View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Digital Commons@Wayne State University

Wayne State University

Wayne State University Theses

1-1-2013

On The Relationship Between The Vocabulary Of Bug Reports And Source Code

Amunugamage Buddhini Wathsala Bandara
Wayne State University,

Follow this and additional works at: http://digitalcommons.wayne.edu/oa_theses

Part of the Computer Sciences Commons

Recommended Citation

Bandara, Amunugamage Buddhini Wathsala, "On The Relationship Between The Vocabulary Of Bug Reports And Source Code" (2013). *Wayne State University Theses.* Paper 289.

This Open Access Thesis is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Theses by an authorized administrator of DigitalCommons@WayneState.

ON THE RELATIONSHIP BETWEEN THE VOCABULARY OF BUG REPORTS AND SOURCE CODE

by

AMUNUGAMAGE WATHSALA BANDARA

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2013

MAJOR: COMPUTER SCIENCE

Approved by:

____

Advisor Date

DEDICATION

To my parents and husband for their endless love, support and encouragement throughout my life.

ACKNOWLEDGEMENTS

It is a great pleasure to show my gratitude to all those who helped me in numerous ways to make this thesis successful. Without their support this thesis might not have been written.

First and foremost, I would like to express my sincere gratitude to my academic advisor Prof. Andrian Marcus for his continuous support on my Masters study and thesis. His guidance, support, knowledge, patience, and supervision helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank my thesis committee members. My sincere thank goes to Prof. Václav Rajlich for the knowledge I have gained about Software Engineering and for being open up to help regarding the subject matters. It is my pleasure to work with Prof. Chandan Reddy during my Masters program and I wish to express my deepest gratitude for the support and knowledge I acquired from him.

I have been blessed to have friends like Sonia Haiduc and Laura Moreno since the beginning of my studies. The immense help, guidance and constructive comments I've received from Sonia were a great help to succeed this thesis. It was a great pleasure to work with Laura and the support I received from her was indescribable. I'm sincerely grateful to them for all the things they have done for me.

I thank my friends Oscar Chaparro, Asha Bandara and Zeyad Hailat for their comments and advices about thesis writing. The feedback I've received from them was undeniable as it was a great advantage to improve my work. I'm also thankful for Nariman Ammar for her willingness to help me and for providing me supporting materials at the very beginning of writing my thesis.

Last but no means least, I'm so grateful for my husband Ruchira Liyanage for his comments regarding my writing. His constant support, patience, devotion and love always were a great motivation for me to get through the hard times of my thesis.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS v

LIST OF FIGURES viii

LIST OF TABLES ix

LIST OF CODE SEGMENTS xii

CHAPTER 1 INTRODUCTION 1

1.1. Motivation 4

1.2. Contribution 6

1.3. Thesis Statement 7

1.4. Thesis Outline 7

CHAPTER 2 BACKGROUND ON TEXT RETRIEVAL 10

2.1. Corpus Creation 11

2.2. Text Pre-processing 13

2.2.1. Tokenization and Compound Terms Splitting 13

2.2.2. Stop Word Removal 16

2.2.3. Stemming 16

2.3. Indexing and Query Formulation 18

2.4. Ranking and Investigation of Results 20

CHAPTER 3 STUDY OF THE RELATIONSHIP BETWEEN BUG REPORTS AND SOURCE CODE 21

3.1. Research Questions 21

3.2. Data Collection 22

3.3. Automating Text Retrieval Process 24

3.4. Automated Corpus Creation 25

3.4.1. Determination of the Granularity 25

3.4.2. Noise Reduction 28

3.5. Automated Text Pre-processing 28

3.5.1. Automated Tokenization and Compound Terms Splitting 28

3.5.2. Automated Stop Word Removal 29

3.5.3. Automated Stemming 31

3.6. Creating Vocabularies 31

3.6.1. Annotated Terms Vocabularies 32

3.6.2. Not-annotated Vocabularies 35

3.7. Planning of Research Questions 38

CHAPTER 4 ANALYSIS OF THE RESULTS AND DISCUSSION 42

4.1. Observations about the Document Size 42

4.1.1. Bug Reports 42

4.1.2. Source Classes 43

4.2. Addressing Research Questions 43

4.2.1. Research Question 1 43

4.2.2. Research Question 2 47

4.2.3. Research Question 3 49

4.2.4. Threats to Validity 51

CHAPTER 5 RELATED WORK 53

CHAPTER 6 CONCLUSIONS AND FUTURE WORK 55

6.1. Conclusions 55

6.2. Future Work 56

APPENDIX A: Summary of the Data Set 57

APPENDIX B: Stop Words 59

APPENDIX C: Summary of the Terms in Code Locations 62

APPENDIX D: SrcML Document 68

APPENDIX E: Summary of the Terms in Bug Reports 70

References 73

ABSTRACT 77

LIST OF FIGURES

Figure 1-1: Searching for concept location in the source code (adapted from [1])………….. 2

Figure 1-2: Bugs of JEdit reported in SourceForge bug tracking system……………………. 3

Figure 1-3: Sample bug report of aTunes system………………………………………………. 4

Figure 1-4: Extraction of common words in source code and bug reports of a software system………………………………………………………………………………………….. 7

Figure 2-1: Framework of an information retrieval system (adapted from [8])……………... 11

Figure 2-2: Text pre-processing steps……………………………………………………………. 13

Figure 3-1: The count of the related patch classes for the bug reports…………..………… 24

Figure 3-2: Vocabularies based on location information……………………………………... 32

Figure 3-3: The automated process of creating locater terms and not-annotated vocabularies…………………………………………………....................................... 37

Figure 4-1: Common vocabulary size of bug reports and source classes in each system without outliers……………………………………………………………………….…………… 45

Figure 4-2: Simpson similarity index between bug reports and source classes of each software system………………………………………………………………………………… 46

Figure E.1: The summary of the bug reports in all the software systems...………………... 70

Figure E.2: The total number of terms of the bug report in each system……..……………. 71

Figure E.3: The unique terms (%) of the bug reports in each software system……..…….. 72

LIST OF TABLES

Table 2-2: Compound terms vocabulary……………………………….……………… 16

Table 2-3: Document-term matrix 18

Table 3-1: Software systems used for the study of common vocabularies……………… 22

Table 3-2: Source code locations……………………………………….……….…… 33

Table 3-3: Annotated terms vocabulary………………………….…….…..………… 33

Table 3-4: Bug title and description extracted from Art of Illusion bug #34 ……………… 34

Table 3-5: Not-annotated vocabulary……………………………….…………...…… 35

Table 3-6: Shared words (without locations) between source class (left) and bug report (right) given in Code Segment 3-3……………………………………………….………… 38

Table 3-7: The common vocabulary (the intersection between source class and bug report illustrated in Table 3-5)……………………………………………..…..…… 39

Table 3-8: Shared words (with locations) between source class (left) and bug report (right) given in Code Segment 3-3………………………………………………..……… 40

Table 4-1: Categorization of the percentage of pairs (common vocabularies) by the number of shared terms………………………………………………………..……… 44

Table 4-2: Shared terms statistics in code locations in the patched classes…….…… 48

Table 4-3: Number of cases that the share terms approach is better, equal or worse than LSI and Lucene.. 50

Table 4-4: Average and median effectiveness of the Shared Terms (ST), LSI and Lucene approaches…………………………………………………………..……… 50

Table A.1: The total number of unique terms in the source classes (class vocabulary size)... 57

Table A.2: The total number of terms in the source classes (class document Size)……. 57

Table A.3: The total number of unique terms in the bug reports (bug vocabulary size)... 57

Table A.4: The total number of terms in the bug reports (bug document Size)…....……. 58

Table A.5: The total number of unique terms shared between the bug reports and the patched classes (common vocabulary size)……………………………………………… 58

Table A.6: The total number of terms shared between the bug reports and the patched classes………………………………………………………………………………….. 58

Table C 1: Shared terms in arguments……………………………………………….... 62

Table C 2: Shared terms in attributes. ………………………………………..…... 62

Table C 3: Shared terms in comments.. ………………………..……………………… 62

Table C 4: Shared terms in literals.. ………………………………………………..….. 63

Table C 5: Shared terms in method calls……………………………………………….. 63

Table C 6: Shared terms in method name.. ………………………………….……... 63

Table C 7: Shared terms in parameter. ………………………………..………… 64

Table C 8: Shared terms in types. ……………………………………………….…... 64

Table C 9: Shared terms in variables.…………………………………...……………… 64

Table C 10: Total terms in arguments..………………………………..……………… 65

Table C 11: Total terms in attributes. …………………………………..……………… 65

Table C 12: Total terms in comments. ………………………………………………….. 65

Table C 13: Total terms in literals……………………………..……………..…... 65

Table C 14: Total terms in method calls. ……………………………………………….. 65

Table C 15: Total terms in method names…………………………………..…………. 65

Table C 16: Total terms in parameter.. ………………………………………………… 67

Table C 17: Total terms in types. ……………………………….………………....... 67

Table C 18 Total terms in variables...…………………………………………………… 67

Table E.1: The total number of terms of the bug report in each system. …………...…… 71

Table E.2: The unique terms (%) of the bug reports in each software system................ 72

LIST OF CODE SEGMENTS

Code Segment 2-1: Granularity of a Source File.. 12

Code Segment 2-2: Compound Terms in Source Code.. 15

Code Segment 3-1: Nested Classes.. 25

Code Segment 3-2: Eliminations of content in source file - Package declaration, import statements, comments and annotations.. 26

Code Segment 3-3: Stop words (strikethrough) shared between code snippet (Left) and bug report (Right).. 29

Code Segment 3-4: A simple Java method.. 33

CHAPTER 1 INTRODUCTION

Software defects or bugs are expensive and pervasive throughout software. The number of defects left in the code is an important measure of software quality [1]. As software development occupies a large amount of human effort, there is no guarantee of producing bug-free software with human intervention. By studying the US software population in development and maintenance, the work of bug fixing has become one of the dominant form of software engineering since the beginning of software development [2]. In the software life span, the bug fixing is performed in certain stages, which leads to the production of quality software. According to the staged model [3], software evolution and servicing stages play an important role in bug fixing process. Adding new features to the system and correction of existing mistakes take place iteratively during the software evolution stage while the software servicing stage entirely focuses on corrections of software faults [3]. As the software developers spend most of their development time on post-delivery activities, such as, bug fixes, numerous studies have been conducted to investigate the different types of software maintenance cost. According to the study carried out by Lientz and Swansons on 487 data processing organizations, defect repairs hold 20% of the total maintenance effort [4].

Hence, locating a bug in a source file is as important as all the other tasks involve in software development. As the base of software evolution and servicing, software change consists of several important phases, such as, initiation, concept location, impact analysis, refactoring, actualization, verification, and conclusion [1]. Among these phases, determining the code location to begin the software change, which is called concept location, is one of the activities undertaken by the developers during the software evolution and servicing. The

concept location is a search process (Figure 1-1) that aims to locate the code snippet where the developer has to modify according to the change request [1].

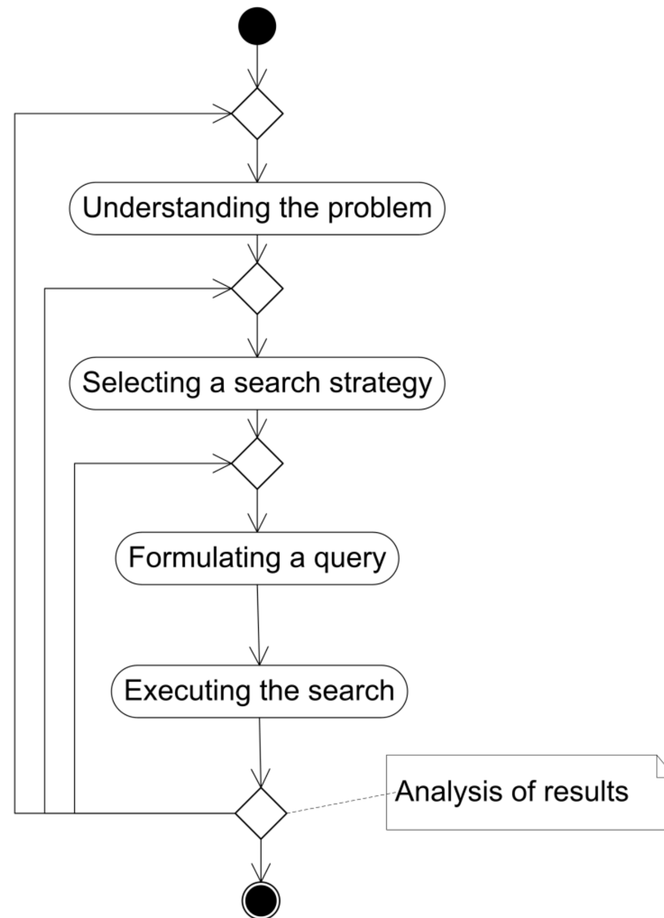


Figure 1-1: Searching for concept location in the source code (adapted from [1])

Software change is initiated with a change request. The change request is a document that demands for a modification of the system, which indicates what has to be accomplished, but leaves out the fact how the adjustment should be made. The change request can be a new feature to be added to the existing system, or a modification to be made to an existing feature, or a bug report. If the request is a bug report, then it is the developer's responsibility to locate the reported bug in the relevant source code and take the necessary actions to fix the problem. Finding the precise location in the defect source

code where the bug has been occurred, is called bug localization or bug location which is an instance of concept location.

Bugs are reported by software users in corresponding bug tracking systems (Figure 1-2). A bug report contains information, such as, bug identification number (bug ID), bug title, bug description, bug type, priority, software versions affected and fixed, current status of the bug, reporter of the bug (a tester or a user), assignee of the bug (developer) etc. (Figure 1-3).

Assignee: Any Status: Any Category: Any Group: Any Submitter:
Keyword: Artifact ID: 0 Filter Reset Permalink

ID	Summary	Status	Opened	Assignee	Submitter	Resolution	Priority
727042	Zombie popup menus after shortcut edit	Closed	2003-04-24	nobody	selimnairb	Invalid	5
2929554	zero-width positive lookbehind assertion	Open	2010-01-10	nobody	https://www.google.com/accounts	None	5
876765	Zero length files are detected as unix files.	Closed	2004-01-14	nobody	nobody	Fixed	5
665335	XSL Transformation fails with Reason "null"	Closed	2003-01-09	robmckinnon	nobody	Fixed	5
735314	XSL Transformation fails after SideKick error	Deleted	2003-05-09	robmckinnon	nobody	Duplicate	5
693019	XSLT plugin's Indent XML command & whitespace	Closed	2003-02-25	robmckinnon	rfletch6	Fixed	5
690580	XSLT plugin: Problem with 'Indent XML'	Closed	2003-02-21	robmckinnon	lehmannm	Fixed	5
557928	XSLT plugin looses the xslt processor.	Closed	2002-05-19	nobody	nobody	Out of Date	5
681804	XSLT indent function modifies element contents	Closed	2003-02-06	robmckinnon	rgs17	Fixed	5
672429	XSLT Indent : CDATA causes multiplying newlines	Closed	2003-01-22	robmckinnon	robertito	Fixed	5
564356	XSLT - ArrayIndexOutOfBoundsException	Closed	2002-06-04	greghmerrill	odlouhy	Wont Fix	5
674409	XSLT 0.4.* & JDK 1.4.*: exceptions when transforming	Closed	2003-01-24	robmckinnon	robmckinnon	Accepted	7
672977	XSLT 0.4.1 - unknown protocol: d	Closed	2003-01-23	robmckinnon	leeturner	Works For Me	5

Figure 1-2: Bugs of JEdit reported in SourceForge bug tracking system [1]

Determining the bug location in the source code is one of the main tasks of the software developers. Out of all the attributes of a bug report, the bug title and the bug description carry the most prominent information for the bug localization process. Due to

[1] http://sourceforge.net/tracker/?group_id=588&atid=100588

that reason, the bug title and the description are considered as one of the dominant sources in studies that examine the relationship between the defect source files and the corresponding bug reports.

