

**SUPPORTING SOURCE CODE COMPREHENSION DURING SOFTWARE
EVOLUTION AND MAINTENANCE**

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Nouh Alhindawi

August, 2013

UMI Number: 3618939

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3618939

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Dissertation written by

Nouh Alhindawi

M.S., Al-Balqa' Applied University, Jordan, 2006

B.S., Yarmouk University, Jordan, 2004

Approved by

Dr. Jonathan I. Maletic, Chair, Doctoral Dissertation Committee

Dr. Feodor F. Dragan, Members, Doctoral Dissertation Committee

Dr. Rouming Jin

Dr. Michael L. Collard

Dr. Catherine L. Smith

Accepted by

Dr. Javed I. Khan, Chair, Department of Computer Science

Dr. James L. Blank, Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES	VIII
LIST OF TABLES	XII
ACKNOWLEDGMENTS	XVI
CHAPTER 1 INTRODUCTION	1
1.1 Goals of the Research.....	3
1.2 Contributions	5
1.3 Publication Notes	6
1.4 Organization	7
CHAPTER 2 BACKGROUND AND RELATED WORK	8
2.1 Software Maintenance Overview	8
2.2 Historical Perspective for Program Comprehension	12
2.3 Information Retrieval in Software Engineering	19
CHAPTER 3 IMPROVING FEATURE LOCATION BY ENHANCING SOURCE CODE WITH STEREOTYPES.....	23
3.1 Approach Hypothesis	26
3.2 Related Work.....	27
3.2.1 Previous Work on Feature Location.....	27
3.2.2 Previous Work on Feature Location Using IR	28
3.3 Method Stereotypes.....	31
3.3.1 Stereotypes Definition.....	31
3.3.2 Method Stereotypes Taxonomy	31

3.4	Latent Semantic Indexing (LSI)	36
3.4.1	Why LSI?	38
3.4.2	LSI Processing Steps	39
3.5	LSI+Stereotypes for Feature Location	43
3.6	Experimental Study	45
3.6.1	Design and Objective of the Experimental Study	45
3.6.2	Evaluation Measures	47
3.6.3	Experiments Feature Selection and Determining Relevant Methods.....	49
3.6.4	Locating Features in HippoDraw System	50
3.6.5	Locating Features in Qt System	55
3.7	Discussion	58
3.8	Threats to Validity.....	66
3.9	Summary	67
	CHAPTER 4 SOURCE CODE INDEXING FOR FEATURE LOCATION	69
4.1	A Case Study of Feature Location with and without Comments	70
4.1.1	Code Comments Overview	71
4.1.2	Code Comments Categorizations.....	74
4.1.3	Case Study Comments Samples.....	77
4.1.4	Evaluation Strategy and Results Discussion	78
4.1.5	Study Recommendations.....	84
4.1.6	Summary	85
4.2	A Case Study of Feature Location with and without Function Calls.....	86

4.2.1	Function Calls Overview	87
4.2.2	Function Calls in Code Comprehension	88
4.2.3	Evaluation Strategy and Discussion.....	91
4.2.4	Summary	96
 CHAPTER 5 LSI-BASED SOLUTION FOR CATEGORIZING SOFTWARE		
	REPOSITORY COMMITS FOR MAINTENANCE	97
5.1	Repository Commits Overview	99
5.2	Version Control Systems.....	101
5.3	Commits Identification.....	102
5.4	Related Works	103
5.4.1	Previous Work on Software Repository Classification.....	103
5.4.2	Previous Work on the use of IR in Software Repository	104
5.5	Case Study: Adaptive Commits Identification.....	107
5.5.1	Latent Semantic Indexing (LSI) for Adaptive Commits.....	108
5.5.2	Case Study Evaluation	113
5.5.3	Experiments Findings.....	115
5.5.4	Discussion	122
5.5.5	Threats to Validity.....	127
5.6	Summary	127
 CHAPTER 6 SOURCE CODE QUERY ASSISTANT BUILDER..... 129		
6.1	Preprocessing Steps.....	131
6.2	Algorithm Pseudo-Code	133

6.3	Tool Program Setup	134
6.4	Tool Usage Instructions	134
6.5	Tool Interface Components Description	136
6.5.1	File Menu	136
6.5.2	User Word Entries	136
6.5.3	User Selected Terms/Entries	138
6.5.4	Synonyms List.....	138
6.5.5	Matching Document Count	138
6.5.6	Co-occurring Term Count	139
6.5.7	Function Name Terms Info	139
6.5.8	Co-occurring Terms	139
6.5.9	Number of Co-occurrences and Total Occurrences	139
6.5.10	Percentage of Matching Functions Containing Term	140
6.5.11	Percentage of All Functions Containing Term.....	140
6.6	Related Work.....	140
6.7	Tool Evaluation	142
6.8	Summary	146
 CHAPTER 7 AN ENVIRONMENT TO CONDUCT EXPERIMENTS IN		
INFORMATION RETRIEVAL FOR SOFTWARE ENGINEERING.....		
7.1	TraceLab Overview	148
7.2	TraceLab Features	149
7.2.1	Components.....	149

