to as
 * part of the walk. The {@link FileFilterUtils} class is useful for combining
 * various filters together. A {@code null} filter means that no
 * filtering should occur and all files and directories will be visited.
 * Note that this functionality is in addition to the filtering by file filter.
 * @param filter  the filter to apply, null means visit all files
 * @param depthLimit  controls how <i>deep</i> the hierarchy is
 *  navigated to (less than 0 means unlimited)
*/
  protected DirectoryWalker(final FileFilter filter, final int depthLimit) {
     this.filter = filter;
     this.depthLimit = depthLimit;
}
```

Figure 7.3 – Example from the source code of system “Commons-IO” showing a method and its header comment from class “FileFilter.java”

Semantic Matching: Similar to the representation of results for the development set, Tables 7.9 and 7.10 show the evaluation results obtained on applying semantic matching rules. Only the identifiers having at least one fragile phrase detected by semantic matching rules are included in the results. As described in the semantic results section of the development set, only precision results are reported in case of semantic matches.

The performance for semantic matching is generally good except. except for a few cases of false positives. Instead of discussing these false positives according to the identifier categories, we discuss these cases for all identifiers together because the factors causing these false positives are

Table 7.8 – Results of the evaluation for identifiers of category- Fields in in **Test Set A**. The columns indicate the number of identifiers searched (Ids), the number of true positives (TP), the number of false positives (FN) the number of false negatives (FN), the precision (P), and recall ($R^{\cup}$, described in §6.5).

Project Name	**Fields**										
	IDs	**Eccore**					**Fraco**				
		TP	*FP*	*FN*	*P*	$R^{\cup}$	*TP*	*FP*	*FN*	*P*	$R^{\cup}$
Commons-IO	52	107	174	10	38	37	127	1	7	99	95
Mockito	13	38	0	17	100	69	59	0	0	100	100
JFreechart	28	120	570	0	17	100	42	6	16	88	72
Findbugs	33	40	3	0	93	100	59	0	0	100	100
JMeter	50	53	27	4	66	93	67	6	0	92	100
Hazelcast	32	72	8	60	90	55	148	10	0	94	100

common for all types of identifiers. In case of Mockito and Commons-IO, the precision dropped due to the use of absurd method identifiers like `byteThat` and `afterWrite`, which resulted in false positives due to the use of unrelated and commonly occurring English words.

7.3 Inter-rater Agreement

The sample selected as Set B contains 50 identifiers per test set systems that are extracted as a subset from Set A through stratified sub-sampling of Set A identifiers and therefore, the evaluation results in terms of precision and recall have already been discussed profoundly. In this section, we compare the benchmark annotated by the two different annotators and observe the amount of agreement or disagreement between them.

Table 7.9 – Results of the semantic matching for Types and Methods in the **Test Set A**.

Project Name	**Class Type**				**Method Type**			
	Ids	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*
Commons-IO	57	132	5	96	127	199	5	98
Mockito	67	151	0	100	110	150	13	92
JFreechart	26	81	0	100	143	179	0	100
Findbugs	59	172	0	100	145	182	0	100
JMeter	44	70	1	96	150	207	1	99
Hazelcast	45	60	0	100	66	86	0	100

Agreement metric: Initially, we used *Cohen's Kappa* statistic to measure the inter-annotator agreement [56]. Kappa measures the inter-rater agreement for categorical items. Because, we also have two categories in our result set i.e., 1 and 0 for fragile and not fragile respectively, it seemed an appropriate choice of a qualitative measure for assessing the agreement. Kappa is usually preferred over the simple percentage agreement calculation due to the additional factor of chance probability which makes it robust than the state-of-art statistics.

While measuring the inter-rater agreement using Kappa, we observed that Kappa is not the right statistic to measure agreement in our case because the false positives were very rare in our result set which drove the value of chance probability close to zero. We observed 100% general agreement probability in most of the cases. When the chance probability becomes zero, the Kappa will always be zero. So in our case, the value of Kappa was always zero or less than 0.3. Therefore, we decided to use the simple percentage measure to calculate agreement between the annotators.