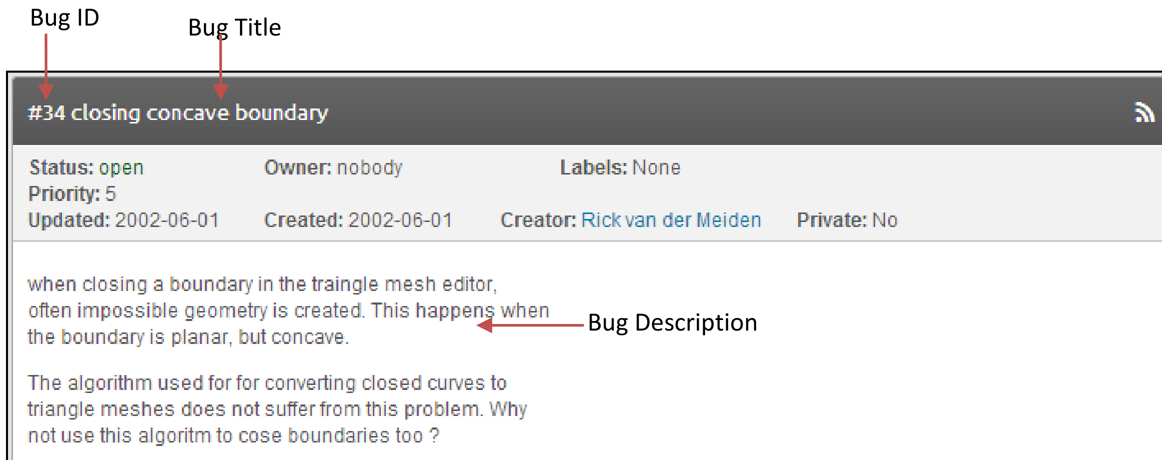


Figure 1-3: Sample bug report of aTunes system

1.1. Motivation

The bug localization is a challenging task due to numerous reasons. One of them is that the initial developers of the source code might not be the ones to fix the bugs reported in latter times, as most of the software projects continue for a long time period. In that case, the assignee of the bug report may not be familiar with source code implementation and he or she may not be able to receive any support from the author of the program, which makes locating the bug more complicated. Sometimes bugs are reported after a considerably long period of time since the delivery of the software. So the author of the defect source code may not remember all the works he or she has done before. Such a situation can make the bug localization a difficult task, even if the author of the source code is assigned for the bug fixing. In addition to the above reasons, lack of sufficient documentation of the software system hinders the opportunity of saving search time of the fault location in the program.

Hence, locating the bug is one of the challenging tasks for its assignees, which costs a major amount of bug fixing effort in terms of monetary and time.

Various approaches are available for concept location, which can be used in bug localization process. Pattern-matching is one of them that checks a sequence of tokens for the presence of the piece of pattern such that the match is precise. Grep is one of the popular concept location techniques used for program comprehension [5], based on pattern-matching approach. Grep is a tool that allows developers to iteratively formulate queries in the form of regular expressions and query the source files. The tool outputs a set of matching lines, but it is the programmer's responsibility to find the concept location by studying the surrounding lines of the code and decide whether the actual location of the concept is found. The tool iterates the process until the desired output is obtained. There is a family of Grep tools that has been developed with some additional options. The agrep [6], egrep [7] and fgrep [7] are few examples. Beside the pattern-matching approach, several text retrieval (TR) based approaches are proposed to partially automate the task of bug localization [8].

Text retrieval is a process of matching text documents against the user formulated queries which differs from text searching. The difference between searching and retrieval is, the outcome of searching is an exact match to the query, which indicates whether the match is found or not. In contrast to searching, retrieval process may obtain more than one solution, ordered by their relevancy to the query, i.e., the outcomes of retrieval do not need to be precisely matched with the query. Both pattern-matching and text retrieval approaches rely on natural language queries, often formulated either manually or automatically, based on the descriptions in the bug reports. Some TR based techniques show many advantages

over pattern-matching approach regarding the concept location, which provide better results than gerp.

The use of TR techniques for the bug localization is based on the assumption that a bug report and its related fault code share an important vocabulary. This assumption is important to create an “effective query” that yields more accurate outcome. In order to construct such a query, the user must be able to predict the query terms in the form of words, phrases and combinations of words such that most of these terms occur in the relevant documents while they do not occur in most of the non-relevant documents [9]. Therefore, certain meaningful words that describe the bug are used in the query to generate more relevant results. We assume these words are shared between the bug and the corresponding fault code. Even though, there are no studies reported to show evidence to support this assumption, it has been shown that such information is beneficial to improve TR based techniques used for the bug localization [10].

1.2. Contribution

Our contribution is to acquire evidence to support the implicit assumption that bug descriptions and the corresponding source code share some significant words which help to map the bug and the defect code. As this assumption has been the base for text retrieval techniques used for bug localization, its assurance would help to enhance the TR based techniques used to determine bug locations in relevant source files. To achieve this task, we proposed a technique to explore the common vocabularies obtained from the bug reports and the patched classes (Figure 1-4). We analyzed these vocabularies to identify significant patterns and the existence of the relationships between bug reports and source code. The results obtained from this analysis can be used to find evidence to support the idea of using

the common vocabularies with text retrieval techniques to determine bug location and ways to improve TR based techniques.

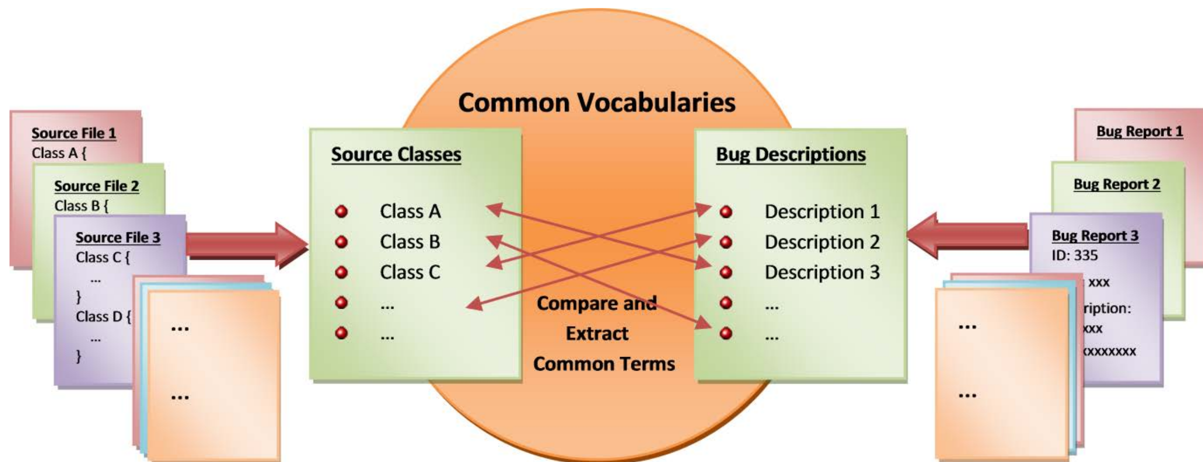


Figure 1-4: Extraction of common words in source code and bug reports of a software system

1.3. Thesis Statement

The following thesis statement is supported by all the contributions made by this thesis as explained in Section 1.2.

Text retrieval techniques are applied on bug localization process based on an implicit assumption that the bug description and the corresponding source code have a common vocabulary. As there is no considerable study has been conducted to find evidence to prove the fidelity of this assumption, such important information is useful to enhance the text retrieval approaches for the bug localization process.

1.4. Thesis Outline

The following chapters discuss in detail the research tasks carried out in this thesis.

Chapter 2 explains the background information related to information retrieval and text retrieval, a subsidiary area of study of information retrieval. It presents important steps involved in text retrieval process under the subsections. The description of the text retrieval process begins with corpus creation which is explained in Section 2.1. Then the text pre-processing techniques, such as, tokenization and compound terms splitting, stop word removal, and stemming are explained under Section 2.2. A description of the indexing and query formulation steps is explained in Section 2.3 which presents two of the most popular indexing methods, Vector Space Model (VSM) and Latent Semantic Indexing (LSI). At the end, Section 2.4 elaborates how the searched documents are ranked depending on their relevance to the query and how to investigate the results obtained.

Chapter 3 presents information about the thesis study. Thesis questions are listed and discussed under Section 3.1. Section 3.2 describes the data used for the thesis study, i.e., bug reports and source files collected from the different software systems. In this study, we build a tool to automate the text retrieval process described in Chapter 2 and to created vocabularies. Under the Section 3.3, we describe this automated process. Section 3.4 describes the corpus creation using the tool, which indicates how the documents are obtained from the source files and the bug reports according to the desired granularity.. Section 3.5 presents the automated text pre-processing steps, such as, tokenization and compound terms splitting, stop word removal and stemming. As our study focuses on the use of common vocabularies in TR process, Section 3.6 describes the automated process of creating the vocabularies for the bug reports and the source code. The measures obtained from the data collection are presented under Section 3.7. Furthermore, how we planned to address the thesis questions is discussed in the same section.

Chapter 4 involves a general discussion of the results and findings of the empirical study that we discussed in Chapter 3. Section 4.1 describes the observation the sizes of the bug and source code documents. The execution of the study design is described in Section 4.2 which discusses the answers obtained for each research question listed in Chapter 3. As it is important to identify the threats that can affect our conclusions, Subsection 4.2.4 describes four types of threats that affect to any research study, namely construct validity, internal validity, external validity, and conclusion validity. Furthermore it discusses how we mitigate the influence of these threats in our study.

Chapter 5 presents some other works carried out related to the study we discussed in this thesis. There are several studies have been conducted to understand how the parts of the speech of the words can be used when describing bug reports. Some studies have built models to categorize bug reports based on different facts, such as, the part of the speech, the word frequency and the distribution across different severity levels. In addition to that, this chapter discusses the linguistic and statistical studies, focus on topics analysis and coherence analysis of bug reports.

Chapter 6 describes the conclusions drawn by the empirical study that we carried out (under Section 6.1). Furthermore, it summarizes the results we obtained. Section 6.2 discusses the importance of validating our work and how to carry out the validation in future. Section 6.2 describes how to extend this work and which kind of improvements have to be made to the current process to reach the expected outcome.

CHAPTER 2 BACKGROUND ON TEXT RETRIEVAL

Text retrieval is a branch of information retrieval (IR) which is a prominent method of information access in present. In the academic field of study, information retrieval is defined as finding unstructured text documents that satisfies an information need within large collections that are stored on computers [11]. Although the information retrieval has been employed over decades, the research interest on this field began rising steadily since 1950s [12, 13]. The wide use of information retrieval systems in modern search engines, are found in commercial and intelligence applications as long ago as the 1960s [12]. For instance, the STAIRS (Storage and Information Retrieval System) developed at IBM in the late 1950's, was a turning point in the field of information retrieval research studies [9, 14]. Interaction with IR systems begins with an information problem which leads to an information need. Information need means what the user needs to know more about a certain topic [11]. The query is a formal statement of the information need, which is created by the user who has the requirement of solving the information problem. This query is compared with the representation of the text and this process may result in multiple matches to the query with different degrees of relevancy.

As a subfield of information retrieval, text retrieval follows a similar framework on information presented in the form of text (Figure 2-1). In other words, text retrieval is an activity of matching text documents against the use-r formulated queries. The usage of text retrieval techniques in software engineering tasks has been drastically increased recently due to the remarkable benefits that it yields on the subject of retrieving textual information in numerous software artifacts, such as, requirement specifications, source code, design and technical documentations and user manuals [15].

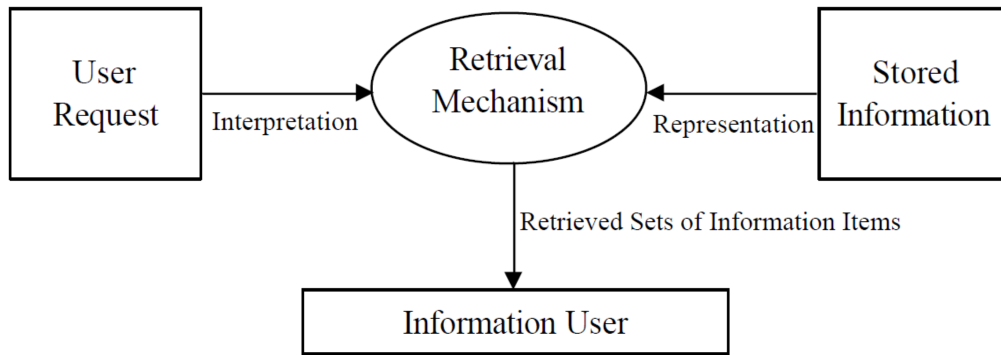


Figure 2-1: Framework of an information retrieval system (adapted from [16])

The following subsections describe some essential steps involved during the text retrieval process.

2.1. Corpus Creation

Text retrieval techniques are applied on the text content stored in a batch of documents. As all the text content of these documents are not important for the retrieval purpose, the first step determines the granularity or the level of details required from the document [1, 11, 17]. For instance, granularity of a user manual can be defined as a chapter, a sub section, a paragraph, a sentence etc. and for a source file, it can be defined as a class, a method, a code block, a line of code etc. [17]. In terms of the bug reports, bug title, bug description, comments, attachments, such as, patched files and error messages, etc. can be used as the components to define the level of details of a certain software defect. Determining the granularity of a document is important, because the results of text retrieval process rely on the level of details we select. This idea can be depicted using the class definition presented in Code Segment 2-1. If we consider the granularity of this source code as a class definition, then the number of occurrences of the word “shape” is seven. But

if the granularity is defined as a method definition, then the frequency of the word “shape” in each method, is one.

```
1.  class Shape {
2.
3.      private int shape_id;
4.      private string shape_name;
5.      private Type shape_type;
6.
7.      public void Shape () {
8.
9.          // empty block
10.
11.     }
12.
13.     public void Shape (string name) {
14.
15.         // empty block
16.
17.     }
18.
19.     public void Shape (string name, int x, int y) {
20.
21.         // empty block
22.
23.     }
24. }
```

Code Segment 2-1: Granularity of a Source File

During the corpus creation, some of the text content found in the document may not be relevant for the text retrieval process as it leads to inaccurate results. On the other hand, the contribution of some artifacts may distort the actual results. Hence, such irrelevant elements are eliminated from the text documents.

2.2. Text Pre-processing

The corpus obtained during the previous step, is normalized by applying the text pre-processing techniques, to obtain accurate results. Following subsections describe common text pre-processing steps used in TR process.

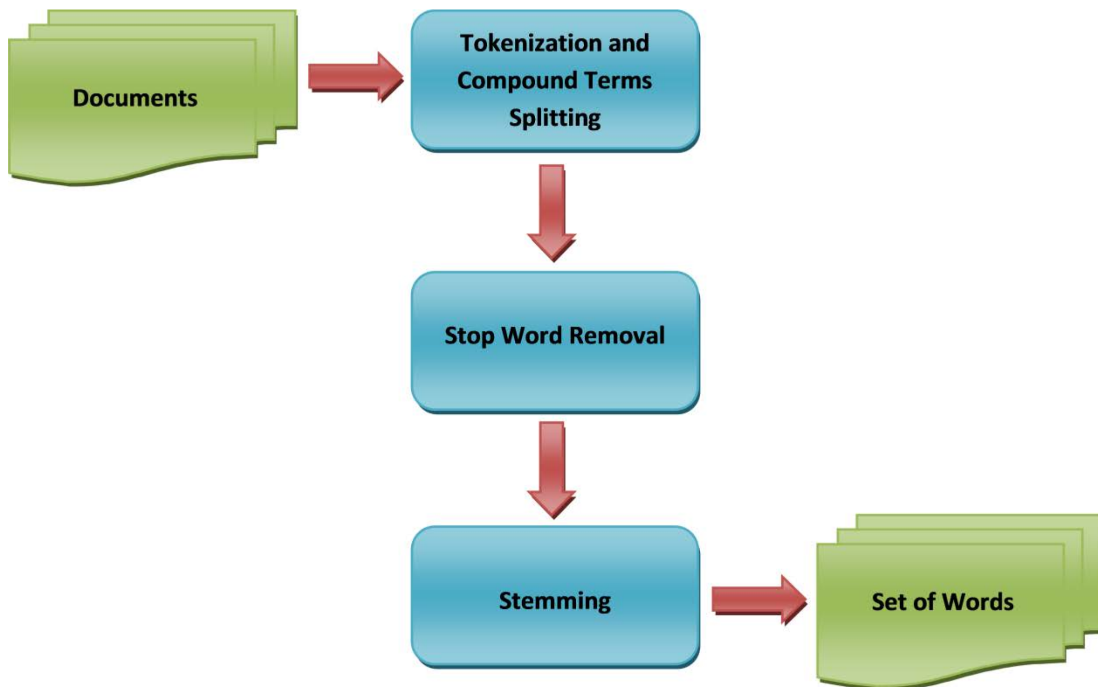


Figure 2-2: Text pre-processing steps

2.2.1. Tokenization and Compound Terms Splitting

Tokenization is a process of breaking down a stream of characters into pieces called tokens. A token can be a word, a phrase, a symbol or any other atomic unit of a language, which conform to a set of certain syntax properties. In tokenization process, a particular text can be disjoined into words or meaningful elements by eliminating white spaces, line breaks, punctuation, brackets, and other delimiters, such as, hyphen, underscore, comma, etc. For

example, the compound text “color-red, number_two” can be divided into four single words such that “color”, “red”, “number”, and “two”.