7.2.2	Working with Components	151
7.2.3	Running an Experiment.....	152
7.3	TraceLab Components	152
7.3.1	LSI Space Builder	152
7.3.2	LSI Querier.....	153
7.3.3	LSI Data Importer	156
7.3.4	LSI Data Exporter	156
7.4	Retrieval Case Study: Traceability Recovery Process	159
7.5	Summary	164
CHAPTER 8 CONCLUSIONS AND FUTURE WORK		165
APPENDIX A AN EXPERIMENT RESULTS OF QT SYSTEM COMMITS		
	CATEGORIZATION WITH AND WITHOUT STEMMING	169
APPENDIX B HIPPODRAW QUERIED FEATURES (11 FEATURES) AND THE		
	STEREOTYPES FOR ALL RELEVANT METHODS.	173
APPENDIX C RULES FOR STEREOTYPE IDENTIFICATION		184
REFERENCES.....		186

LIST OF FIGURES

Figure 2-1. The IEEE maintenance process activities.	10
Figure 3-1. A code snippet of the HippoDraw C++ Class DataSource after re- documenting with the method stereotypes.	35
Figure 3-2. LSI Steps: The corpus is represented as a term-document matrix (term x document), then the matrix is then subject to SVD, computes the term and document vector spaces.	37
Figure 3-3. Retrieving the results for a query (q).	43
Figure 3-4. The feature location process used in this study. First, stereotypes are computed and added as comments in the source code. Next preprocessing is done to produce a corpus as input to Latent Semantic Indexing (LSI). LSI produces a vectorized representation of the corpus that queries can be made against.	44
Figure 3-5. Precision results for the HippoDraw case study show that LSI+S (blue) had an equal or higher precision then LSI (yellow) alone.	53
Figure 3-6. Recall results for the HippoDraw case study show that LSI+S (blue) had an equal or higher recall then LSI (yellow) alone.	53
Figure 3-7. Precision results for the Qt case study show that LSI+S had better precision then LSI in almost all cases.	55
Figure 3-8. Recall results for the Qt case study show that LSI+S had better recall then LSI in almost all cases.	58
Figure 4-1. A feature diagram for source code indexing.	72
Figure 4-2. A snippet for an example about documentary comments [Spuida 2002].	76

Figure 4-3. Asnippet for an example about descriptive comments [Spuida 2002].	76
Figure 4-4. Qt-system experiments results average.....	80
Figure 4-5. HippoDraw-system experiments results average.	81
Figure 4-6. KOffice-system experiments results average.....	81
Figure 4-7. Ranking comparison for all relevant methods of all taken systems queries. Three cases taken, the red color shows the percentage of relevant methods that best answered when including the comments. The yellow color shows the percentage when excluding the comments, and finally the blue color shows the percentage when including and excluding the comments do the same.	82
Figure 4-8. Comparison results (Recall) for the relevant methods of all queries. Three cases taken, one with including all comments, and one without including any comments, and the finally one, is when including the comments except the bug comments.....	82
Figure 4-9. Comparison results (Precision) for the relevant methods of all queries. Three cases taken, one with including all comments, and one without including any comments, and the finally one, is when including the comments except the bug comments.....	84
Figure 4-10. The mandatory actions that must be considered when indexing source code.	89
Figure 4-11. Recall results for Qt system experiment.	92
Figure 4-12. Precision results for Qt system experiment.....	92
Figure 4-13. Average of recall and precision for Qt system experiment results.	93

Figure 4-14. Recall results for HippoDraw system experiment.	93
Figure 4-15. Precision results for HippoDraw system experiment.	94
Figure 4-16. Average of recall and precision for HippoDraw system experiment results.	95
Figure 4-17. Average of recall and precision for KOffice system experiment results.	95
Figure 5-1. Repository commits categorization steps.	98
Figure 5-2. A Snippet of KOffice subversion log.	102
Figure 5-3. Adaptive commits identifying approach.	109
Figure 5-4. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for KOffice.	124
Figure 5-5. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for Extargear/Graphics.	125
Figure 5-6. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for OSG.	126
Figure 6-1. QueBA algorithm pseudo-code.	133
Figure 6-2. A snapshot of an input text file for a list of code function's names.	135
Figure 6-3. Tool interface components.	137
Figure 6-4. Average of recall results for the Qt experiments.	143
Figure 6-5. Average of recall results for the HippoDraw experiments.	144
Figure 7-1. Home page for TraceLab showing the component's library.	150
Figure 7-2. LSI Space Builder component.	151
Figure 7-3. LSI Querier component.	153

Figure 7-4. An example of experiment set up of how to preprocess a loaded corpus and set of queries to the LSI Space Builder and LSI Querier respectively.	154
Figure 7-5. An example of experiment set up of how to use the LSI Space Builder with the LSI Data Exporter.	155
Figure 7-6. LSI Data Importer component.	156
Figure 7-7. LSI Data Exporter component.	157
Figure 7-8. An example of an experiment set up of how to use LSI Querier, and LSI Data Importer to query the corpus, which was saved to the file system. The queries are preprocessed using the right side of the graph.	158
Figure 7-9. Snapshot for the results of running one query sample (Results with first 2000 documents & LSI Dimensionality =300).....	163
Figure 8-1. Steps for automatically identifying and re-documenting the source code with method stereotypes [Dragan et al. 2006].	184