Table 7.10 – Results of the semantic matching for Fields and Local Variables in the **Test Set A**.

Project Name	**Field Type**				**Local Variable Type**			
	Ids	*TP*	*FP*	*P*	*Ids*	*TP*	*FP*	*P*
Commons-IO	17	17	0	100	-	-	-	NA
Mockito	0	0	0	NA	-	-	-	NA
JFreechart	42	0	0	100	-	-	-	NA
Findbugs	3	3	0	100	-	-	-	NA
JMeter	9	9	0	100	9	0	0	NA
Hazelcast	2	2	0	100	-	-	-	NA

Percentage Agreement Calculation: For every fragile phrase detected by the tool, the annotators made a binary decision marking it as fragile or non-fragile. In addition to that, each annotator marked the number of fragile instances missing in each class file, i.e the Recall (R*) measure as described in §6.5. In order to calculate percentage agreement between the annotators, for each system in the test set B, we count the number of phrases marked fragile (TP), non-fragile (FP) and the number of instances marked missing per identifier (FN) by both the annotators.

For each category of annotation, i.e. true positives (TP), false positives (FP) and false negatives (FN), we calculate the agreement between annotators by dividing the number of cases agreed upon by both the annotators by the maximum number of cases in that specific category marked by either of the annotators. Consider an example from the annotations results shown in Table 7.11 where in case of Commons-IO, the external annotator marked 222 cases as true positives and the lead annotator marked 223 cases as true positives. Therefore, to calculate agreement between the two annotators for the true positives cases, the percentage agreement is 222/223 $\approx$ 100%. Similarly for false positives and false negatives, the agreement calculated is 0% and 2/12 $\approx$ 17%.

Comparison of results: The results of evaluation conducted by two annotators are presented in Tables 7.11 and 7.12. Each table is divided into two parts - the first part shows results of annotation by the external annotator and second part shows the annotation results generated by the lead annotator i.e., the author of this thesis. These tables present the data annotations before resolving the conflicts between annotators. The data shows an agreement of 96% in case of lexical matches and 100% in case of semantic matches before resolving conflicts.

As it can be noted in the Table 7.12, the annotation results for the semantic matching rules have exactly the same number of false positives and negatives attributing to 100% agreement between the annotators. In case of lexical rules, the annotators agree over almost all the systems except Commons-IO, Mockito and Findbugs.

Both the annotators, i.e. the author of this thesis and external, annotated the data separately. In order to assess and improve the quality of the annotations done by the external annotator, both annotators completed annotations for one system at a time and compared the annotated datasets to resolve any conflicts before annotating the next system's dataset. This exercise helped us to avoid the trivial conflicts that arose due to the miscalculation of file-based missing instances or due to the partially incorrect understanding of instructions given to the external for annotation task i.e. a few mis-characterizations of the identifier types resulting in incorrect annotations. For example, both annotators started by annotating the dataset of Commons-IO system and on comparing the annotated datasets, we found that the external annotator reported 9 missing instances of a method named `directoryFileFilter`. However, 5 out of 9 missing instances were a result of miscalculation whereas 3 belonged to the class `DirectoryFileFilter` and not the method. We addressed two more instances of conflicts where the external annotator incorrectly marked the phrase related to a field type element as missing whereas the phrase was related to the formal parameter and not the field. Resolving the conflicts registered in the dataset of Commons-IO helped the external annotator to correctly understand the information provided in various columns of the dataset resulting in

better annotation of the data of the remaining 5 test set systems.

The conflicts registered in other two systems, i.e. Findbugs and Mockito, were due to the ambiguity between two same identifiers of different categories. An example of such conflicting comments is shown below:

```
/* ....
 * The gotcha is that Mockito does the validation <b>next time</b>
 * you use the framework (e.g. next time you verify, stub, call mock etc.).