Compound terms are formed as a result of composition of two or more words, which can be generated by following a particular naming convention. For instance, according to the convention established by Sun Microsystems, Java class names and method names follow CamelCase notation and variables follow lowerCamelCase style. Each word in a compound term that followed the camel case style is separated by an uppercase letter. For instance, the compound term “CamelCase” can be split into two words such that “Camel” and “Case”. A compound term that follows Hungarian notation always begins with a prefix that encodes the actual data type or a mnemonic that describes the purpose of the variable. The first letter of the prefix is always a lower case letter and all the other words in this compound term begin with upper case letters. For example, the compound term “strHungarianNotation” can be broken down into three words such that “str”, “Hungarian”, and “Notation”.

Since words can be combined without any restrictions, it increases the vocabulary size. This vast increment of the vocabulary size leads to the sparse data problem. Table 2-1 and Table 2-2, created from Code Segment 2-2, illustrate this issue. If the vocabulary of this code segment neglects keywords, then the size of compound terms vocabulary will be much larger than the size of single terms vocabulary. This situation could be understood by looking at Table 2-1 and Table 2-2. Table 2-2 contains more words compared to the Table 2-1, but with less frequency. Hence, the compound terms vocabulary results the sparse data problem

```
1. class Shape {
2.
3.      private String shapeName;
4.      private String shapeType;
5.
6.       public void Shape () {
7.
8.            // empty block
9.
10.      }
11.
12.      public void printShapeName (String name) {
13.
14.          // empty block
15.
16.      }
17.
18.      public void printShapeType (String type) {
19.
20.          // empty block
21.
22.      }
23. }
```

Code Segment 2-2: Compound Terms in Source Code

Table 2-1: Single terms vocabulary

Word	Frequency
shape	6
name	3
type	3
print	2

Table 2-2: Compound terms vocabulary

Word	Frequency
Shape	2
shapeName	1
shapeType	1
type	1
name	1
printShapeName	1
printShapeType	1

2.2.2. Stop Word Removal

Words which are filter out from the text documents due to their unproductiveness and irrelevance to the text retrieval process, are known as stop words. This filtering may take place prior or after the text pre-process. In English, many of the frequently used words are considered as stop words, such as, “a”, “the”, “of”, “and”, etc. Access modifiers (e.g. “private”, “public”), primitive data types (e.g. “int”, “boolean”, “double”), control flow statement (e.g. “if”, “for”, “switch” “while”, “do”) and keywords of exceptions, class and interface declarations (e.g. “catch”, “exception”, “class”, “interface”) are few of the example for the keywords found in the programming languages. Since the stop words are considered as noise, removing them from the text documents reduces the indexing size. And the reduction of indexing size improves the efficiency of text retrieval process.

2.2.3. Stemming

Stemming is a process of reducing all the words with the same root (stem), to a base form, by chopping each derived and inflected word. For example, “give”, “gives”, “gave”, given” and “giving” are forms of the same lexeme as they have a similar semantic interpretation. In spite of all these morphological variants, they can be written as “give”, in general. In the text retrieval process, stemming is beneficial, because it reduces the

vocabulary size of the corpus or indexing files, which prevents having sparse data problem. Hence, stemming conduces to improves retrieval effectiveness by matching more words in a document despite of their lexical form.

Different varieties of methods have been followed to address the problem of conflation, such as, affix removal, character string truncation, letter bargain, word segmentation and linguistic morphology [18]. Affix removal algorithms are the most common methods among them, which remove suffixes or prefixes from the words that form the same meaning of the stem [19]. The Lovins stemmer (1968) and the Porter stemmer (1980) are two of the most common suffix removal stemming algorithms used in information retrieval. Julie Beth Lovins has published the first stemmer in 1968 which was a great influence for later works related to this area. Then, Porter stemmer written by Martin Porter in 1980 was widely used for English stemming, which consists of about 60 rules. The Porter stemmer has been created based on the fact that the most of the suffixes in the English language have been built up by the conflation of smaller and simpler suffixes. As a result of the extension of his work, the Snowball framework has been built for writing stemming algorithms not only for English, but also for other languages, such as, Romance, Germanic, Russian, Turkish, Uralic and Scandinavian. According to the conclusions of Chris Paice, the error rate of the Porter stemmer is less than Lovins stemmer [20].

There are two main errors found in stemming which are under-stemming and over-stemming. Under-stemming occurs when two words with the same morphological interpretation are not stemmed to the same root. That is known as a false-negative. Due to the loss of matching words, under-stemming causes the low sensitivity by spreading a single concept over variety of different stems. On the other hand, over-stemming means stemming

two words with different stems to the same root which is known as a false-positive [20]. Over-stemming results lower precision due to the dilution of the stems' meaning.

2.3. Indexing and Query Formulation

Indexing is a process of creating systematic arrangement of entries that are used to locate information in a document. This arrangement of entries is called an index. The corpus indexing maps the documents in the corpus with the frequency of unique terms occurring in the documents. The document-term matrix is a mathematical representation of an indexing, which represents the documents as rows, the unique terms as the columns and the frequency of each term in a particular document in the matrix cells. Table 2-3 illustrates the idea behind the document-term matrix.

Table 2-3: Document-term matrix

	Term 1	**Term 2**	**Term 3**	**…**	**Term N**
Document 1	t_{11}	t_{12}	t_{13}	…	t_{1N}
Document 2	t_{21}	t_{22}	t_{23}	…	t_{2N}
…	…	…	…	…	…
Document M	t_{M1}	t_{M2}	t_{M3}	…	t_{MN}

There are popular models used to represent the document-term matrix, such as, Vector Space Model (VSM) and Latent Semantic Indexing (LSI). Both VSM and LSI render text by following the bag of words model which presents the text as a collection of words, regardless of the order or grammar. VSM is an algebraic representation of the document-term matrix, which is developed by Gerard Salton in 1975. This model represents the documents and the user queries in the form of vectors as follows. d_j represents the document vector and q represents the query vector.

$$d_j = (t_{1i}, t_{1i}, \ldots, t_{1i}) \quad \textbf{Equation 2} - \textbf{1: Document Vector}$$

$$q = (t_{1j}, t_{1j}, \ldots, t_{1j}) \quad \textbf{Equation 2} - \textbf{2: Query Vector}$$

In VSM, the term is a basic concept, such as, a single word, set of words or a phrase. Each of these terms defines a distinct dimension in n-dimensional space. Each entry in the matrix corresponds to the weight of a term in the document. Different methods are followed to decide the weight, such as, term frequency (TF) and inverse document frequency (IDF). If a term exists in a specific document, then the value corresponds to that document in the relevant dimension is non-zero. If the term is not found in the document, then that value becomes zero.

In general, the query vector is treated as a document, so the same process is followed regarding the query vector. Completion of the query and the document representation, leads to the requirement of finding the similarity between the document vectors and the query vector. This can be calculated by using Cosine Similarity. VSM is a simple and efficient model, which facilitates ranking documents according to their relevance. But it may carry some drawbacks. One of them is that the multiple words refer to the similar context. This is called synonymy which leads to poor recall. Another major problem in VSM is polysemy which causes by the words having more than one distinct meaning. This problem leads to poor precision.

Latent semantic indexing (LSI) was proposed to overcome the issues of synonymy and polysemy in VSM [21]. This technique is used to analyze the relationships between a collection of documents and the terms which are represented as term-document matrix. As the initial step in LSI process, term-document matrix should be constructed and then weighting functions are applied to the matrix.

2.4. Ranking and Investigation of Results

At the end of the query formulation followed by computation of semantic similarities between the query and each document in the corpus, a list of documents is retuned as the output of the retrieval process. The documents listed in the results set are ranked according to their relevancy to the query, formulated starting from the most relevant document to the least relevant one. The similarity measure chosen during the indexing process is used for this ranking procedure, hence, it is important to select the matching similarity measure to obtain better results.

The ordered list of documents is studied to ensure whether the relevancy of the documents with the highest ranks, really contain the information that we are looking for, i.e., checking the existence of the relevancy between that document and the query. It's programmer's responsibility to decide whether the documents that he or she is looking for are listed among the set of top ranked results. If the certain documents are not found in that set, then a new query has to be reformulated with different keywords and follow the ranking procedure. If the desired results are produced, then the search is considered as successful and the process is terminated.

CHAPTER 3 STUDY OF THE RELATIONSHIP BETWEEN BUG REPORTS AND SOURCE CODE

As we discussed before, most of the TR based techniques are applied on the bug localization process based on the implicit assumption that the bug reports and the relevant source files share a common vocabulary. We conducted this empirical study with the objective of exploring the accuracy of this assumption. We analyzed the common vocabularies of the bug reports and the source code to find evidence to support this assumption.

3.1. Research Questions

To accomplish our objective, we address the following research questions during this study:

RQ1 To what extent are the vocabularies of bug reports reflected in the identifiers and comments of classes?

The use of the effective queries helps to save a lot of retrieval time. To formulate such an effective query, it is important to understand how well a bug report is described in the relevant fault code. Therefore, we built the research question RQ1 to find out how well bug vocabularies reflect the content of source classes.

RQ2 What is the code location (i.e., class name, method name, attribute name, etc.) of the shared words between bug reports and patched classes?

The words occurring in certain code locations of the source code may appear more frequently in the bug reports than the other code locations. When formulating the

queries, it is beneficial to have this information as it helps to improve the bug localization. We built RQ2 to find out the existence of such code locations.

RQ3 Is the number of shared words between bug reports and classes, an adequate measure to support bug localization?

We use the terms share between the bug reports and the source code as a measurement when addressing the research questions RQ1 and RQ2. It is important to determine the accuracy of our approach as it influences the conclusions we draw. Therefore, RQ3 was formulated to assess this measurement.

The planning of these research questions are described in Section 3.7 in detail.

3.2. Data Collection

Table 3-1: Software systems used for the study of common vocabularies

System	Version	Number of Classes	Number of Bug Reports	Number of Patched Classes
ADempiere	3.1.0	1,896	16	16
Art of Illusion	2.4.1	570	10	13
aTunes	1.10	439	17	22
Eclipse	2.0	7,689	13	14
Eclipse	3.5	22,980	40	74
JEdit	4.2	801	18	27
Total		34,376	114	166

The objective of our study is to explore the vocabularies of the bug reports and the source code to assess the existence of any relationship between them. To follow this examination, we used the bug reports and the source files that belong to six open source software systems listed in Table 3-1. The source code belong to the given systems and the versions, were downloaded from their online repositories. We extracted the titles and

descriptions from the bug reports of the same versions manually and stored them in a repository for future use.

These Java based software systems belong to different problem domains and they have been used for previous studies [22, 23], conducted regarding the text retrieval approach. ADempiere[2] is an industrial strength open source software solution that addresses business areas, such as, enterprise resource planning (EPR), supply chain management (SCM), customer relationship management (CRM) and point of sale (POS) solution. ADempiere 3.1.0 version used in our study was released on Oct 16, 2006. Art of Illusion[3] is an open source 3D modeling and rendering studio, which is a Java language based free software package. Art of Illusion 2.4.1 version was released on Feb 28, 2007. aTunes[4] is a free open source audio player and organizer, which supports audio file formats, such as, mp3, Ogg, wma, flac, wav and mp4. It facilitates the users to edit tags, organize playlist and rip audio CDs easily. This full-featured software system is implemented in Java language and its version 1.10 was released on Sep 27, 2008. Eclipse[5] is a multi-language integrated development environment (IDE), which is written mostly in Java programming language. It consists of a base workspace and an extensible plug-in system to add new features to the system. In our study, we employed two versions of Eclipse software system, i.e., version 2.0 and version 3.5. JEdit[6] is another open source, Java language based software system, which is a text editor that runs in any operating system with Java support, including Windows, Linux, Mac OSX and BSD.

As shown in Table 3-1, our data set consists of 114 bug reports and 34,375 source classes in total. Eclipse 3.5 system is the largest source file collection out of all the software

[2] http://adempiere.org/site/
[3] http://www.artofillusion.org/
[4] http://www.atunes.org/
[5] http://www.eclipse.org/
[6] http://www.jedit.org/

systems and it contains over 20,000 Java classes. The largest bug reports count was 40 reports, which also belong to the same Eclipse version. Out of all the source classes, 166 of them were patched while fixing those bugs.

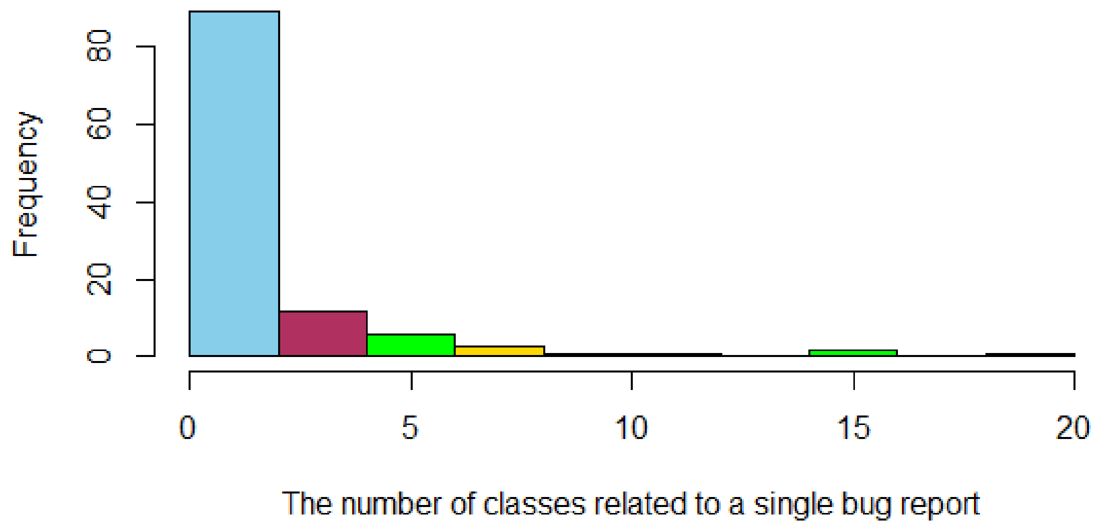


Figure 3-1: The count of the related patch classes for the bug reports

Most of the bug reports in the data collection (88.77%), were related to one or two patched classes (Figure 3-1). The summary of the data set we used is available in Appendix A and C.