LIST OF TABLES

Table 3-1. Taxonomy of method stereotypes as given in [Dragan et al. 2006]. The taxonomy is mainly focused on the C++ programming language. Methods may be labeled with one or more stereotypes.	32
Table 3-2. Details of the corpus used as input to LSI for each of the two systems used in the experimental study.....	47
Table 3-3. HippoDraw Feature description, applied query, and the number of relevant methods for each feature.....	52
Table 3-4. Result of HippoDraw system for three measurements; Total effort measurement (Σ EM), Position of first relevant document (PFR), and Position of last relevant document (PLR).....	54
Table 3-5. Qt Features descriptions; feature name, query used, and number of relevant methods to each feature.	56
Table 3-6. Result of Qt system for three measurements; Total effort measurement (EM), Position of first relevant document (PFR), and Position of last relevant document (PLR).	57
Table 3-7. The difference between the positions of the first relevant and the last relevant method for each query result in Hippodraw and Qt. The last column is the percentage improvement using LSI+S.....	61

Table 3-8. Description of eight bugs (which corresponding to 14 features) from Qt bug reports. The table cloumn's show the bug number, followed by the number of features that relate to each bug, the component name, and the number of relevant methods.....	64
Table 3-9. Comparison results for locating the relevant methods for bug 11204.....	64
Table 3-10. Distribution of stereotypes for the relevant methods over both studies. The other 15 were a variety of different stereotypes with no one category making up more than 2%.....	65
Table 3-11. Stereotypes types for the relevant methods of the feature “remove item”. ...	66
Table 4-1. Comments Density for the three systems, computed based on the number of lines of code of each system separately.....	80
Table 5-1. Adaptive and non-adaptive commits for the examined systems.	115
Table 5-2. Frequency of the top 12 average terms in the adaptive commits and their frequency in non-adaptive commits.	116
Table 5-3. Details of the used corpora. total number of terms for each system, vocabulary size (number of terms after stop list), number of parsed documents, and the dimensionality used for each system.	117
Table 5-4. KOffice topics and the related terms for each topic.	119
Table 5-5. Extragear/Graphics topics and the related terms for each topic.	120
Table 5-6. OSG topics and the related terms for each topic.	121
Table 5-7. The size of the union set reported as a ratio of the total discovered adaptive commits.	126

Table 6-1. Details of the corpora that were used in the experimental study.....	143
Table 6-2. List of all relevant methods/functions for modify mode feature.....	145
Table 7-1. Elements of the KDE/KOffice source code documentation and list settings used in the experiments.	161
Table 7-2. Recovered links, recall, and precision using cosine value threshold for KDE/KOffice.....	162
Table 8-1. The resulted topics without stemming, number of topic chosen =5.....	169
Table 8-2. The resulted topics without stemming, number of topic chosen =10.....	170
Table 8-3. The resulted topics with stemming, number of topic chosen =5.....	170
Table 8-4. The resulted topics with stemming, number of topic chosen =10.....	171
Table 8-5. The resulted topics for the period 2005-2007 with stemming, number of topic chosen =5.....	172
Table 8-6. The resulted topics for the period 2008-2010 with stemming, number of topic chosen =5.....	172
Table 8-7. Stereotypes type of all relevant methods for the feature “change font size”. 173	
Table 8-8. Stereotypes type of all relevant methods for the feature “change font style italic”.	174
Table 8-9. Stereotypes type of all relevant methods for the feature “update zoom mode”.	175
Table 8-10. Stereotypes type of all relevant methods for the feature” change printer settings”.	176

Table 8-11. Stereotypes type of all relevant methods for the feature” add item to canvas”.	
.....	177
Table 8-12. Stereotypes type of all relevant methods for the feature” remove item from canvas”	
.....	178
Table 8-13. Stereotypes type of all relevant methods for the feature” change mouse property”	
.....	179
Table 8-14. Stereotypes type of all relevant methods for the feature” change cut color”.	
.....	180
Table 8-15. Stereotypes type of all relevant methods for the feature” change representation color”	
.....	181
Table 8-16. Stereotypes type of all relevant methods for the feature” make new display”.	
.....	182
Table 8-17. Stereotypes type of all relevant methods for the feature” update axis modeling”	
.....	183
Table 8-18. Stereotypes Identification Rules.	185

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my wonderful parents, *Talal* and *Ghazyah*, for their enormous support, infinite patience, and unwavering belief towards me, as always. My loving brothers and sisters, *Mohammad*, *Ahmad*, *Khalid*, *Khawla*, *Reem*, *Jihan* and *Khadejih*, have always been on my side. There is a litany of family members and friends who are not individually mentioned here, but they certainly made a difference.

I would like to show my greatest appreciation and gratitude to my research advisor, *Prof. Jonathan I. Maletic*, who was always there in times of great needs and deeds, without his guidance and persistent help, this dissertation would not have been possible. I am grateful to all my colleges and friends in software engineering development laboratory (SDML), *Computer Science Department*, and *Kent State University*.

Finally, I also greatly thank my dissertation committee for their appreciated services, efforts, valuable feedback, and participations.

Thank you one and all.

Nouh Alhindawi

August, 2013, Kent, Ohio

CHAPTER 1

Introduction

Software evolution is a very costly, broad, and complicated problem as it requires very deep understanding of the target system source code. Moreover, professional developers must be familiar with the system undergoing change in order to accomplish the required maintenance tasks. The process of expressing the behavior, the organization, the components relationship, and the architecture of the software that are not explained in the documentation requires great effort to be complete and precise. Therefore, while exploring and searching the source code, the developer must take into account both the structural characteristics of the source code and the nature of the problem domain, for example, internal comments, external documentations, variable names, and annotations. This constitutes the problem of program comprehension [Maletic and Marcus 2001, Maletic and Kagdi 2008, Cleary et al. 2009]. Comprehension activities constitute a major portion of modern software project maintenance and evolution efforts and requires roughly 40 percent of the whole cost of any software project [Turver and Munro 1994]. Other estimates show that programmers spend more than half of their time in exploring and reading the source code [Binkley and Lawrie 2010, Binkley and Lawrie 2010] when adding new features to a system.