 * ...
 */
```

Consider the method `framework` of class `Mockito` is renamed and the comment shown above is under processing to detect the fragile phrases. In this case, the external annotator marked this comment as true positive with respect to the method `framework` whereas the lead annotator marked it as false positive because, according to the lead annotator, it refers to the Mockito framework in general and not the method. There are 13 such instances related to this same method in the dataset accounting for the major percentage of disagreement between the two annotators.

On resolving all the conflicts between annotators, we achieved an agreement of 100% in case of both lexical matches and semantic matches. And finally, the results of comparison between the Set B and the dataset created from the output of Eccore show a 100% inter-annotator agreement and therefore, the breakdown of Eccore results is not presented in the table. High inter-annotator agreement proves that the task of marking fragile phrases is of low-subjectivity and, therefore, mitigates the threats to validity related to manual annotations done by a single annotator.

Table 7.11 – Evaluation results of **lexical** matches for all 50 identifiers in **Test Set B** evaluated by both annotators. The columns indicate the number of true positives (TP), the number of false positives (FP), the number of file-relative false negatives (FN), the percentage agreement in case of fragile phrases (FP), the percentage agreement is case of non-fragile phrases (NFP) and the percentage agreement in case of missing fragile instances (MFP) per system.

Project Name	**External Annotator**			**Lead Annotator**			**Agreement %**		
	TP	*FP*	*FN*	*TP*	*FP*	*FN*	*FP*	*NFP*	*MFP*
Commons-IO	222	1	12	223	0	2	99	0	17
Mockito	763	23	82	747	39	82	98	59	100
JFreechart	240	0	6	240	0	6	100	100	100
Findbugs	188	3	6	191	0	6	98	0	100
JMeter	170	3	9	170	3	9	100	100	100
Hazelcast	160	1	7	160	1	7	100	100	100
Overall Percentage Agreement							99%	60%	86%

Table 7.12 – Evaluation results of **semantic** matches for all 50 identifiers in **Test Set B** evaluated by both annotators. The columns indicate the number of true positives (TP), the number of false positives (FP), the number of file-relative false negatives (FN), the percentage agreement in case of fragile phrases (FP), the percentage agreement is case of non-fragile phrases (NFP) and the percentage agreement in case of missing fragile instances (MFP) per system.

Project Name	**External Annotator**			**Lead Annotator**			**Agreement %**		
	TP	*FP*	*FN*	*TP*	*FP*	*FN*	*FP*	*NFP*	*MFP*
Commons-IO	79	1	0	79	1	0	100	100	100
Mockito	85	0	4	85	0	4	100	100	100
JFreechart	78	0	0	78	0	0	100	100	100
Findbugs	61	0	2	61	0	2	100	100	100
JMeter	70	0	1	70	0	1	100	100	100
Hazelcast	66	0	0	66	0	0	100	100	100
Overall Percentage Agreement							100%	100%	100%

CHAPTER 8
CONCLUSIONS

This thesis formalizes the problem of detecting fragile comments with respect to the rename refactorings and proposes a novel rule-based approach for detecting fragile phrases in the source code comments by taking into account the type of identifier being renamed, its morphology, the scope of the identifier and the location of the comments. By limiting the input of the analysis to general programming language features and common naming conventions, our approach can remain general-purpose and detect fragile phrases in the comments of any type.

Although our approach relies on language-independent principles, we developed a prototype for the Java programming language as an Eclipse plug-in called *Fraco*. We evaluated *Fraco* on a sample of 1800 identifiers taken from twelve medium to large sized systems and, when possible, compared the results against *Eccore*, Eclipse's identifier reference replacement feature. While detecting fragile comments for type declarations, both Fraco and Eccore performed at par. However, when renaming fields, the performance of Eccore showed dramatic unreliability, with precision varying between 20% and 100% between systems, and recall varying between 15% and 100%. In contrast, the more sophisticated rule set of Fraco showed a precision of 100% for nine of the twelve systems, and a recall above 90% for all test systems except Mockito. In fact, Mockito was a really good test system for our approach because of the unusual names used for the program elements and the excellent results achieved by Fraco with such names proves that our approach is robust to the variations in the identifier patterns and can achieve a satisfactory results. While Eccore currently does not even support the detection of fragile comments when renaming methods

and local variables, Fraco was able to detect fragile phrases in these cases with precision and recall of 95% or above for a majority of systems.