3.3. Automating Text Retrieval Process

We followed the text retrieval process described in Chapter 2 to extract vocabularies from bug reports and source code. As we explained in Section 3.2, the data set we collected from different software systems contains 114 bug reports and over 30,000 source files including patched files. It takes considerably long period of time to create the vocabularies for each of these files manually. Hence, we built a software tool that automates the steps of text retrieval process and builds the vocabularies. In other words, this tool creates two separate corpora for the bug reports and the source files, then, applies text pre-processing

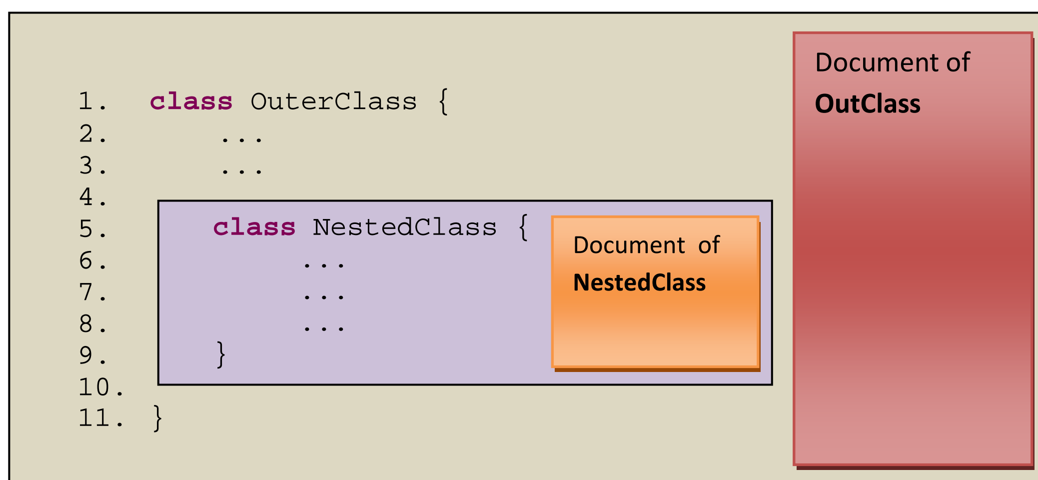
techniques on the documents belong to each corpus and creates vocabularies from these documents. The following sections elaborate how the tool we built performed during each text retrieval activity.

3.4. Automated Corpus Creation

As the first task of text retrieval process, we create the corpora for the source files and the bug reports by extracting desired text from them. In our study, we consider the bug reports as queries and the source code as documents that were matched with queries. The source files belong to each software system were the initial input to our tool to create the source code corpus while the bug reports' corpus was built using the content of each bug report extracted before.

3.4.1. Determination of the Granularity

We define the granularity of the source code document as class level. Hence, class or interface definition of each source file was extracted by the tool and this block of code was used as the input document for further process. In future reference, we use the term "class" to represent both class and interface definitions of a source file.



Code Segment 3-1: Nested Classes

If the source file contained only a single class, only one document is created. In case of having multiple classes in a single source file, separate documents are created for each and every class definition. If nested classes exist in a source file, then for each class, an individual document is created. In that case inner classes are considered as a part of the outer class, i.e., the content of the nested class included in the document created from the outer class. This idea is illustrated in Code Segment 3-1.

```
1.  /*
2.  * Data Handler
3.  * Created on 03-21-2013
4.  */
5.  package dataview;
6.
7.  import java.sql.Connection;
8.  import java.sql.DriverManager;
9.  //import java.sql.ResultSet;
10.
11.  /**
12.  * Class that utilizes database access.
13.  */
14.  class DataAccess {
15.       …
16.       …
17.       …
18.
19.       @Override
20.       public void Create() {
21.           …
22.           …
23.           …
24.       }
25.
26.       …
27.       …
28.       …
29.  }
```

Code Segment 3-2: Eliminations of content in source file - Package declaration, import statements, comments and annotations

According to the granularity we defined for the source files, all the content resides out of the class definition, were eliminated from the source code documents. For instance, text contents of lines 5, 7, and 8 in Code Segment 3-2 were turned down in this elimination process. The required granularity of the source files was the class definition. Hence, the tool filtered out the rest of the code from each source file except the class.

Hence, the elements that reside out of that scope, such as, Java package declaration and import statements were ignored when creating the source code documents. During this filtering procedure, the documentation (javadocs) that describes the class definition is concerned as a part of the document. But the rest of the comments (e.g. line comments, block comments and javadocs) lie outside the class definition was eliminated. This situation can be depicted by using Code Segment 3-2. The block comment spreads over line1 to 4 and the line comment resides in line 9 are ignored as they do not belong to the class definition. However, the documentation (javadoc) written in line 11 to 13 are left in the document as it belong to the class definition, even it does not reside in the class body. We decide whether a certain comment belongs to the class definition, if it lies just above the class declaration statement such that no blank lines exist in between the comment and the class declaration statement. Code Segment 3-2 illustrates such a valid block comment (from line 11 to 13) that belongs to the class definition.

In our study, the bug title and the description together are defined as the granularity of the bug report. Our tool read these bug titles and description from the repository that they were stored. Therefore, bug titles and the descriptions do not require any nose reduction as they were already filtered out before. We formulate queries using these bug reports.

3.4.2. Noise Reduction

This step eliminates noise that can be found in the documents and queries. Noise can produce misleading outcome due to its irrelevancy to our study purpose. As we focus on the content of the source class definition, we check for noise found inside the class body. As we convert source classes to SrcML [24] document when creating the vocabularies (Section 3.6) considering the locations of the words. As SrcML [24] tool has an issue regarding the conversion of class and method annotations, it causes inaccurate results if we don't remove those annotations from the class. Therefore, we considered class and method annotations as an unwanted content. Even if they lie inside a class, For this reason, annotations were removed from the source documents in our corpus. For example, the annotation "@Override" (line 19) in Code Segment 3-2, is removed from the code segment.

3.5. Automated Text Pre-processing

Text pre-processing or corpus normalization is inevitable in this study as the words extracted from the documents at this stage, are the fundamental units used to create vocabularies for further analysis. We discuss about the steps involved in text pre-processing in Section 2.2. Following subsections describe how each step in the text pre-processing is applied on our data set to obtain the desired outcome.

3.5.1. Automated Tokenization and Compound Terms Splitting

As we considered Java language based source files, identifier names do not consist of certain characters like white space, brackets, punctuation etc, but they may contain numbers. Hence, our tool tokenizes identifier names by numbers and delimiters (hyphen and underscore). However, code artifacts, such as, comments and string literals can contain white space, line breaks, punctuation and brackets, in addition to delimiters. So those characters are used to tokenize such elements. At the same time, the source code may

contain compound terms due to the various naming conventions as we discussed in Section 2.2.1. These compound terms are split by the uppercase letter during this process.

Bug reports may contain some terms appeared in the source code, such as, identifier names when the bug is described using the words in the error messages and the exceptions. In that case, it is essential to apply the same pre-processing techniques on bug documents since they also contain compound words and terms to be tokenized. In our study, we considered words that are made up of only alphabetical characters. Hence, any string which consists of numeric characters were split by those numeric values. For example, an identifier name “title2desc4” is split into two words “title” and “desc”, by numeric characters “2” and “4”. This rule is applied on both bug reports and source files similar to the other compound terms splitting rules.

3.5.2. Automated Stop Word Removal

As we explained in Section 2.2.2, stop words are filtered out from text documents since they are not useful and relevant to the text retrieval process. .

```
1.     …
2.     …
3.   public String GetGrade(int score) {
4.
5.       if (score >= 60) {
6.          return “pass”;
7.       }
8.       else {
9.          return “fail”;
10.      }
11.      …
12.      …
13.      …
14.  }
```

[1]
~~Return~~ grade ~~for~~ invalid score

Description:

…
~~However~~ ~~if~~ score exceeds 100, ~~it~~ ~~still~~ ~~returns~~ PASS ~~and~~ ~~does~~ ~~not~~ indicate ~~any~~ error message. ~~The~~ score ~~should~~ ~~be~~ validated ~~before~~…
…
…
…

Code Segment 3-3: Stop words (strikethrough) shared between code snippet (Left) and bug report (Right)

We created a list of stop words to be removed from both bug and source code documents. This list contained the most frequently used English words, as well as the keywords used in programming languages (Appendix B). Our automated text retrieval procedure compares each word picked from the document, with its stop word list and filters out unnecessary words. The set of strikethrough words in Code Segment 3-3 is an example for stop words.

Sometimes the keywords found in the source files, can be found in the bug reports with or without the same meaning. Code Segment 3-3 illustrates this situation by comparing a code snippet and a bug report. The words “if” and “return” appear in both code snippet and bug report. In the source code, these words are considered as keywords. In contrast to that, in bug report “if” acts as a subordinating conjunction used in English grammar while “return” is applied as a verb. Yet both words are eliminated from the code document and the bug document because, we apply the same stop removal method on both types of documents.

In addition to the removal of the frequently used English words and programming keywords, we filter out words that are made up of single characters. After tokenization and compound words splitting, single characters may remaine as words. For instance, let’s consider an identifier name “cellA1_s”. As a result of tokenization, we obtain two terms “cellA1” and “s”. Then we split the compound term “cellA1” and it results two words “cell” and “A”. At the end of the whole tokenization and compound word splitting process, we have three words “cell”, “s” and “A”. Since the last two words are only single characters, they do not provide any valuable contribution to our study. Therefore, such words are considered as noise.

3.5.3. Automated Stemming

Stemming is an activity that reduces family of words to their root, by cutting off the derived and inflected words. As we discussed in Section 2.2.3, there are variety of different algorithms have been implemented to handle the stemming programmatically. In our study, we followed Porter Stemmer[7], a suffix removal stemming algorithm that reduces the same lexeme to the stem. This algorithm chops off the end characters of a string to map it into the root form. For example, all the words "divide", "dividing", "divided" are mapped to "divid" after applying Porter Stemmer.

3.6. Creating Vocabularies

Vocabularies are formed by words in the documents and the frequencies of their occurrences. To answer all the thesis questions RQ1, RQ2 and RQ3, it is essential to create vocabularies from the source files and the bug reports with and without considering the location. The location means a particular artifact in the document. The source code and the bug reports carry different types of locations. For example, the words of a source code document can be extracted from variables, comments, literals etc. In our study, the words come from bug reports, belong to either bug title or bug description. Figure 3-2 presents the types of vocabularies we created for our study. We call the vocabularies that keep location information, as *annotated vocabularies* and the vocabularies that do not carry location information as *not-annotated vocabularies*.

[7] http://tartarus.org/~martin/PorterStemmer/

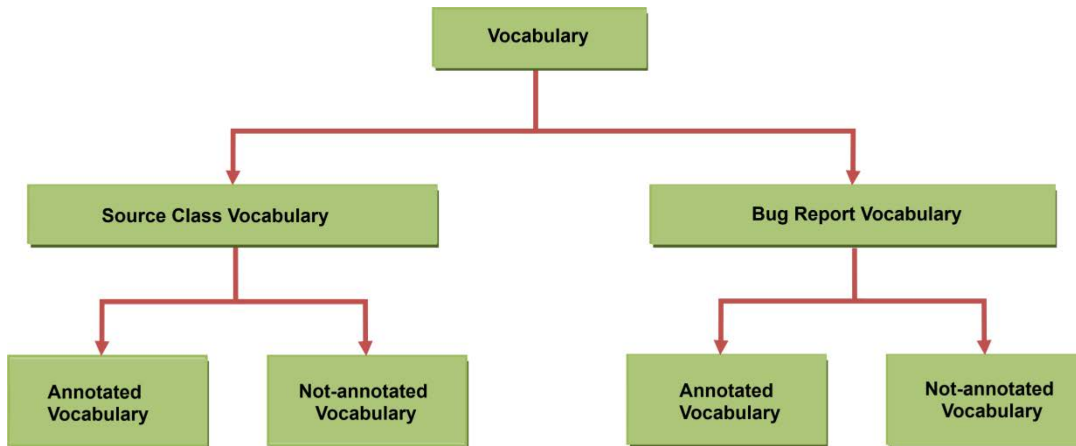


Figure 3-2: Vocabularies based on location information

3.6.1. Annotated Terms Vocabularies

Annotated terms vocabulary is a mapping of each word to its location and the frequency of each location. The elements of annotated vocabulary are listed as follows.

- Words found in the document (source code or bug report)
- The location of the word
- The frequency of the words occurring in each location

The locations of the words found in the source files are determined based on the SrcML [24] toolkit by executing it as a part of our tool. After creating the corpus for the source files, we converted source documents to SrcML documents which are comprised of the elements derived from the programming language. The content of SrcML documents is encoded on XML (Appendix D). As the elements of SrcML files correspond to the various artifacts found in the source code, such as, classes, methods and comments, we determine the code locations of each word in these XML files according to the predefined list of source code locations (Table 3-2).

Table 3-2: Source code locations

Location	Artifacts in Source Code
Argument	Argument name
Attribute	Attribute name
Class Name	Interface or a class name of the source code document
Comment	Line comment or block comment or docs (e.g. javadoc)
Literal	String literals
Method Call	Method call
Method Name	Method Name
Parameter	Method parameter name
Type	Class or interface names found inside the class definition we considered as source code document
Variable	Local variable name

As shown in Table 3-2, we consider ten code locations which are meaningful for further processing of our study. The first column of this table indicates the name of the code location and the second column describes the corresponding artifact.

```
1.  private static double  GetResult(int a, int b) {
2.
3.     double denominator = (a + b) * 0.2;
4.
5.     if (denominator != 0) {
6.           return a / Math.abs(denominator);
7.     }
8.     else {
9.          System.out.println("No Result. Divide by zero.");
10.     }
11.     return -1;
12. }
```

Code Segment 3-4: A simple Java method

Table 3-3: Annotated terms vocabulary

Word	Location	Frequency
ab	Method Call	1
denomin	Argument	1
	Variable	2
divid	Literal	1
get	Method Name	1
math	Type	1
print	Method Call	1
result	Literal	1
	Method Name	1
zero	Literal	1

Table 3-3 lists the mapping of the words obtained from Code Segment 3-4, to the code locations and frequencies. These words are extracted after applying text pre-processing techniques described in Section 3.5. For instance, *variable* is one of the locations that the word "denominator" occurring in the Code Segment 3-4. The frequency of the word "denominator" in location *variable* is two as it appeared twice as a variable in lines 3 and line 5. In addition to that, this word occurred once as an *argument* in Line 6. Hence, the frequency of the occurrence of the word "denominator" in the location *variable* is two while its occurrence in the location a*rgument* is one. Table 3-3 shows the annotated vocabulary created for code snippet in Code Segment 3-4.

Compared to the source code vocabularies, the bug vocabularies consist of two locations, *title* and *description*. For example, consider the word "concave" occurring in both *title* and *description* as shown in Table 3-4. It appears once in the *title* and once in the *description*. So the frequencies of the occurrence of word "concave" in the bug *title* and *description* is one.

Table 3-4: Bug title and description extracted from Art of Illusion bug #34[8]

Location	Content
Title	Closing **concave** boundary
Description	When closing a boundary in the triangle mesh editor, often impossible geometry is created. This happens when the boundary is planar, but **concave**. The algorithm used for converting closed curves to triangle meshes does not suffer from this problem. Why not use this algorithm to close boundaries too?

3.6.2. Not-annotated Vocabularies

Not-annotated vocabularies map each distinct word found in a document (source code or bug report) with its frequency. Table 3-5 shows the not-annotated vocabulary created for Code Segment 3-4. Compared to annotated vocabulary (Table 3-3), not-annotated vocabulary sums up all the location frequencies of each word. e.g. the frequency of the word “denomin” in not-annotated vocabulary is 3, which is divided between two locations (*argument* and *variable*) in annotated vocabulary (Table 3-3).

Table 3-5: Not-annotated vocabulary

Word	Frequency
ab	1
denomin	3
divid	1
get	1
math	1
print	1
result	2
zero	1

[8] http://sourceforge.net/p/aoi/bugs/34/

The Figure 3-3 summarizes the automated process of creating vocabularies. When creating the not-annotated vocabularies, the output of the corpus creation step was used as the direct input for text pre-processing. In contrast to that, when creating the annotated vocabulary, an additional step was taken place in between corpus creation and text pre-processing, which is the conversion of documents (source classes) to SrcML documents.