Understanding a software system is a prerequisite before making any changes to that system. It requires the developer to gather the scattered information across the software systems (source code), and then present the extracted information in readable

and understandable view. This task is time consuming and error prone, especially when the system is large and complex. Quite a lot of research has been done investigating ways to decrease the time and the effort needed to understand a system. In the last decade, researchers have proposed techniques that help in gathering the most important scattered information and presenting it in a good manner that helps in understanding the intended system [Salton and McGill 1983, Maletic and Marcus 2001, Poshyvanyk 2009].

When adding a new feature or modifying existing features in a system, programmers must identify which parts of source code are most relevant to the intended feature. Identifying these relevant parts in the context of Software Engineering is called feature location, which is also considered as a part of the incremental change procedure. A feature is defined as the behavior of the system that is observed based on user's choice. A feature is an observable aspect of a system while a concept is defined as a human-oriented expression of the computational objective [Wilde and Scully 1995, Marcus et al. 2004, Liu et al. 2007, Poshyvanyk 2009, Dit et al. 2011]. So, we can say that a feature is a concept that is coupled to executions with some predefined input.

This dissertation is focused on the problem of comprehension to support the evolution of large-scale software systems. The research concerns how software engineers locate features and concepts along with categorizing changes within very large bodies of source code along with their versioned histories. More specifically, we examine how advanced Information Retrieval and text matching can be utilized and enhanced to support these software engineering tasks.

1.1 Goals of the Research

During the past 10 years Information Retrieval (IR) and Natural Language Processing (NLP) approaches have been used to help address the problem of feature location [Marcus et al. 2004, Pollock et al. 2007, Poshyvanyk et al. 2007, Cleary et al. 2009, Poshyvanyk et al. 2009, Dit et al. 2011, Hill et al. 2011, McMillan et al. 2011, Poshyvanyk et al. 2013]. These techniques treat the identifiers and comments within the source code as a corpus and then advanced methods are used then for indexing and searching within the corpus. The documents sought here are typically methods or functions within the system. The identifiers and comments in the source code represent what is called semi-structured textual information [Marcus et al. 2004, Poshyvanyk et al. 2007]. This information when examined and analyzed is very valuable for maintaining software systems. Thus, using Information Retrieval techniques to leverage this information assists the developer in maintenance tasks such as feature location [Maletic and Marcus 2000, Marcus et al. 2004, Binkley and Lawrie 2010, Dit et al. 2011, McMillan et al. 2011], and supports design of incremental changes to the software [Poshyvanyk 2009].

While IR and NLP approaches have shown to be useful there is room to improve the accuracy of these methods. Software and comments are not natural language. So any mapping from natural language queries to source code will typically be imperfect.

This research is not aimed at directly improving IR or NLP approaches. Rather it is aimed at understanding how additional information can be leveraged to improve the final results. For example, what information could be added to source code (in the form

of comments) that would improve the results of an IR approach for the task of feature and concept location? That is, how can we augment the source code (corpus) with new or derived information in a manner that will improve the accuracy of query results, or by excluding some of source code artifacts that have a negatively effect on source code indexing.

The idea is that we can enrich the corpus so the model built by the IR method better models the system. This information must be an abstraction beyond the identifiers and comments already contained within the code. We feel that abstract descriptions of low-level program behavior that can be derived directly from the code could be a valuable source for improving the accuracy of such activities.

Adding new terms to a corpus is a form of supervision for an unsupervised method. Apriori knowledge is often used to direct and supervise machine-learning and information-retrieval approaches [Perotte et al. 2011]. Here, we derive this information from the corpus itself. Others have used similar approaches based on ontological information [Müller et al. 2004] and inferred semantics from term distribution [Teevan 2001]. From an information theoretic standpoint the addition of relevant information will improve the results of an information-retrieval technique [Huibers et al. 1996, Binkley and Lawrie 2003]. That is, more information is better, so long as you don't add noise.

An example of such information is method/function stereotype information [Dragan et al. 2006]. Stereotypes are abstract descriptions of the behavior and roles of a method or function which can be derived via static or dynamic program analysis and easily put back into the code as annotations (i.e., comments). Another example is call

graph information. This can easily be extracted from a program via static program analysis and added to each function via a comment.

Another source for augmentation is the semantic information about words that available in tools such as WordNet [Oram 2001, Stanchev 2012]. When searching for a feature or concept in source code a developer may not use the actual term used within the code. Thus augmentation synonyms may be beneficial.

In summary the goals of this dissertation are to investigate different methods for deriving and deploying additional information in conjunction with IR and NLP approaches. Additionally, the work attempts to identify the types of information best suited for enhancing results on specified tasks.

1.2 Contributions

The main contributions of this work involve improving on the results of previous work in feature location and source code querying. However, the research also involves the development of a number of software tools that have broader applications to many other software engineering problems. The contributions of this thesis are outlined below.