8.1 Future Work

The three clear avenues for future work in this area are *comment repair*, a broadening of the definition of *semantic matching* and extending the approach to cover all types of changes done manually or automatically. Currently, our approach detects fragile phrases and repairs the fragile phrases detected by lexical matching rules only because the fragile phrases detected by semantic matches require a context-aware algorithm that not only analyzes a declaration's source code context but also the takes into account the scope and multiple different references to a declaration throughout a system's source code. Although it is unlikely that the problem of repair is fully automatable for semantic matches, it will be interesting to explore how to differentiate matches that can be reliably repaired automatically from those that require developer assistance.

As for semantic matching, our current approach purposefully remains very close to the lexical layer because it targets the invalidation of textual references to the specific identifiers. Currently, our approach does not include synonym detection or expansion of abbreviations to match the fragile comments that could be using expanded versions of the identifier which is renamed. In the greater context of software evolution research, the problem of detecting general inconsistencies between source code and unstructured text remains a major challenge, which we could modestly approach with an expansion of the semantic rules to include additional components such as synonyms analysis, expansion of abbreviations and entity recognition.

Currently, our approach is limited to detecting fragile comments only for renamed identifiers. We plan to extend the detection of fragile comments for all types of changes done manually or automatically using tools like *ChangeDistiller* [13] so that our tool covers the full spectrum of changes made in the source code of a system and help the developer to automate the process of detecting and repairing fragile comments.

BIBLIOGRAPHY

[1] "Eclipse's Refactoring API's documentation", http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm, Verified on 5 October, 2017.

[2] "JetBrains", https://www.jetbrains.com/dotnet, Verified on 5 October, 2017.

[3] "Automatic Mapping Among LexicoGrammatical Annotation Models (AMALGAM).", http://www.comp.leeds.ac.uk/amalgam/amalgam/amalghome.htm, Verified 5 October, 2017.

[4] Venera Arnaoudova, Laleh M Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gael Gueheneuc, "Repent: Analyzing the nature of identifier renamings", *IEEE Transactions on Software Engineering*, 40:502–532, 2014.

[5] Dave Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell, "To camelcase or under_score", In *Proceedings of the 17th International Conference on Program Comprehension, 2009.*, pages 158–167, 2009.

[6] "Checkstyle Home Page", http://checkstyle.sourceforge.net/, Verified 5 October, 2017.

[7] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa, "Natural language processing (almost) from scratch", *Journal of Machine Learning Research*, 12:2493–2537, 2011.

[8] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation", *Software Quality Journal*, pages 1–27, 2016.

[9] Barthélémy Dagenais and Martin P Robillard, "Using traceability links to recommend adaptive changes for documentation evolution", *IEEE Transactions on Software Engineering*, 40:1126–1146, 2014.

[10] Danny Dig and Ralph Johnson, "How do APIs evolve? A story of refactoring", *Journal of software maintenance and evolution: Research and Practice*, 18:83–107, 2006.

[11] "Eclipse Home Page", https://eclipse.org/, Verified 5 October, 2017.

[12] "Eclipse documentation - Current Release Eclipse Neon", http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm, Verified on 5 October, 2017.

[13] Beat Fluri and Harald C Gall, "Classifying change types for qualifying change couplings", In *Proceedings of the 14th International Conference on Program Comprehension*, pages 35–45, 2006.

[14] Beat Fluri, Michael Wursch, and Harald C Gall, "Do code and comments co-evolve? on the relation between source code and comment changes", In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, 2007.