For future reference, we define the document size or query size as the total number of words occurring in the document. In that case, each word is counted even if it is occurred repeatedly in the document. The size of the vocabulary is determined by the total number of unique words appeared in the document. In this case, the repeated occurrences are not counted. For instance, the document size for Code Segment 3-4 is eleven and its vocabulary size is eight.

dys

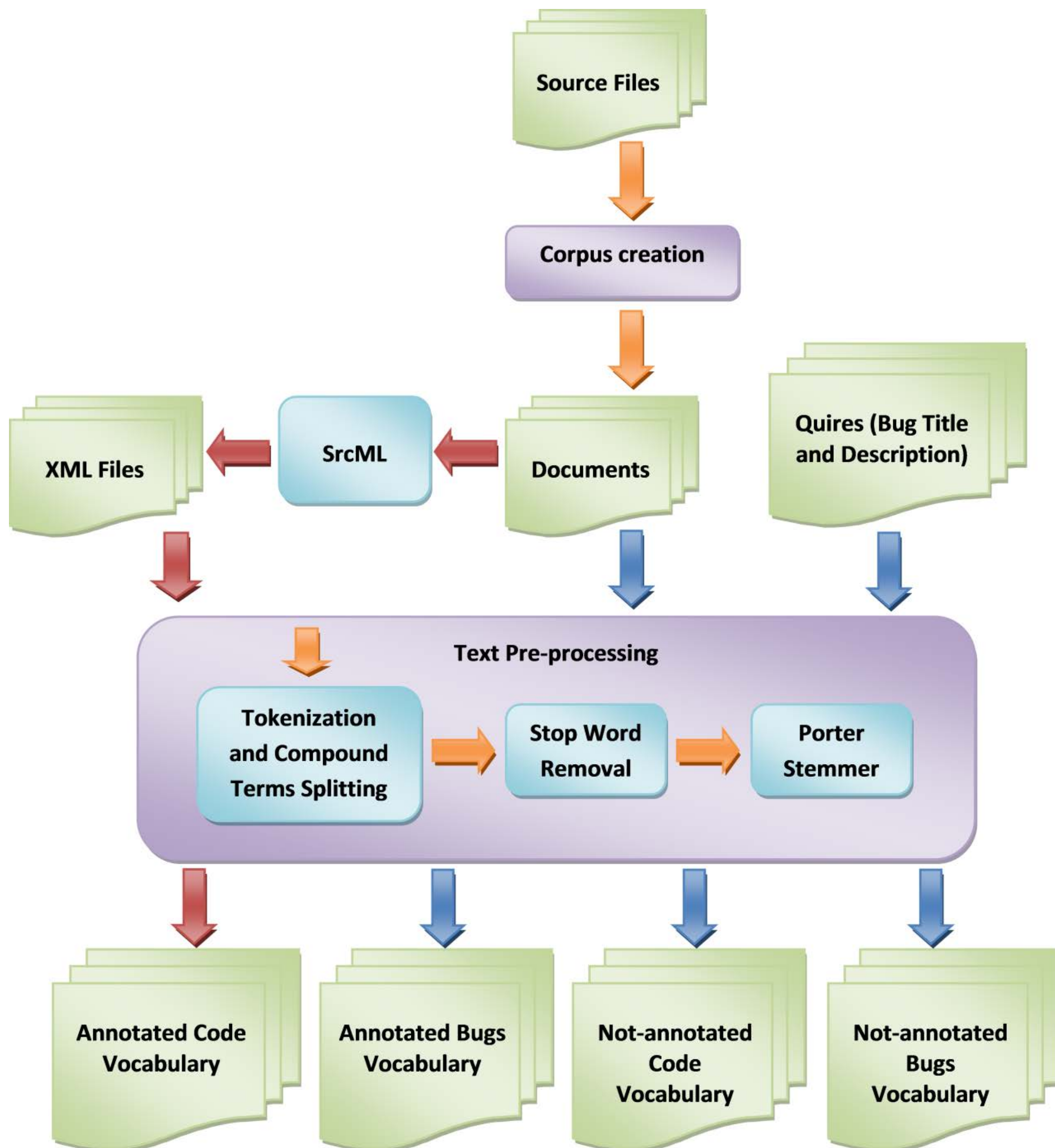


Figure 3-3: The automated process of creating locater terms and not-annotated vocabularies

3.7. Planning of Research Questions

We obtain vocabularies for each source class and bug report of all the software systems. This subsection describes how we address the thesis questions using these vocabularies.

Table 3-6: Shared words (without locations) between source class (left) and bug report (right) given in Code Segment 3-3

Word	Frequency
fail	1
get	1
grade	1
pass	1
score	2
string	1

Word	Frequency
error	1
exce	1
grade	1
indicate	1
invalid	1
messag	1
pass	1
score	1
	2
valid	1

Planning RQ1:

Question: To what extent are the vocabularies of bug reports reflected in the identifiers and comments of classes?

To answer this question, we selected both bug vocabularies and source class vocabularies belong to each software system. In this case, we did not consider the location of words. We extracted the shared words between each possible pair of bug vocabulary and source class vocabulary of a specific system. This collection of shared words is an intersection between the two vocabularies, which is called the common or shared

vocabulary of the bug report and the source class. For example, Table 3-6 illustrates the vocabularies of the source class and the bug report presented in Code Segment 3-3. These two vocabularies share two words "grade" and "score" which is the common vocabulary of the document (source class) and the query (bug report). Then we compute the size of this common vocabulary (i.e., the number of unique words) and the frequency of the occurrence of each word in the bug report and the source class. For instance, the common vocabulary for the vocabularies mentioned in Table 3-6 is illustrated in Table 3-7.

Table 3-7: The common vocabulary (the intersection between source class and bug report illustrated in Table 3-5)

Word	The frequency of the occurrence in source class	The frequency of the occurrence in bug report
grade	1	1
score	2	3

For the same pair of bug vocabulary and the source class vocabulary, we measured the Simpson Similarity Index, given by the number of shared words between the bug reports and the source class divided by the minimum size of their vocabularies. The analysis of these measures and the results of this analysis are discussed in Section 4.2.1.

Planning RQ2:

Question: What is the code location (i.e., class name, method name, attribute name, etc.) of the shared words between bug reports and patched classes?

In order to address this question, we obtained *annotated vocabularies* for patched classes and compute the size of each location we considered (Table 3-2). For example, the size of the location *method name* in the source class vocabulary given in Table 3-8 is two. For the same vocabulary, the size of the *class* location is one. We extracted the set of shared words between each source class location and the bug report location, and the

frequency of the words in the location. We analyzed these vocabularies to answer RQ2. This analysis and the results obtained are discussed on Section 4.2.2.

Table 3-8: Shared words (with locations) between source class (left) and bug report (right) given in Code Segment 3-3

Word	Location	Frequency
fail	Literal	1
get	Method Name	1
grade	Method Name	1
pass	Literal	1
score	Parameter	2
string	Class	1

Word	Location	Frequency
error	Description	1
exce	Description	1
grade	Title	1
indicate	Description	1
invalid	Title	1
messag	Description	1
pass	Description	1
score	Title	1
	Description	2
valid	Description	1

Planning RQ3:

Question: Is the number of shared words between bug reports and classes an adequate measure to support bug localization?

To address RQ3, we consider the size of the common vocabulary between the bug report and the source class that belong to each system. Also we compare the similarities between the source classes and the bug reports of the same system (these similarities are listed at the end of this section) with two text retrieval approaches, namely the Latent Semantic Indexing (LSI) and the Lucene implementation of the Vector Space Model. Both these approaches are widely used for bug localization. Since the bug reports are considered as queries, we retrieve the source classes, seeking for an existence of the relevancy

between the source code and the bug report. To apply these approaches, first, we created document-term matrices for both source classes and bug reports.

For each query, i.e., the bug title and the description together, we rank the classes of each system based on three measures listed as follows.

(1) The number of shared terms between the class and the bug report, i.e., size of the common vocabulary between the source class and the bug reports
(2) The cosine similarity between the class and the query using LSI
(3) The cosine similarity between the class and the query using Lucene

The documents (source classes) are ranked according to the similarity measures from the highest to lowest that received using both LSI and Lucene implementation of VSM. Higher similarity measure of the document indicates that it is more relevant to the query. We compare the effectiveness of each technique, i.e., the rank of the first patched class in the list. Next chapter explains the results and findings we obtained by following the procedure mentioned above, the analysis and the results of RQ3 are presented in Section 4.2.3..

CHAPTER 4 ANALYSIS OF THE RESULTS AND DISCUSSION

In this chapter, we analyze the results that we obtained by following the approaches mentioned in Chapter 3 and discuss about the observations that we can made.

4.1. Observations about the Document Size

4.1.1. Bug Reports

According to the analysis of bug reports' size, most of the reports, in terms of percentage, 75% of them had 64 terms or less (Figure E.1, Table E.1 in Appendix). Compared to the other systems, Eclipse 3.5 had the largest bug report in the corpus which contained 250 terms (Figure E.2, Table E.1 in Appendix). The shortest report consisted of 8 terms, which belonged to ADempiere 3.1.0 (Figure E.2, Table E.1 in Appendix). The average size of a bug report was 56 terms (Table E.1 in Appendix). Approximately, in average, two third of the terms (65.86%) in each bug report was unique (Table E.2 in Appendix). The largest bug report that belonged to Eclipse 3.5, also contained the largest set of unique terms, consisting of 114 terms ((Figure E.2, Table E.2 in Appendix). The bug report with the smallest set of unique terms belonged to aTunes 1.10, which contains of 5 terms ((Figure E.2, Table E.2 in Appendix).

To verify the correlation between the size of the bug report and the size of its vocabulary, we calculated the Spearman coefficient between those two variables. The output showed a strong monotonic relationship between the bug document size and its vocabulary size ($r = 0.96$ and p-value < 0.01), which indicate that the number of unique terms increases with the bug report size. In other words, the larger the bug report, the more unique terms it contains.

4.1.2. Source Classes

Compared to the bug reports, the code vocabularies contained a larger number of terms such that 75% of the classes had more than 82 terms. Average size of a class was 464 terms which is approximately 85% more terms than the largest bug report in the corpus. Similar to the result obtained for the bug reports, the largest class in the collection belonged to Eclipse 3.5 system. There were more than one shortest class documents in the collection such that each consisted of one term. These documents belonged to Art of Illusion 2.4.1, Eclipse 2.0, and Eclipse 3.5. Furthermore, in average, less than 27.1% of the terms in classes were unique. Similar to the bug reports, we found the correlation between the number of unique terms and the document size of the source classes by calculating the Spearman correlation. The result showed a very strong correlation between these two variables ($r = 0.95$, p-value < 0.01) which indicate that larger the class, more unique terms it contains.

4.2. Addressing Research Questions

4.2.1. Research Question 1

We computed the set of common vocabularies between source classes and bug reports for 1,077,074 pairs, considering every possible combination of these documents per each software system. We obtained the common vocabularies for all the pairs of documents and queries of each system, regardless of their relevance to each other. In other words, this set of common vocabularies contains both relevant and non relevant pairs. The result obtained by analyzing these vocabularies, indicated that 75% of the pairs (i.e., 808,928) shared between 1 and 13 terms. Yet 21.68% of the pairs did not have any shared terms while 3.22% (i.e., 34,646) pairs contained more than 10 shared terms (Table 4-1). *This*

result directs to the conclusion that bug reports share terms with a large number of classes belong to a software system which is almost 80% in our study.

Table 4-1: Categorization of the percentage of pairs (common vocabularies) by the number of shared terms

System	Shared Pairs of <Bug Report, Source Class>		
	No Terms	1 <= Number of Terms <= 13	Number of Terms > 13
ADempiere 3.1.0	17.82%	81.89%	0.29%
Art of Illusion 2.4.1	19.39%	78.91%	1.7%
aTunes 1.10	31.5%	67.94%	0.56%
Eclipse 2.0	18.83%	79.09%	2.09%
Eclipse 3.5	22.03%	74.47%	3.5%
JEdit 4.2	22.92%	75.76%	1.32%
Total	21.68%	75.10%	3.22%

The above analysis was performed to determine the relevancy of a source class to a bug report without considering whether they are related to each other or not. We followed the same procedure to check whether the same result could be obtained by analyzing the bug reports and their corresponding source classes (patched classes). Therefore, we analyzed all the pairs of bug reports and their patched classes which was a subset of the previous collection of pairs. This subset is referred as the *patched subset* while its complement is called as the *non patched subset*. The patched subset consisted of 166 related elements. This analysis indicates that the 99.6% pairs, in other words, 165 out of 166 total pairs, shared some common terms in the patched subset. Similarly, 78.32% of pairs belong to non-patched subset, shared a non-empty set of terms. The only pair of documents in the patched subset that didn't share any term belonged to aTunes 1.10 system and its patched class contained 54 unique terms while the bug report consisted of only 5 unique terms. Figure 4-1 illustrates the number of unique shared terms (common

vocabulary size) between bug reports and source classes (patched and non patched classes) belong to each system by omitting the outliers.

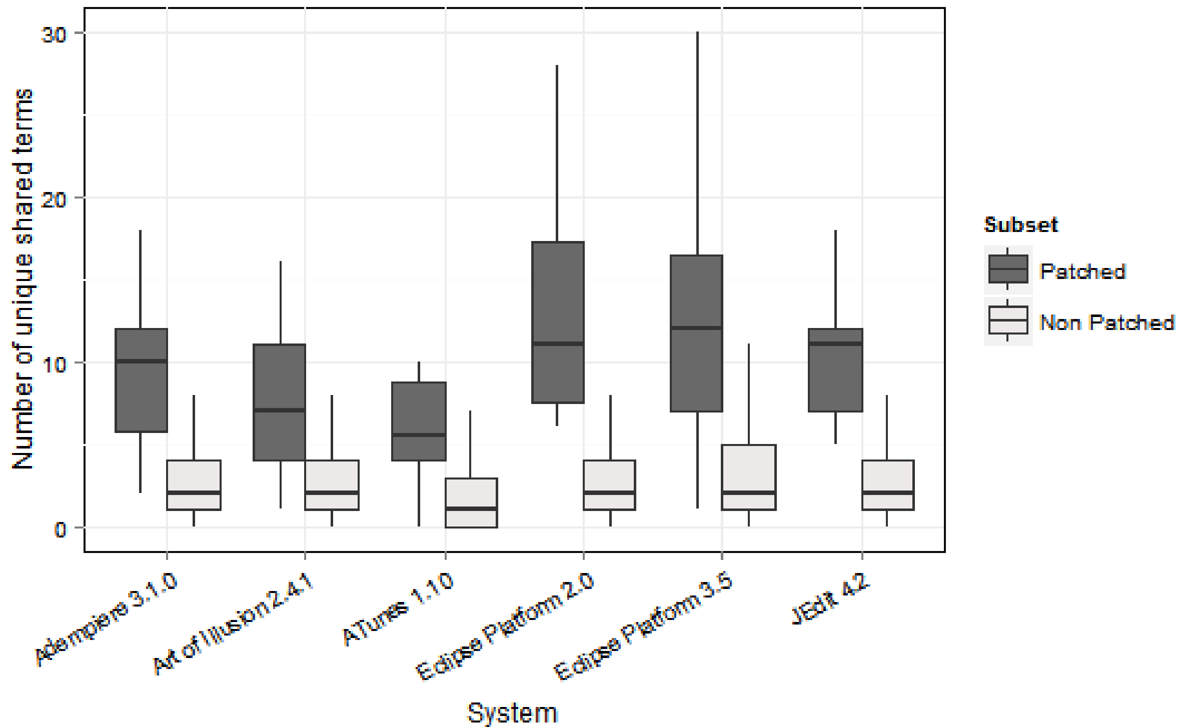


Figure 4-1: Common vocabulary size of bug reports and source classes in each system without outliers (Appendix A)

The results we obtained from Mann-Whitney test [25] showed the difference of shared terms in these patched and non patched subsets is statistically significant for all the software systems (p-value < 0.05). According to the results, the patched classes had 11.7 average shared terms with their relevant bug reports. Meanwhile the other classes (non patched) shared 3.39 terms in average with the bug reports. *Due to these observations, we conclude that bug reports share more terms with the patched classes than the non patched classes, in average.*

To find out any difference in the size between the patched and non patched classes, the **Spearman correlation** similarity index was computed. The Simpson similarity index is suitable for the situations where there exits any size difference between two documents as it recognizes the overlap with respect to the smaller document in size. According to Figure 4-2 (outliers have been ignored in the boxplots), for each system, the similarity between bug reports and patched classes had a higher significant level (p-value < 0.05) than the similarities computed between bug reports and non patched classes. We observed that in average, similarity value of non patched subset is 0.11 and for patched subset, it is 0.37.