- Demonstrates that the addition of statically derived information from source code can improve the results of IR methods applied to the problem of feature location.
- Shows the effects of excluding certain textual information (e.g., comments and function calls) when performing source code indexing for feature and concept location.

- Demonstrates an IR-based method of natural language topic extraction (semantically) that assists developers in gaining an overview of past maintenance activities based on software repository commits.
- Demonstrates the use of problem and solution domain knowledge and word meaning in augmenting user queries for feature and concept location.
- Introduces a platform for enhancing program comprehension and facilitates software engineering research.

1.3 Publication Notes

CHAPTER 3 results are published at the 29th IEEE International Conference on Software Maintenance (ICSM'13) [Alhindawi et al. 2013]. CHAPTER 4 results are written up and will be submitted to the 20th Working Conference on Reverse Engineering (WCRE'13) [Alhindawi et al. 2013]. CHAPTER 5 results are also written and will be submitted to the 20th Working Conference on Reverse Engineering (WCRE'13) [Alhindawi et al. 2013]. CHAPTER 6 results are planned to be submitted to the 36th International Conference on Software Engineering (ICSE'14) [Alhindawi et al. 2014]. CHAPTER 7 is published at the 33rd IEEE International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'13) [Alhindawi et al. 2013].

1.4 Organization

The remainder of the dissertation is organized as follows. A brief overview on software maintenance for general background, program comprehension, and Information Retrieval (IR) in Software Engineering (SE) is introduced along with a review of the literature on those topics in CHAPTER 2. A novel approach (LSI+S) to improve feature location by enhancing the corpus of source code with static information is presented in CHAPTER 3. CHAPTER 4 presents a study that examines the effects of excluding comments and function calls when performing source code indexing for feature and concept location purposes. CHAPTER 5 introduces an IR based approach for categorizing repository commits based on maintenance types into adaptive, corrective, perfective, and preventive. A novel platform tool to assist with the creation of queries for any software artifacts is presented in CHAPTER 6. An environment to conduct experiments in Information Retrieval for Software Engineering is introduced in CHAPTER 7. Finally, we conclude in CHAPTER 8 along with the discussion on open issues and future directions.

CHAPTER 2

Background and Related Work

This chapter presents a brief overview on software maintenance for general background. Following is a discussion of program comprehension and the work done in this field. Lastly, the subject of Information Retrieval (IR) in Software Engineering is introduced along with a review of the literature on that topic.

2.1 Software Maintenance Overview

Software maintenance stems from the broader domain of Software Engineering. Generally, software maintenance is defined as the adaptation and modification of a software product after delivery for a number of motives. The first of these is the need to fix faults that may present themselves at later stages. Secondly, it is to gain the most significant performance from the software. The third of these reasons is to ensure that the software meets most of the modern requirements. Finally software maintenance facilitates future maintenance exercise, and gives the software the ability to deal with new environments [Lehman 1980].

Software maintenance is one of the major elements of the software life cycle; this is traced back to the fact that it plays a major, important, and central role in the software development process [Lehman et al. 1997]. As estimated in [Storey 2005], in the world, there exist over 100 billion unstructured, patched and poorly documented lines of code in software productions. Thus, this makes the vitality of software maintenance even more

pronounced, for it is difficult to implement any changes to these productions, and solve the problems that may arise within those types of productions, especially the already delivered ones [Banker et al. 1991]. Figure 2-1 shows the maintenance process activities that the developers perform for updating software based on any change request.

To further add to what was mentioned earlier, almost fifty percent of the development work done is dedicated to maintenance tasks [Lientz et al. 1978, Lientz and Swanson 1980] ,therefore; improvements in this field are capable of significantly decreasing the costs associated with the development process, and have the potential to save developer's time and efforts. Finally, any improvements introduced would positively influence software productivity.

Software maintenance is thought of as the last phase of the software development life cycle. Following the release and delivery of the product to the end users, the experienced maintainers preserve and keep maintaining the software, updating it with reference to user's change requests, and responding to changes occurring in the environment, in order to keep it up to date.