[15] Martin Fowler and Kent Beck, *Refactoring: Improving the design of existing code*, Addison-Wesley Professional, 1999.

[16] Peter Fritzson, Adrian Pop, Kristoffer Norling, and Mikael Blom, "Comment and Indentation Preserving Refactoring and Unparsing for Modelica", In *Proceedings of the 6th International Modelica Conference*, pages 3–4, 2008.

[17] Xi Ge, Quinton L DuBose, and Emerson Murphy-Hill, "Reconciling manual and automatic refactoring", In *Proceedings of the 34th International Conference on Software Engineering*, pages 211–221, 2012.

[18] James Gosling, Bill Joy, Guy Steele, Bilad Bracha, and Alx Buckley, *The Java® Language Specification, Java SE 8 Edition*, Feb 2015, https://docs.oracle.com/javase/specs/jls/se8/html/index.html. Verified 5 October, 2017.

[19] Jhe-Jyun Guo, Nien-Lin Hsueh, Wen-Tin Lee, and Shi-Chuen Hwang, "Improving Software Maintenance for Pattern-Based Software Development: A Comment Refactoring Approach", In *Proceedings of the International Conference on Trustworthy Systems and their Applications*, pages 75–79, 2014.

[20] Emily Hill, "Integrating natural language and program structure information to improve software search and exploration", PhD thesis, University of Delaware, 2010.

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

[22] "Writing doc comments for the Javadoc tool", http://www.oracle.com/technetwork/articles/java/index-137868.html, Verified 5 October, 2017.

[23] Zhen Ming Jiang and Ahmed E Hassan, "Examining the evolution of code comments in PostgreSQL", In *Proceedings of the 2006 International workshop on Mining Software Repositories*, pages 179–180, 2006.

[24] David Kawrykow and Martin P Robillard, "Non-essential changes in version histories", In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360, 2011.

[25] Ninus Khamis, René Witte, and Juergen Rilling, "Automatic Quality Assessment of Source Code Comments: The JavadocMiner.", In *Proceedings of the 15th International Conference on Applications of Natural Language to Information Systems*, pages 68–79, 2010.

[26] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan, "A field study of refactoring challenges and benefits", In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, page 50, 2012.

[27] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley, "Effective identifier names for comprehension and memory", *Innovations in Systems and Software Engineering*, 3:303–318, 2007.

[28] Ben Liblit, Andrew Begel, and Eve Sweetser, "Cognitive perspectives on the role of naming in computer programs", In *Proceedings of the 18th annual psychology of programming workshop*, 2006.

[29] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al., *Introduction to information retrieval*, volume 1, Cambridge university press Cambridge, 2008.

[30] Christopher D Manning and Hinrich Schiitze, "Foundations of Statistical Natural Language Processing".

[31] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky, "The Stanford CoreNLP natural language processing toolkit.", In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[32] George A Miller, "WordNet: a lexical database for English", *Communications of the ACM*, 38:39–41, 1995.

[33] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker, "Automatic generation of natural language summaries for Java classes", In

Proceedings of the 21st International Conference on Program Comprehension, pages 23–32, 2013.

[34] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker, "JSummarizer: An automatic generator of natural language summaries for Java classes", In *Proceedings of the 21st International Conference on Program Comprehension*, pages 230–232, 2013.

[35] Gail C Murphy, Mik Kersten, and Leah Findlater, "How are Java software developers using the Eclipse IDE?", *IEEE software*, 23:76–83, 2006.

[36] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black, "How we refactor, and how we know it", *IEEE Transactions on Software Engineering*, 38:5–18, 2012.

[37] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig, "A comparative study of manual and automated refactorings", In *Proceedings of the European Conference on Object-Oriented Programming, Springer*, pages 552–576, 2013.