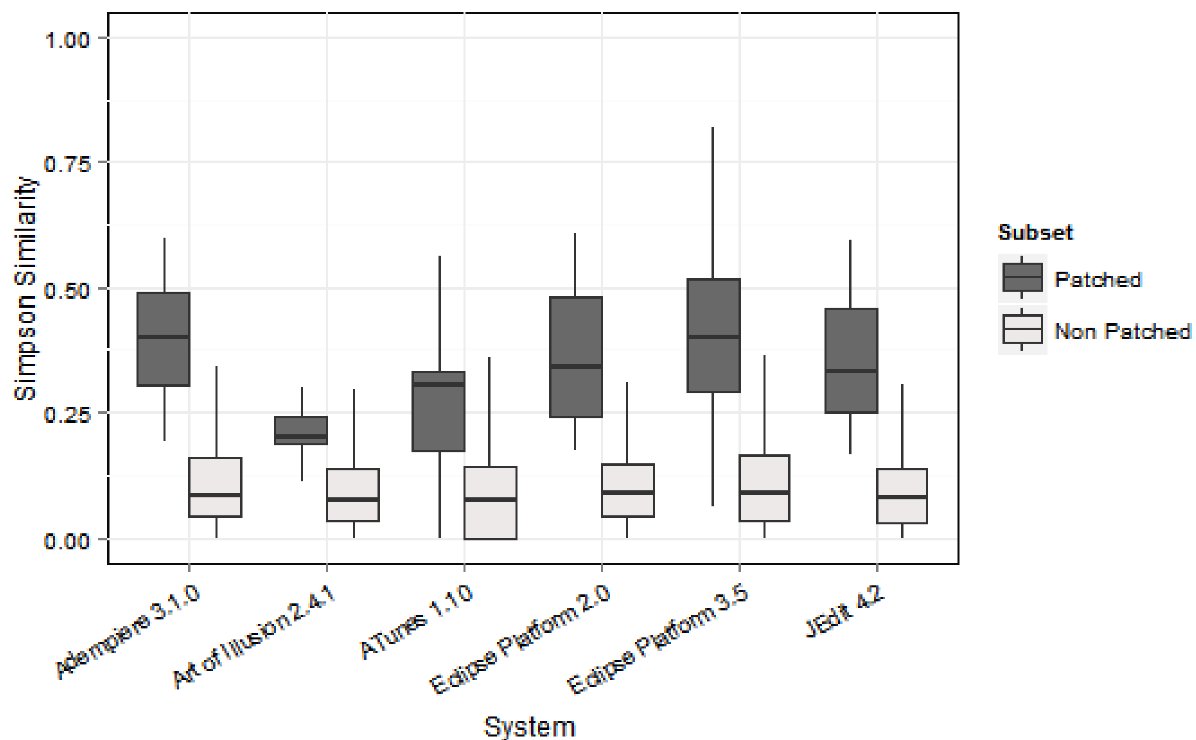


Figure 4-2: Simpson similarity index between bug reports and source classes of each software system

Furthermore, we found that 75% of the pairs belong to the patches subset resided above the Simpson similarity of 0.24 while 88.59% of the pairs in the non patched set lied below the same similarity value. This observation confirmed the previous results we

obtained. *Hence, we conclude that in average, bug reports have higher textual similarity with the patched classes than the non patched classes.*

RQ1 Answer:

We conclude that the bug reports share terms with a large number of source classes (almost 80%) and the most of these terms are shared with the patched classes than the non patched classes. We also found that the bug reports have higher textual similarity with the patched classes than the non patched classes.

4.2.2. Research Question 2

Recent research studies argue that the use of code location information for source code retrieval process, improves the bug localization [10]. Therefore, we analyzed the code locations of the terms shared between patched classes and relevant bug reports. Table 4-2 indicates the results we obtained by this inspection.

According to the results, it is apparent that the terms in the class vocabularies were not evenly distributed over the code locations, as the measurements of terms for different code locations disseminate with different values. Compared to the other code locations, the *comments* contributed the highest percentage of the terms for the patched class, which was 26.98% of the vocabulary size (Table 4-2). Terms found in *method calls* and *types* also carried the next highest values which are 20.28% and 12.69% respectively (Table 4-2). This observation was somewhat we expected, as c*omments, method calls* and *types* were more verbose when describing the defects in a bug report. Furthermore, we studied the contribution of each code location to the total shared terms of the class and we discovered that the majority of shared terms appeared in *comments* (27.33%, in average), *method calls* (16.99%, in average), and *types* (15.16%, in average) of the patched classes. The Spearman correlation showed a strong relationship between the number of terms and the

number of shared terms in all the code locations (r > 0.6, p-value < 0.01) except for the *class names*. *Due to this outcome, we state that the code locations which are more verbose tend to contribute more to the common vocabulary between the patched class and its relevant bug report.*

Table 4-2: Shared terms statistics in code locations in the patched classes

Code Location	Average Percentage of Terms	Average Percentage of Shared Terms	Average Contribution to the Total Shared Terms
Argument	11.71%	19.06%	11.34%
Attribute	8.40%	20.10%	8.45%
Comment	26.98%	19.86%	27.33%
Literal	2.99%	21.40%	4.64%
Method call	20.28%	15.78%	16.99%
Method name	4.40%	21.52%	4.58%
Parameter	4.25%	19.97%	3.81%
Type	12.69%	20.69%	12.97%
Class Name	0.55%	53.57%	1.70%
Variable	7.86%	15.68%	7.11%

A short location size does not mean it is contribution to the common vocabulary, is also small. For instance, a set of shared terms belong to a certain location may be smaller in the size, compared to the number of shared terms of the other locations. But in reality, the contribution of that smaller set of words to the shared vocabulary can be larger compared to its size as the majority of the words belong to that code location is a part of the common vocabulary. Because of that, we computed the number of common terms in each code location with respect to its size (i.e., the percentage of shared terms for each location) as listed in Table 4-2. According to the results we obtained, *class names* shared quite the highest percentage of the terms which is 53.57%. The other code locations contributed the percentage of the shared terms between 15% and 21% with their bug reports. *We conclude*

that the names of the patched classes are more likely to have shared terms with relevant bug reports than the terms belong to the other code locations.

RQ2 Answer:

We conclude that the verbose code locations tend to share more words between a patched class and a corresponding bug report. We also state that the *class names* are more likely to have shared terms with the relevant bug reports than the other code locations.

4.2.3. Research Question 3

To find out whether the number of shared terms between bug reports and classes provide an adequate measure to support the bug localization process, we developed a simple technique. In this technique, the number of shared terms (ST) was used as a measure for locating the source classes relevant to the bug reports. A corpus for each system was built as described in section 3.4. Then we indexed each corpus with Lucene and Latent Semantic Indexing (LSI) (d = 100 for LSI). For each query (bug report) of each system, the documents (source classes) were ranked using ST, LSI and Lucene respectively. Then the top ranked documents were compared for effectiveness. Table 4-3 reports the number of cases that the shared terms perform better, equal or worse than Lucene and LSI. The average and median effectiveness for the above ST, LSI and Lucene approaches are listed in Table 4-4. In this technique, a lower effective measure indicated better results.

Table 4-3: Number of cases that the shared terms approach is better, equal or worse than LSI and Lucene

System	LSI			Lucene		
	Better	Equal	Worse	Better	Equal	Worse
ADempiere 3.1.0	9 (56%)	0 (0%)	7 (44%)	1 (6%)	1 (6%)	14 (88%)
Art of Illusion 2.4.1	4 (40%)	0 (0%)	6 (60%)	0 (0%)	0 (0%)	10 (100%)
aTunes 1.10	9 (53%)	0 (0%)	8 (47%)	1 (6%)	1 (6%)	15 (88%)
Eclipse 2.0	11 (85%)	0 (0%)	2 (15%)	2 (15%)	1 (8%)	10 (77%)
Eclipse 3.5	24 (60%)	0 (0%)	16 (40%)	9 (22%)	3 (8%)	28 (70%)
JEdit 4.2	15 (83%)	0 (0%)	3 (17%)	5 (28%)	1 (6%)	12 (66%)
Total	72 (63%)	0 (0%)	42 (37%)	18 (16%)	7 (6%)	89 (78%)

Table 4-4: Average and median effectiveness of the shared terms (ST), LSI and Lucene approaches

System	ST		LSI		Lucene	
	Average	Median	Average	Median	Average	Median
ADempiere 3.1.0	41	19	124	23	11	4
Art of Illusion 2.4.1	87	35	84	30	52	10
aTunes 1.10	26	10	31	18	9	3
Eclipse 2.0	180	80	1152	681	49	3
Eclipse 3.5	430	66	915	120	594	5
JEdit 4.2	22	7	64	57	11	3
Total	192	25	492	57	223	4

The performance of ST were unanticipated as it worked much better than LSI in 63% of the cases with a better median and average effectiveness considering the data collected from all the software systems. Art of Illusion 2.4.1 was the only system where LSI showed a slightly better performance. But still the difference in median effectiveness (5 positions) is small compared to the improvements ST brings in the case of the other systems, e.g. Eclipse 3.5 indicates a difference of 601 positions. By conducting the Wilcoxon test, we observed a significant performance of ST compared to LSI (p-value < 0.05). Meanwhile Lucene performed better than both ST and LSI. The Wilcoxon test on the effectiveness

values indicated again that the difference in performance of the three approaches (i.e ST, LSI and Lucene) was statistically significant (p-value <0.05). After all, there were situations that ST gave better results than Lucene. JEdit 4.2 (28% of the cases), Eclipse 3.5 (22% of the cases), and Eclipse 2.0, (22% of the cases) were few of the examples that shows better performances of ST than Lucene. For some of the bugs, the difference in effectiveness was striking. For instance, the case of bugs 304784 and 29950 in Eclipse 3.5, the improvement over Lucene was of 7549 and 4960 positions, respectively. These outliers in Eclipse 3.5 described the better average effectiveness obtained by ST for this system, compared to Lucene. To explain these results, we need to conduct further studies. *Based on the above results, we conclude that the bug localization is supported by the number of shared terms between bug reports and source classes better than LSI does, still it does not perform as well as Lucene.*

RQ3 Answer:

We conclude that the number of shared words between the bug reports and the source classes supports the bug localization better than some other measures such as LSI, but still does not perform as well as Lucene.

4.2.4. Threats to Validity

This section discusses the threats to the validity of our study , organized by the threats category [17, 26].

Construct validity focuses on the relation between experiment and observation. The selection of appropriate measures and algorithms influences the accuracy of the whole study and the conclusions. In our case, we adapted effectiveness measure and statistical tests, such as, Mann-Whitney test, to evaluate results obtained regarding the bug

localization. Due to the good estimation provided by these measure and tests, they are widely used in the concept location tasks.

Internal validity considers the variables that can influence the outcome and whether there are sufficient evidences to support the conclusions. In this study, we concerned about the reliability of the documents and queries we used, to mitigate the threats that can affect to our results. We extracted the queries (bug titles and descriptions) from publicly available bug tracking systems. We followed a well-defined procedure to extract vocabularies from the source files and the bug reports.

External validity concerns the generalization of the results we obtained. We conducted our study on a collection of documents belong to six software systems which contained 166 patched classes along with 114 bug reports. As these inputs are relatively small, it is important to validate our work to identify the threats that can affect our conclusions. Therefore, this study needs to be replicated on comparatively larger data sets to see whether the same conclusions can be drawn. In other words, we need to expand this study for a larger number of bug reports and patched classes to see whether the same results can be obtained for the new input.

Conclusion validity concerns how accurate our approach with actual results we obtained. We drew our conclusions referring to the intersection between the bug vocabularies and the source vocabularies. Our conclusions provided the evidence to support the assumption that the bug reports and the relevant source classes share a common vocabulary. We followed Mann-Whitney and Wilcoxon statistical tests to show the significance of the results we obtained.

CHAPTER 5 RELATED WORK

It is importance to understand how the problems related to defect source code are described in bug reports, to improve the bug localization. Due to this importance, several studies related to the linguistic and statistical analysis, have been conducted to describe the text content found in bug reports.

Title of a bug report is one of the important elements that used to describe a bug. Several interesting trends about the bug title, have been discovered by Ko et al. [27] after analyzing the bug title's parts of speech of the words. The results of this study indicated which parts of the speech of the words are supportive when describing the bug reports. Another similar linguistic analysis of bug titles is carried out by Sureka and Indukuri et al. [28] to examine how people describe software bugs. This study aimed to identify the feasibility of building a predictive model to categorize the bug reports, based on the part of the speech, word frequency and the distribution across different severity levels, such as, bug importance (*major, minor, enhancement, critical*). This study revealed that bug titles, in general, do not carry enough information to build a highly accurate classifier which supports categorization of bug titles into different predefined severity levels. But it showed that some certain categories, such as, *enhancement* and *critical* determine the words which can be used to build a model with a reasonable accuracy. Han et al. [29] performed a qualitative and quantitative topic analysis on bug reports of Android systems to obtain evidence of Android fragmentation (hardware and software) within these bug reports. The study applied Latent Dirichlet Allocation (LDA) on original bug reports and Labeled LDA on manually labeled bug reports using feature oriented terms. Then it computed the average relevance between each individual bug report to each topic and analyzed the performance of two types of topics sets. According to the results, the study concluded that bug reports carry important

evidence regarding certain type of fragmentations. In addition to the above studies, some research workers paid attention on the development of automated methods to measurie the quality of bug reports. The research study followed by Linstead et al. [30] focused on coherence which is a quality metric that measures the report quality by capturing the clarity of the reports through the analysis of text found in them. This study also adapted LDA for mining the text content of bug reports.

Even though our study also focused on text analysis, it adapted bug reports for analysis process along with source code. Our objective was to determine any relationship exists between vocabularies of bug reports and source classes. Recent work [10] conducted regarding bug localization, proposed a novel approach for term weighting on information collected from structural text (source code). This study proposed boosting the query terms that occur in certain locations within the source code methods. Using this work as the base for our study, we extended the analysis process from method to source classes. Furthermore, we took an extensive list of code locations that comprised of *class names, method names* and *comments*.

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

We conducted our study to find whether the bug reports and related defect source code share important terms. To accomplish our purpose, we analyzed the vocabularies shared between source classes (documents) and bug reports (queries). This chapter summarizes the conclusions we draw and discusses what extension can be made to this study to verify these conclusions and which kind of enhancements will be made in future.

6.1. Conclusions

According to the analysis of common vocabularies between bug reports and source classes, we observed that the bug reports shared terms with a large number of classes and more terms are shared between the bug reports and the patched classes than non patched classes. Hence, these evidences establish the implicit assumption that the source code shares many terms with bug reports. Furthermore, we can state that the patched classes carry a higher number of shared terms than the rest of the classes in a software system. Moreover, due to the results captured by studying code locations of the common vocabularies, we draw the conclusion that *class names* are more likely to appear in common vocabularies with bug reports than the other code locations. Nevertheless, more frequent code locations have more terms in common with the bug report. By means of the above conclusions, we believe that TR based concept location techniques can be improved by using this information. We obtained results that support the idea of using shared terms to determine the relevant classes for a selected bug report. Not only that, but also we saw some TR techniques (Lucene in our study) works better than others while some techniques, (LSI in our case) may perform worse.