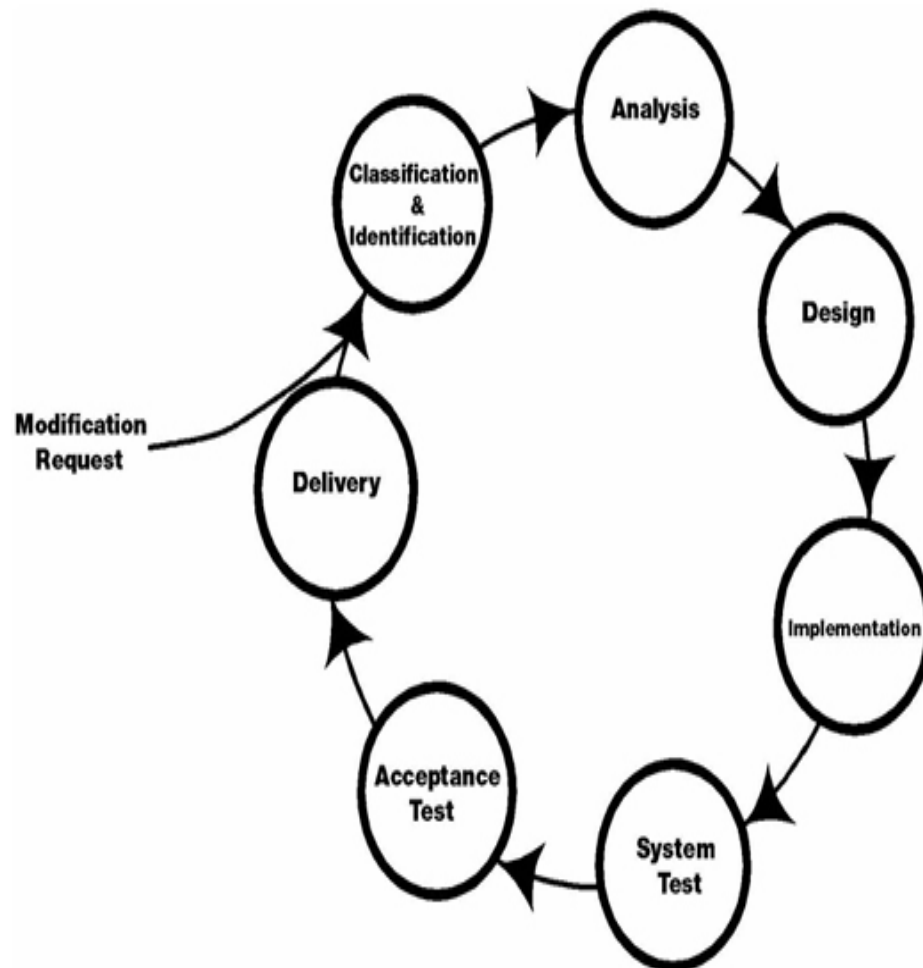


Figure 2-1. The IEEE maintenance process activities.

Characteristically, software maintenance activities are classified into four main types or parts. These four types can distinguish any change applied to the software system. The first one is the corrective maintenance; and it is concerned with fixing bugs, logic and design errors, and coding errors in the source code [Maletic and Reynolds 1994].

The second type of software maintenance is adaptive maintenance; it concentrates on adapting the software to new environments (hardware or software). This particular

type of maintenance activity is performed less frequently than other types of maintenance such as corrective maintenance [Schach et al. 2003]. In chapter five, an automatic approach for identifying the adaptive commits from software repository is presented.

The third of these maintenance types is the perfective maintenance. Perfective maintenance is targeted at modernizing the software according to changes in the user's requirements. It is primarily utilized to enhance the system's functions with the intent of improving the performance of the software, along with providing a user interface that is friendly more. An example of perfective maintenance would be modifying a program specializing in accounting, to include a new union payment [Niessink and Vliet 2000].

Finally, there is preventive software maintenance. This genre of software maintenance handles the affairs of software documentation updates (e.g., adding comments). Furthermore, the developers specializing in this type of maintenance dedicate much of their efforts towards producing a software that is more maintainable and more understandable for future tasks [Niessink and Vliet 2000].

As a general rule, corrective maintenance is considered traditional maintenance by researchers, whereas all the other types of maintenance are considered software evolution [Bennett and Rajlich 2000].

When developers deal with a large and a rather complex software system, it is not easy to make changes without having a complete and utter understanding the interactions and the relations that exist between the different system components [Maletic and Marcus 2001, Hussein et al. 2009]. Therefore, there arises an urgent need for developers to be precise and punctual about why they are trying to comprehend the software, what they

are trying to comprehend, who's trying to comprehend it, and when they need to do so [Kagdi et al. 2007].

Erdos and Sneed [Erdos and Sneed 1998] were able to produce a novel tool to support software maintenance. Additionally, they have proposed that a programmer in the process of maintaining unfamiliar software must answer the following significant questions:

1. Where is a particular function invoked?
2. What are the arguments and results of a function?
3. How does control flow reach a particular position?
4. Where is a particular variable set, used or needed?
5. Where is a particular variable identified?
6. Where is a particular data object accessed?
7. What are the inputs and outputs of a unit?

Using tools throughout software maintenance definitely makes the tasks simpler, and enhances the effectiveness and the output of the software. Moreover, reusing software increases productivity and improves maintainability by employing the already existing software parts [Bennett and Rajlich 2000, Binkley and Lawrie 2010].

2.2 Historical Perspective for Program Comprehension

Typically, the study of program comprehension can be characterized by two instruments, which are the theories and the tools available in this regard. The theories gain their importance in the sense that they supply rich clarification about how

developers understand any system's software. In addition to the theories, there are the tools that are utilized to support and help in comprehension activities [Storey 2005].

In general, the purpose of comprehension is mainly dependent on the task of interest. That is to say, there must be some cause to force the development team to comprehend software artifacts. For example, a developer may try to localize a bug/feature, or assess possible or obtainable changes to an API (Adaptive Changes). Most frequently a specific concept or particular feature is inspected in the software, and this concept/ feature is most often related to a user change request [Kagdi et al. 2007].

Program comprehension is one of the most important steps in addressing many Software Engineering and maintenance tasks. It is extremely crucial for correctly gathering knowledge about the program at hand [Shneiderman and Mayer 1979, Rist 1986]. This knowledge is usually diverse, meaning that several aspects are integrated into it like maintenance [Littman et al. 1986, Mayrhauser and Vans 1997], documentation [Etzkorn et al. 1999], debugging [Hartman 1991, Mayrhauser and A 1994], reuse [Biggerstaff and Richter 1987, Kim and Stohr 1998], and verification [Choi and Scacchi 1990, Canfora et al. 1993].