[38] Wyatt Olney, Emily Hill, Chris Thurber, and Bezalem Lemma, "Part of speech tagging Java method names", In *Proceedings of the International Conference on Software Maintenance and Evolution, 2016*, pages 483–487, 2016.

[39] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou, "Listening to programmers Taxonomies and characteristics of comments in operating system code", In *Proceedings of the 31st International Conference on Software Engineering*, pages 331–341, 2009.

[40] "The Penn Treebank Project", http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html, Verified 5 October, 2017.

[41] Joël Plisson, Nada Lavrac, Dr Mladenić, et al., "A rule based approach to word lemmatization", 2004.

[42] Inderjot Kaur Ratol and Martin P. Robillard, "Detecting Fragile Comments", In *Proceedings of the 32nd International Conference on Automated Software Engineering*, pages 11, 2017, To appear.

[43] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber, "Retaining comments when refactoring code", In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 653–662, 2008.

[44] Giriprasad Sridhara, "Automatically Detecting the Up-To-Date Status of ToDo Comments in Java Programs", In *Proceedings of the 9th ACM India Software Engineering Conference*, pages 16–25, 2016.

[45] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker, "Towards automatically generating summary comments for java methods", In *Proceedings of the International Conference on Automated Software Engineering*, pages 43–52, 2010.

[46] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools", In *Proceedings of the 16th International Conference on Program Comprehension, 2008.*, pages 123–132, 2008.

[47] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker, "Generating parameter comments and integrating with method summaries", In *Proceedings of the 19th International Conference on Program Comprehension, 2011*, pages 71–80, 2011.

[48] Daniela Steidl, Benjamin Hummel, and Elmar Juergens, "Quality analysis of source code comments", In *Proceedings of the 21st International Conference on Program Comprehension*, pages 83–92, 2013.

[49] Lin Tan, "Leveraging code comments to improve software reliability", PhD thesis, University of Illinois at Urbana-Champaign, 2009.

[50] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou, "/* iComment: Bugs or bad comments?*", In *Proceedings of the 21st Symposium on Operating Systems Principles*, pages 145–158, 2007.

[51] Lin Tan, Ding Yuan, and Yuanyuan Zhou, "Hotcomments: how to make program comments more useful?", In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 7–9, 2005.

[52] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies", In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[53] Yuan Tian, David Lo, and Julia Lawall, "Automated construction of a software-specific word similarity database", In *Proceedings of Software Evolution Week Conference on Software Maintenance, Reengineering and Reverse Engineering, 2014*, pages 44–53, 2014.

[54] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson, "Use, disuse, and misuse of automated refactorings", In *Proceedings of the 34th International Conference on Software Engineering*, pages 233–243, 2012.

[55] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Roshanak Zilouchian Moghaddam, and Ralph E Johnson, "The need for richer refactoring usage data", In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and Usability of Programming Languages and Tools*, pages 31–38, 2011.

[56] Anthony J Viera, Joanne M Garrett, et al., "Understanding interobserver agreement: the kappa statistic", *Fam Med*, 37:360–363, 2005.

[57] Edmund Wong, Jinqiu Yang, and Lin Tan, "Autocomment: Mining question and answer sites for automatic comment generation", In *Proceedings of the 28th International Conference on Automated Software Engineering*, pages 562–567, 2013.

[58] Zhenchang Xing and Eleni Stroulia, "Refactoring practice: How it is and how it should be supported- an Eclipse case study", In *Proceedings of the 22nd International Conference on Software Maintenance*, pages 458–468, 2006.

[59] Jinqiu Yang and Lin Tan, "SWordNet: Inferring semantically related words from software context", *Empirical Software Engineering*, 19:1856–1886, 2014.

[60] Annie TT Ying, James L Wright, and Steven Abrams, "Source code that talks: an exploration of Eclipse task comments and their implication to repository mining", In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[61] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall, "Analyzing APIs documentation and code to detect directive defects", In *Proceedings of the 39th International Conference on Software Engineering*, pages 27–37, 2017.