6.2. Future Work

We conducted our study on a relatively small collection of patched classes and bug reports. as we discussed in Section 4.2.4, it is important to validate our work to identify the threats that influence our conclusions. Therefore, in future, we replicate this study on comparatively larger data sets with considerations of additional text retrieval techniques under various configurations.

On the other hand, all the software systems we employed were implemented using Java based technologies. In other words, they were written in Java language. But it is important to extend this work on software systems which are written in other programming languages, such as, C++, as different languages carry different syntax. In future, we will extend our study on software systems written in different programming languages, with the enhancements mentioned before.

APPENDIX A : Summary of the Data Set

Table A.1: The total number of unique terms in the source classes (class vocabulary size)

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	2,735	74	106.5	170.94	132	246.25	345	92.93
Art of Illusion 2.4.1	1,837	60	129	141.31	129	158	211	46.72
ATunes 1.10	2,439	14	77.25	110.86	105.5	136.5	218	54.95
Eclipse Platform 2.0	2,026	83	110	144.71	126	159.75	375	72.66
Eclipse Platform 3.5	13,180	20	108	178.11	146	214	538	113.63
JEdit 4.2	5,764	26	113	213.48	175	355	531	141.59
General	27,981	14	103	168.56	140.5	199	538	107.41

Table A.2: The total number of terms in the source classes (class document Size)

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	34,032	390	631.75	2127	1022	2751.75	6411	2207.85
Art of Illusion 2.4.1	13,902	268	894	1,069.38	945	1,325	1,887	511.05
ATunes 1.10	19,997	50	353.75	908.95	811	1,259.5	2,405	654.73
Eclipse Platform 2.0	16,229	285	842.75	1,159.21	1,029	1,616.75	2,667	630.85
Eclipse Platform 3.5	135,551	31	558.75	1,831.77	1,056	2,216	8,943	2,010.48
JEdit 4.2	62,382	54	632.5	2,310.44	1,109	3,923	15,098	3,039.36
General	282,093	31	558	1,699.36	1,012	1,975.75	15,098	2,004.49

Table A.3: The total number of unique terms in the bug reports (bug vocabulary size)

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	404	6	18.25	25.25	24	35	50	11.78
Art of Illusion 2.4.1	485	17	26	37.31	36	43	66	16.48
ATunes 1.10	543	5	13	24.68	18	29	58	16.79
Eclipse Platform 2.0	546	16	25	39	35	46.25	106	22.37
Eclipse Platform 3.5	2,742	8	17	37.05	32	49	114	23.94
JEdit 4.2	1,019	11	28.5	37.74	33	41	67	16.04
General	5,739	5	18	34.57	32	43	114	20.73

Table A.4: The total number of terms in the bug reports (bug document Size)

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	650	8	27	41	39	47	105	23
Art of Illusion 2.4.1	719	23	36	55	44	69	108	28
ATunes 1.10	798	9	17	36	24	41	101	28
Eclipse Platform 2.0	843	22	38	60	55	64	171	37
Eclipse Platform 3.5	5,282	10	25	71	51	111	250	56
JEdit 4.2	1,798	11	40	67	46	61	156	46
General	10,090	8	28	61	45	69	250	47

Table A.5: The total number of unique terms shared between the bug reports and the patched classes (common vocabulary size)

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	150	2	6	9	10	12	18	5
Art of Illusion 2.4.1	101	1	4	8	7	11	16	4
ATunes 1.10	133	0	4	6	6	9	17	4
Eclipse Platform 2.0	182	6	8	13	11	17	28	6
Eclipse Platform 3.5	1,029	1	7	14	13	18	53	10
JEdit 4.2	348	5	8	13	11	16	38	8
General	1,943	0	6	12	10	15	53	8

Table A.6: The total number of terms shared between the bug reports and the patched classes

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	313	3	12	20	15	27	46	12
Art of Illusion 2.4.1	193	1	9	15	12	24	30	9
ATunes 1.10	254	0	6	12	10	14	49	10
Eclipse Platform 2.0	349	8	17	25	22	28	59	13
Eclipse Platform 3.5	2,512	1	12	34	29	38	126	29
JEdit 4.2	739	6	16	27	20	26	104	23
General	4,360	0	10	26	21	31	126	24

APPENDIX B : Stop Words

a
about
above
according
accordingly
across
actually
afterwards
again
against
all
almost
alone
along
already
also
although
always
am
among
amongst
an
and
another
any
anybody
anyhow
anyone
anything
anyway
anyways
anywhere
apart
are
around
as
asm
assert
at
cry
default
definitely
despite
did
do
does
doing
double
downwards
due
during
dynamic_cast
each
eg
either
else
elsewhere
enough
enum
entirely
especially
et
etc
ever
every
everybody
everyone
everything
everywhere
except
explicit
export
extern
extends
false
far
few
final
in
inasmuch
inc
indeed
indicate
indicated
indicates
inline
instanceof
insofar
instead
int
interface
into
inward
is
it
its
itself
just
know
knows
lately
later
latter
latterly
less
long
lest
like
liked
likely
look
looking
looks
ltd
mainly
many
may
other
others
otherwise
ought
our
ours
ourselves
out
outside
over
overall
own
package
particular
particularly
per
perhaps
please
possible
presumably
probably
provides
private
protected
public
quite
rather
re
return
really
reasonably
regarding
regardless
regards
register
reinterpret_cast
relatively
respectively
return
sure
take
taken
tell
template
tends
than
thank
thanks
thanx
that
thats
the
their
theirs
them
themselves
then
thence
there
thereafter
thereby
therefore
therein
theres
thereupon
these
they
thin
think
this
thorough
thoroughly
those
though
through
throughout
throw
throws
whatever
when
whence
whenever
where
whereafter
whereas
whereby
wherein
whereupon
wherever
whether
which
while
whither
who
whoever
whole
whom
whose
why
will
willing
with
within
without
would
yet
you
your
yours
yourself
yourselves

auto
be
became
because
become
becomes
becoming
been
beforehand
behind
being
believe
below
beside
besides
between
beyond
bool
boolean
both
break
but
by
byte
came
can
cannot
cant
case
catch
certain
certainly
char
class
clearly
co
come
comes
con
concerning
consequently
consider
finally
float
for
former
formerly
friend
furthermore
further
give
given
gives
go
goes
going
gone
goto
got
gotten
had
happens
hardly
hasnt
have
having
he
hence
her
hereafter
hereby
herein
hereupon
hers
herself
hi
him
himself
his
hither
hopefully
how
howbeit
however
maybe
me
meanwhile
merely
might
mine
moreover
mostly
much
must
mutable
my
myself
namely
namespace
native
nearly
need
needs
neither
never
nevertheless
new
no
nobody
none
noone
nor
not
nothing
normally
now
nowhere
normally
null
obviously
of
off
often
oh
ok
okay
said
same
saw
say
saying
says
secondly
see
seeing
seem
seemed
seeming
seems
seen
selves
sensible
serious
seriously
several
shall
she
short
should
signed
since
sincere
sizeof
so
some
somebody
somehow
someone
something
sometime
sometimes
somewhat
somewhere
static
soon
sorry
static_cast
still
thru
thus
to
together
too
took
top
toward
towards
transient
tried
tries
true
truly
try
trying
typedef
typeid
typename
un
under
unfortunately
union
unless
unlikely
unsigned
until
unto
upon
us
useful
using
usually
various
very
via
virtual
void
volatile
want
wants
was

considering
const
const_cast
continue
corresponding
could
couldnt
hundred
i
ie
if
implements
immediate
import
on
once
ones
only
onto
operator
or
strictfp
struct
such
super
switch
synchronized
system
way
wchar_t
we
well
went
were
what

APPENDIX C : Summary of the Terms in Code Locations

Table C-1: Shared terms in arguments

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	428	0	10	27	17	50	66	23
Art of Illusion 2.4.1	221	0	3	17	7	14	109	29
ATunes 1.10	284	0	2	13	6	17	57	16
Eclipse Platform 2.0	405	1	11	29	27	31	82	25
Eclipse Platform 3.5	4,700	0	2	64	27	105	377	82
JEdit 4.2	1,168	0	3	43	14	22	718	136
General	7,206	0	3	43	16	46	718	81

Table C-2: Shared terms in attributes

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	423	0	0	26	14	56	81	30
Art of Illusion 2.4.1	180	0	0	14	0	13	75	23
ATunes 1.10	206	0	0	9	1	8	81	19
Eclipse Platform 2.0	521	0	13	37	24	45	129	39
Eclipse Platform 3.5	1,873	0	0	25	7	37	227	43
JEdit 4.2	518	0	0	19	1	15	263	51
General	3,721	0	0	22	6	30	263	39

Table C-3: Shared terms in comments

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	1,084	9	28	68	34	72	244	73
Art of Illusion 2.4.1	263	0	11	20	15	18	62	19
ATunes 1.10	525	0	3	24	12	27	97	30
Eclipse Platform 2.0	685	7	16	49	27	44	201	55
Eclipse Platform 3.5	9,730	0	31	131	75	137	1,157	192
JEdit 4.2	2,795	1	8	104	33	149	786	162
General	15,082	0	12	91	39	108	1,157	153

Table C-4: Shared terms in literals

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	819	0	9	51	31	81	193	57
Art of Illusion 2.4.1	49	0	1	4	5	5	10	3
ATunes 1.10	139	0	0	6	2	6	49	12
Eclipse Platform 2.0	278	0	5	20	16	30	67	20
Eclipse Platform 3.5	536	0	0	7	0	6	111	17
JEdit 4.2	253	0	1	9	4	10	61	14
General	2,074	0	0	12	3	12	193	26

Table C-5: Shared terms in method calls

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	1,316	0	8	82	36	64	657	162
Art of Illusion 2.4.1	256	0	5	20	8	22	113	30
ATunes 1.10	619	0	4	28	16	34	193	42
Eclipse Platform 2.0	629	10	16	45	41	53	119	34
Eclipse Platform 3.5	4,283	0	4	58	25	68	653	94
JEdit 4.2	2,466	3	17	91	37	55	1,378	260
General	9,569	0	6	58	24	56	1,378	133

Table C-6: Shared terms in method name

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	183	0	2	11	5	16	54	15
Art of Illusion 2.4.1	113	0	5	9	8	13	25	7
ATunes 1.10	140	0	1	6	4	9	29	8
Eclipse Platform 2.0	188	0	4	13	11	22	38	11
Eclipse Platform 3.5	1,152	0	2	16	9	17	124	21
JEdit 4.2	442	0	2	16	3	16	166	33
General	2,218	0	2	13	7	14	166	20

Table C-7: Shared terms in parameter

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	122	0	0	8	2	14	29	11
Art of Illusion 2.4.1	31	0	0	2	2	2	12	3
ATunes 1.10	148	0	0	7	0	4	59	15
Eclipse Platform 2.0	131	0	2	9	7	10	34	11
Eclipse Platform 3.5	1,814	0	0	25	8	27	169	39
JEdit 4.2	283	0	1	10	4	9	95	21
General	2,529	0	0	15	4	14	169	29

Table C-8: Shared terms in types

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	388	0	6	24	10	19	101	33
Art of Illusion 2.4.1	290	0	16	22	19	25	80	19
ATunes 1.10	233	0	1	11	4	10	65	16
Eclipse Platform 2.0	444	2	13	32	20	37	89	29
Eclipse Platform 3.5	5,404	0	11	73	37	109	321	82
JEdit 4.2	888	0	11	33	28	47	150	31
General	7,647	0	7	46	19	61	321	63

Table C-9: Shared terms in variables

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	505	0	4	32	15	26	157	49
Art of Illusion 2.4.1	82	0	0	6	6	11	15	5
ATunes 1.10	151	0	1	7	3	10	43	10
Eclipse Platform 2.0	200	0	5	14	14	22	33	12
Eclipse Platform 3.5	2,175	0	0	29	7	30	279	51
JEdit 4.2	876	1	4	32	15	27	403	76
General	3,989	0	1	24	8	23	403	49

Table C-10: Total terms in arguments

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	3,595	9	54	225	90	476	734	252
Art of Illusion 2.4.1	1,997	27	102	154	105	239	355	97
ATunes 1.10	2,307	2	33	105	66	173	434	103
Eclipse Platform 2.0	2,287	18	78	163	148	211	379	111
Eclipse Platform 3.5	17,207	0	34	233	88	367	974	271
JEdit 4.2	6,649	0	43	246	139	363	1,836	357
General	34,042	0	43	205	105	300	1,836	252

Table C-11: Total terms in attributes

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	1,919	0	20	120	65	187	374	130
Art of Illusion 2.4.1	1,243	31	44	96	61	96	291	76
ATunes 1.10	1,954	0	14	89	52	139	448	104
Eclipse Platform 2.0	1,987	26	62	142	113	197	301	99
Eclipse Platform 3.5	6,886	0	26	93	50	108	770	119
JEdit 4.2	3,438	0	13	127	62	87	1,588	308
General	17,427	0	26	105	61	128	1,588	160

Table C-12: Total terms in comments

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	6,024	57	105	377	224	389	1,276	392
Art of Illusion 2.4.1	1,553	36	78	119	78	158	270	80
ATunes 1.10	4,331	6	96	197	147	211	727	178
Eclipse Platform 2.0	3,316	42	108	237	139	175	1,209	296
Eclipse Platform 3.5	50,533	8	157	683	316	543	8,527	1,400
JEdit 4.2	19,415	9	94	719	300	1,407	3,931	923
General	85,172	6	100	513	209	437	8,527	1,039

Table C-13: Total terms in literals

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	3,747	0	30	234	118	514	650	245
Art of Illusion 2.4.1	245	0	2	19	24	27	37	13
ATunes 1.10	475	0	6	22	14	26	111	26
Eclipse Platform 2.0	608	0	15	43	45	55	104	34
Eclipse Platform 3.5	1,614	0	0	22	2	15	421	56
JEdit 4.2	1,756	0	8	65	35	128	231	64
General	8,445	0	1	51	15	45	650	107

Table C-14: Total terms in method calls

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	10,521	54	178	658	299	758	2,477	800
Art of Illusion 2.4.1	3,002	37	225	231	251	266	478	116
ATunes 1.10	5,875	6	93	267	232	284	1,167	263
Eclipse Platform 2.0	3,418	46	220	244	252	262	415	101
Eclipse Platform 3.5	22,270	0	36	301	142	347	1,729	385
JEdit 4.2	13,050	15	122	483	344	615	3,760	717
General	58,136	0	74	350	231	390	3,760	481

Table C-15: Total terms in method names

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	862	3	13	54	29	48	212	64
Art of Illusion 2.4.1	1,351	8	38	104	90	90	310	100
ATunes 1.10	1,070	4	12	49	31	73	142	43
Eclipse Platform 2.0	724	12	36	52	40	77	98	28
Eclipse Platform 3.5	4,317	0	18	58	39	96	303	59
JEdit 4.2	2,540	2	14	94	41	146	754	149
General	10,864	0	17	65	40	90	754	82

Table C-16: Total terms in parameter

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	1,006	0	12	63	28	74	329	88
Art of Illusion 2.4.1	642	9	20	49	49	77	83	28
ATunes 1.10	607	0	6	28	20	46	74	24
Eclipse Platform 2.0	603	3	16	43	26	77	122	40
Eclipse Platform 3.5	5,964	0	19	81	44	102	375	90
JEdit 4.2	2,956	1	16	109	53	195	531	121
General	11,778	0	15	71	38	87	531	87

Table C-17: Total terms in types

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	2,943	23	59	184	72	340	524	189
Art of Illusion 2.4.1	2,253	64	121	173	172	175	333	82
ATunes 1.10	2,517	9	51	114	107	167	268	78
Eclipse Platform 2.0	2,005	44	120	143	145	163	232	48
Eclipse Platform 3.5	17,665	4	41	239	123	449	1,004	242
JEdit 4.2	6,517	8	58	241	209	444	1,064	237
General	33,900	4	60	204	133	268	1,064	204