In Software Engineering, program comprehension is constantly taken into consideration, and it poses as a serious concern for the developers. When new programmers are assigned to an old code, they often complain about understanding it, and express their views about the code being unintelligible; therefore, software comprehension is very crucial and is especially needed in the occasions when old seasoned programmers leave their projects. That is, the absence of the original

programmers slow down the understanding of the software, and thus negatively impacts comprehension.

Unfortunately, the usual case is that the programmers who originally developed the system are no longer available to assist, or sometimes parts of the software may be certified from a third party that monitors the maintenance process. In both of these situations, the developers who are designated for maintaining the system must understand it [Brooks 1983, Storey 2005]. In other words, it is of an absolute necessity that every associate on the maintenance team develop a comprehensive understanding of the software [Toffolon and Dakhli 2008].

Source code contains a lot of information that is either peripheral or hidden by other components. One useful approach that developers have suggested is to facilitate program understanding and program maintenance by extracting and clearly representing the information that is most important in source code.

The research field of program comprehension is characterized as rich, containing various and mixed topics, which coupled with changes in models, and research environment in the last few decades. The comprehension process can be categorized into two basic styles; the first being Top- down comprehension, while the second is Bottom-up comprehension. For Top-down comprehension, Brooks [Brooks 1983] hypothesizes that developers usually understand a completed program in a top-down fashion by restructuring facts about the area, topics, and objectives of the program, and linking those facts to the system's source code. Soloway and Ehrlich [Littman et al. 1986] examined

the style of Top-down comprehension, and concluded that this style is used when the code or type of code is recognizable.

The second category is Bottom-up comprehension [Shneiderman and Mayer 1979], which supposes that developers initially read the software code lines, and then make an effort to group them into an advanced level of abstraction. Subsequently, the new levels are combined incrementally until the developers come to acquire a deep understanding of the intended software program. Pennington also describes the Bottom-up model [Pennington 1987]. She concludes that at the beginning of the comprehension process, developers build up an abstraction for control flow of the program; this abstraction contains the order and the sequence of the most important operations in the program.

Von Mayrhauser and Vans [von Mayrhauser and Vans 1993], provided rather important recommendations regarding tool maintenance for reverse engineering tasks comprehension. They identified fundamental information needs according to recognized tasks; additionally, they recommended a set of capabilities for tools that satisfy those needs. Martin and McClure [MARTIN and MCCLURE. 1983] concluded that using an automated tool during system maintenance decreases the effort and time needed noticeably.

If developers set out to accomplish a simple task such as finding why a particular variable has an unacceptable value in a simple program, then, only a small portion of code must be understood in order to be changed. However, a system's software may have many fundamental problems or the system itself may be complex and large. In other

words, the point here is that larger and more complex software projects are in crucial need for management control. In this case, it is better for the developers to re-engineer all or parts of the system software because they need to account for the entire interactions taking place within the system to perform system software re-engineering.

This would be in addition to the fact that in this case, it is essential for developers to realize and comprehend how the different fragments and components of the software are related, how the software is built, and what effect any modification may cause [Storey 2005]. Van Vliet [Vliet 2000] concluded that less maintenance is needed when less code is written.

Generally, the field of program comprehension is up to date with respect to supporting tools that are either new or adapted to address program comprehension requirements for new software development and maintenance tasks [Penta et al. 2007]. Storey [Storey 2005], reviewed some of the key cognitive theories of program comprehension that have appeared over the past three decades, and he explored how the tools that are generally used at the present were developed and updated to improve and support program comprehension tasks. In [Storey et al. 1997, Storey 2005] the authors introduced user studies to discover how, and how well, different program understanding tools in fact assist programmers in understanding the software artifacts.

Software comprehension tools aid engineers in capturing the benefit of new added code. They are necessary as economic demands require a maintenance engineer to rapidly and successfully develop comprehension of the parts of source code that are relevant to a maintenance request. In general, the tools make program comprehension more effective

[Binkley and Lawrie 2010]. In [Penta et al. 2007], the authors concluded that any program-comprehension tool has to be proven to generate benefits throughout maintenance tasks. Therefore, program comprehension tools play a supporting role in other Software Engineering activities such as design, development, maintenance, and re-documentation.

There are a lot of tools that were built in order to help in program understanding, and to simplify the comprehension task for a maintainer. For instance, SNiFF+¹, is one of the best well-known commercial tools, and it was produced to assist in source code understanding and to facilitate maintenance tasks. Ghinsu is a program understanding framework described in [Livadas and Alden 1993], and, SeeSoft [Eick et al. 1992] is a tool for visualizing software statistics from a variety of sources. Such tools are helping drastically in improving and accelerating a developer's overview of complex system software [Niessink and Vliet 2000]. Moreover, those tools have practical benefits in terms of generating fewer bugs or an easier time comprehending a new piece of source code.

Other tools employ IR for both the comprehension task and understanding task during initial software development and during software maintenance and evolution [Storey and Muller 1995, Binkley and Lawrie 2010].

In addition, researchers with the goal of improving the comprehension process and saving developer's time and effort presented a set of recommended tools to guide

¹ <http://www.openntf.org>

system software navigation while exploring and understanding a system. Mylar [Kersten and Murphy 2005] used a degree-of-interest model to distinguish and mark the non-relevant files from the file explorer in Eclipse. NavTracks [Singer et al. 2005] supported a tool that recommends which files are related to the currently chosen files. Deline et al [DeLine et al. 2005] , also presented a framework to improve the software navigation process. On the other hand, Robillard [Robillard and Murphy 2003], presented a FEAT tool that is capable of providing suggestions using graphs manually created by users, to enhance navigation effectiveness and improve the comprehension process. RedHat Source-Navigator² is another tool that is being developed to assist in understanding complex system software.