Table C-18 Total terms in variables

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	3,099	2	46	194	77	282	675	230
Art of Illusion 2.4.1	1,701	17	91	131	108	160	334	86
ATunes 1.10	1,149	2	19	52	38	76	170	46
Eclipse Platform 2.0	1,413	17	48	101	76	151	235	73
Eclipse Platform 3.5	10,483	0	14	142	72	252	879	184
JEdit 4.2	5,798	3	55	215	94	285	1,704	332
General	23,643	0	20	142	73	187	1,704	202

APPENDIX D : SrcML Document

D.1 Sample Source File

```
package log;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;

public class LogFile {

	private static String dirName;

	public LogFile() {
		dirName = "NotificationLog";
	}

	/* Copy log file
	*/
	public static void CopyLogFile(File logFile) throws IOException {

		if (logFile != null) {
			File logCopy = new File("notification_log_copy.txt");
			InputStream file = new FileInputStream(logFile);

			file.close();
			System.out.println("File copied.");
		}
	}
}
```

D.2 SrcML Document created from the source file in A.1


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<unit xmlns="http://www.sdml.info/srcML/src" language="Java" filename="E:\
\Resources\TestMethods.java"><package>package <name>log</name>;</package>

<import>import <name>java</name>.<name>io</name>.<name>File</name>;</import>
<import>import <name>java</name>.<name>io</name>.<name>FileInputStream</name>;</import>
<import>import <name>java</name>.<name>io</name>.<name>InputStream</name>;</import>
<import>import <name>java</name>.<name>io</name>.<name>OutputStream</name>;</import>
<class><specifier>public</specifier> class <name>LogFile</name> <block>{

        <decl_stmt><decl><type><specifier>private</specifier> <specifier>static</specifier>
<name>String</name></type> <name>dirName</name></decl>;</decl_stmt>

        <constructor><specifier>public</specifier>
<name>LogFile</name><parameter_list>()</parameter_list> <block>{
        <expr_stmt><expr><name>dirName</name> = "NotificationLog"</expr>;</expr_stmt>
        }</block></constructor>
        <comment type="block">/* Copy log file
        */</comment>

        <function><type><specifier>public</specifier> <specifier>static</specifier>
<name>void</name></type>
<name>CopyLogFile</name><parameter_list>(<param><decl><type><name>File</name></type>
<name>logFile</name></decl></param>)</parameter_list> <throws>throws
<argument><expr><name>IOException</name></expr></argument></throws> <block>{

        <if>if <condition>(<expr><name>logFile</name> !=
        <name>null</name></expr>)</condition><then> <block>{

        <decl_stmt><decl><type><name>File</name></type> <name>logCopy</name> =<init>
<expr>new
<call><name>File</name><argument_list>(<argument><expr>"notification_log_copy.txt"</expr></argumen
t>)</argument_list></call></expr></init></decl>;</decl_stmt>

        <decl_stmt><decl><type><name>InputStream</name></type> <name>file</name> =<init>
<expr>new
<call><name>FileInputStream</name><argument_list>(<argument><expr><name>logFile</name></expr>
</argument>)</argument_list></call></expr></init></decl>;</decl_stmt>


        <expr_stmt><expr><call><name><name>file</name>.<name>close</name></name><argument_
list>()</argument_list></call></expr>;</expr_stmt>

        <expr_stmt><expr><call><name><name>System</name>.<name>out</name>.<name>println</n
ame></name><argument_list>(<argument><expr>"File
copied."</expr></argument>)</argument_list></call></expr>;</expr_stmt>

        }</block></then></if>

    }</block></function>

}</block></class></unit>
```

APPENDIX E : Summary of the Terms in Bug Reports

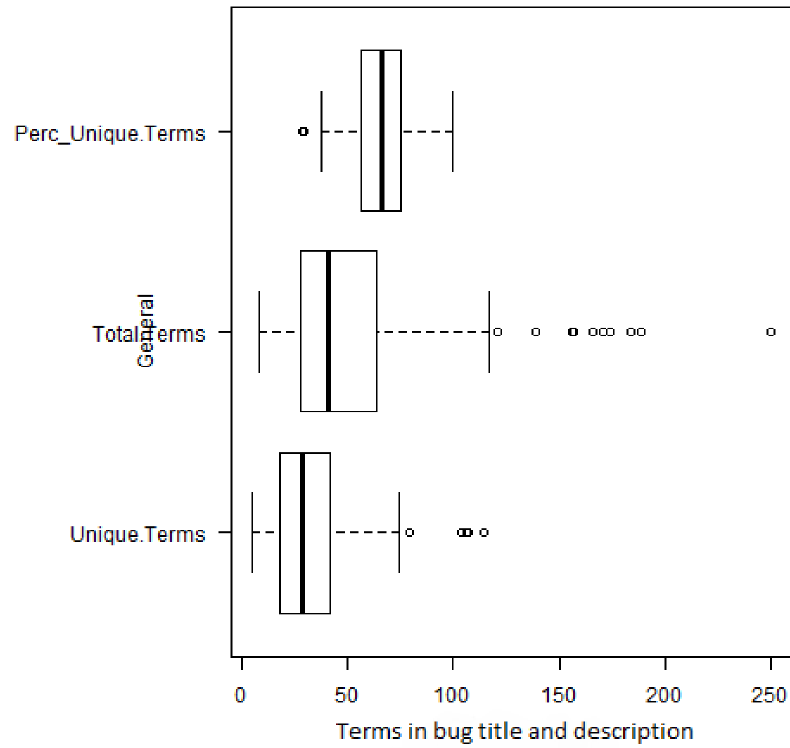


Figure E.1: The summary of the bug reports in all the software systems

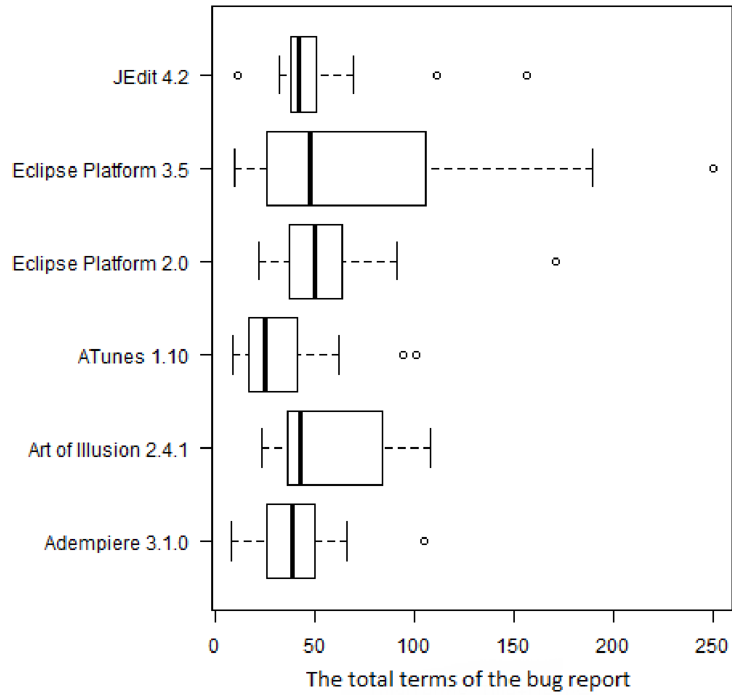


Figure E.2: The total number of terms of the bug report in each system

Table E.1: The total number of terms of the bug report in each system

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	650	8	27	40.62	39	47	105	23.24
Art of Illusion 2.4.1	557	23	36	55.7	42	80	108	29.66
ATunes 1.10	607	9	17	35.71	25	41	101	27.7
Eclipse Platform 2.0	779	22	37	59.92	50	64	171	38.95
Eclipse Platform 3.5	2,900	10	27	72.5	47	99	250	61.36
JEdit 4.2	934	11	38	51.89	41	50	156	32.71
General	6,427	8	28	56.38	41	64	250	45.48

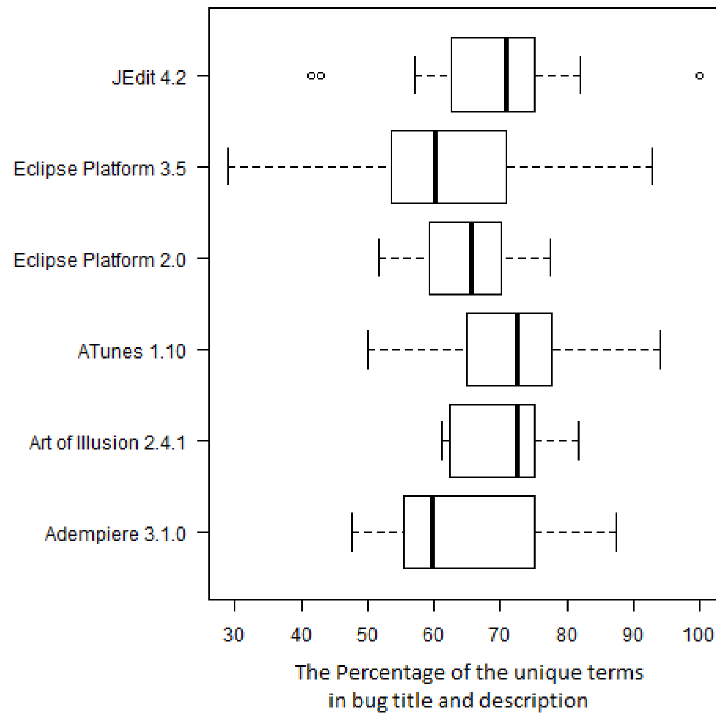


Figure E.3: The unique terms (%) of the bug reports in each software system

Table E.2: The unique terms (%) of the bug reports in each software system

System	n	min	q1	mean	median	q3	max	sd
Adempiere 3.1.0	1,041.754	47.61905	55.51	65.11	59.64	75	87.5	13.29
Art of Illusion 2.4.1	708.369	61.11111	63.41	70.84	72.54	74.73	81.81818	7.55
ATunes 1.10	1,218.176	50	65	71.66	72.5	77.78	94.11765	10.84
Eclipse Platform 2.0	856.9377	51.64835	59.32	65.92	65.71	70.21	77.5	8.4
Eclipse Platform 3.5	2,453.035	28.92562	54.24	61.33	60.19	70.89	92.85714	14.92
JEdit 4.2	1,229.853	41.44144	62.59	68.33	70.82	74.38	100	13.53
General	7,508.124	28.92562	56.78	65.86	66.4	74.8	100	13.12

References

[1] Rajlich, V., Software Engineering: The Current Practices. 2011: CRC Press.

[2] Jones, C., The Economics of Software Maintenance in the Twenty First Century (2006). Ryan North and James Choi: Leveraging Software Performance Engineering to Enhance the Maintenance Process, 2006. **68**.

[3] Rajlich, V.T. and K.H. Bennett, A staged model for the software life cycle. Computer, 2000. **33**(7).

[4] Takang, A.A. and P.A. Grubb, Software maintenance: concepts and practice. 1996.

[5] Singer, J. and T. Lethbridge, What's so great about 'grep'? implications for program comprehension tools. WWW: http://wwwsel. iit. nrc. ca/~ singer/grep/greptxt. html, 1997.

[6] Wu, S. and U. Manber, 'Agrep—A Fast Approximate Pattern-Matching Tool. Usenix Winter 1992, 1992: p. 153-162.

[7] Abou-Assaleh, T. and W. Ai, Survey of global regular expression print (grep) tools. 2004, Citeseer.

[8] Marcus, A., et al. An information retrieval approach to concept location in source code. in Reverse Engineering, 2004. Proceedings. 11th Working Conference on. 2004. IEEE.

[9] Blair, D.C. and M. Maron, Full-text information retrieval: further analysis and clarification. Information Processing & Management, 1990. **26**(3): p. 437-447.

[10] Bassett, B. and N.A. Kraft, Structural Information Based Term Weighting in Text Retrieval for Feature Location.

[11] Manning, C.D., P. Raghavan, and H. Schütze, Introduction to information retrieval. Vol. 1. 2008: Cambridge University Press Cambridge.

[12] Sanderson, M. and W.B. Croft, The history of information retrieval research. Proceedings of the IEEE, 2012. **100**(13): p. 1444-1451.

[13] Mizzaro, S., Relevance: The whole history. Journal of the American society for information science, 1997. **48**(9): p. 810-832.

[14] Djoerd Hiemstra, R.B.-Y., STRUCTURED TEXT RETRIEVAL MODELS. 2009.

[15] Haiduc, S., et al. Evaluating the specificity of text retrieval queries to support software engineering tasks. in Software Engineering (ICSE), 2012 34th International Conference on. 2012. IEEE.

[16] Yan, H., W. Adviser-Grosky, and F. Adviser-Fotouhi, Techniques for improved lsi text retrieval, in Computer Science. 2006, Wayne State University.

[17] Haiduc, S., Supporting Text Retrieval Query Formulation in Software Engineering. 2013, Wayne State University. p. 181.

[18] Hull, D.A., Stemming algorithms: a case study for detailed evaluation. JASIS, 1996. **47**(1): p. 70-84.

[19] Frakes, W.B. and C.J. Fox. Strength and similarity of affix removal stemming algorithms. in ACM SIGIR Forum. 2003. ACM.

[20] Jivani, A.G., A Comparative Study of Stemming Algorithms. Int. J. Comp. Tech. Appl, 2011. **2**(6): p. 1930-1938.

[21] Isbell, C. and P. Viola, Restructuring sparse high dimensional data for effective retrieval. 1998.

[22] Haiduc, S., et al., Automatic query reformulations for text retrieval in software engineering. Proceedings of the 2013 International Conference on Software Engineering, 2013: p. 842-851.

[23] Scanniello, G. and A. Marcus. Clustering support for static concept location in source code. in Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. 2011. IEEE.

[24] Collard, M.L., J.I. Maletic, and A. Marcus. Supporting document and data views of source code. in Proceedings of the 2002 ACM symposium on Document engineering. 2002. ACM.

[25] Fagerland, M.W. and L. Sandvik, The Wilcoxon–Mann–Whitney test under scrutiny. Statistics in medicine, 2009. **28**(10): p. 1487-1497.

[26] Feldt, R. and A. Magazinius. Validity Threats in Empirical Software Engineering Research-An Initial Survey. in SEKE. 2010.

[27] Ko, A.J., B.A. Myers, and D.H. Chau. A linguistic analysis of how people describe software problems. in Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on. 2006. IEEE.

[28] Sureka, A. and K.V. Indukuri. Linguistic analysis of bug report titles with respect to the dimension of bug importance. in Proceedings of the Third Annual ACM Bangalore Conference. 2010. ACM.

[29] Han, D., et al. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. in Reverse Engineering (WCRE), 2012 19th Working Conference on. 2012. IEEE.

[30] Linstead, E. and P. Baldi. Mining the coherence of GNOME bug reports with statistical topic models. in Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. 2009. IEEE.

ABSTRACT

ON THE RELATIONSHIP BETWEEN THE VOCABULARY OF BUG REPORTS AND SOURCE CODE

by

AMUNUGAMAGE WATHSALA BANDARA

December 2013

Advisor: Dr. Andrian Marcus

Major: Computer Science

Degree: Master of Science

The use of text retrieval techniques on concept location and bug localization yields remarkable benefits. The artifacts found in source code and bug reports contain important information related to the bug localization process. When locating the bugs, it is a programmer's task to formulate effective queries such that most of the predicted terms in the query appear in the relevant defect code, but not in most of the non-relevant source files. These queries are built based on the textual content found in the bug reports, especially the bug title and the description. A large body of research uses bug descriptions to evaluate bug localization techniques using text retrieval. All these studies are conducted under the implicit assumption that the bug description and the relevant source code files share important terms. This paper presents an empirical study that explores this conjecture. We found that bug reports share more terms with the patched classes than with the other classes in the software system. Moreover, the study revealed that the class names are more likely to share terms with the bug descriptions than other code locations. We also found that

more verbose parts of the source code, such as, comments share more words. Furthermore, we discovered that the shared terms may be better predictors for bug localization than some other text retrieval techniques, such as, LSI.