The Searchable Bookshelf [Elliott Sim et al. 1999], is designed to help in producing and navigating software structure diagrams. Rigi³, enables the users to visualize different aspects of a software system (subsystem, files, etc.) using diagrams shapes, and it also shows interactions between the different system components. SHriMP [Storey and Muller 1995], employs hyperlinks in order to navigate the source code, and gives a better view of the source code components.

Researchers in the field of Software Engineering suggested and used alternative approaches that do not involve giving great amounts of attention to software comprehension. Examples of such approaches include Refactoring [Fowler 1999].

² <http://www.sources.redhat.com/sourcenav>

³ http://www.rigi.cs.uvic.ca/downloads/pdf/rigi-5_4_4-manual.pdf

Refactoring tries to improve the software's interior construction, maintainability, and comprehensibility, without changing software's behavior/functionality.

There have been some usability experiments relevant to evaluating program comprehension tools [Storey 2005]. Bellay and Gall conducted a comparative evaluation of five reverse engineering tools using a case study and an evaluation framework [Bellay and Gall 1998].

2.3 Information Retrieval in Software Engineering

Within the area of software engineering, researchers have presented many IR methods in the last few decades. These methods are currently employed for many different goals, and they include traditional approaches such as signature and inversion [Faloutsos and Oard 1995, Maletic and Kagdi 2008]. Other methods try to filter and extract more information about documents to achieve better performance. Such methods include the use of parsers, syntactic information extracting, and Natural Language Processing techniques. Much of this work deals with natural language text and generally the techniques are intended for performing indexing, classification, and retrieval of text documents [Marcus et al. 2004, Binkley and Lawrie 2010, Binkley and Lawrie 2010]. Marcus and Maletic [Marcus and Maletic 2003] concluded that the use of IR methods in Software Engineering tasks is helpful, successful and productive.

The IR methods used to deal with source code production and to build a profile for each document (based on the granularity level chosen). The profile is defined as a summarized description or a new representation of the original document that is easier to control and work with. Users can decide which information to include in each document

profile, only meaningful information is typically extracted and integrated into the profile [Marcus et al. 2004, Dit et al. 2011].

It is very costly to build knowledge base for parsing approaches to extract semantic information from source code and related documentation. Using IR methods to extract these kinds of information has proved to be efficient, with the capacity to produce fine quality and low cost outcomes [Marcus et al. 2004].

In software programming, meaningful identifier names are generally selected by programmers. Furthermore, by using the comments, the ideal programmer always describes the source code with useful and meaningful information. Thus, source code contains important and significant domain knowledge that can be extracted and expressed [Maletic and Marcus 2000, Maletic and Marcus 2000, Marcus et al. 2004]. IR techniques have proved their effectiveness in expressing and discovering these types of information.

In Software Engineering, IR methods were used early in the context of indexing reusable software components and automatically constructing libraries [Maarek et al. 1991]. Nonetheless, in recent times IR methods have been used in solving the problems of software maintenance and development tasks such as traceability link recovery [Marcus and Maletic 2003], features and concept location [Poshyvanyk et al. 2006, Liu et al. 2007, Poshyvanyk et al. 2007, Poshyvanyk and Marcus 2007, Revelle et al. 2010], and source code clustering and summarization [Haiduc et al. 2010, Savage et al. 2010]. In [Poshyvanyk et al. 2006], IR techniques have been used to evaluate and assess the subsequent cost required to make modifications, and also to identify parts of a program in need of anticipatory maintenance tasks. Poshyvanyk and Marcus [Poshyvanyk and

Marcus 2007] employed these methods in the assignment of bug's fixing based on problem explanation reports. IR methods have also been utilized to find and capture coupling and cohesion of classes [Marcus and Poshyvanyk 2005, Marcus et al. 2008].

For the purpose of naming and detecting abstract data types in procedural code and to discover clones, Marcus and Maletic [Marcus and Maletic 2001], employed IR successfully and efficiently to achieve these goals. Wilde et al [Wilde et al. 1992], used IR methods to recommend an ordered list of professional developers to help in the completion and implementation of software change requests (e.g., bug reports and feature requests).

The Vector Space Model [Salton et al. 1975, Dit et al. 2011], Latent Semantic Indexing (LSI) [Marcus et al. 2004], and Latent Dirichlet Allocation (LDA) [Blei et al. 2003, Linstead et al. 2008, Tian et al. 2009] are examples of IR techniques that have been successfully applied in the context of Software Engineering [Marcus et al. 2004, Poshyvanyk et al. 2007, Binkley and Lawrie 2010, Dit et al. 2011].

Marcus and Maletic [Maletic and Marcus 2000] and Maletic and Valluri [Maletic and Valluri 1999] were the first researchers who investigated LSI's potential use in software maintenance. They utilized similarity measures between source code components in order to cluster and classify these components. Afterwards, Maletic and Marcus continued their work in [Maletic and Marcus 2001] to define a number of metrics for comprehension. These metrics use the profile produced by the application of LSI to the matrix of the source code. Marcus et al. [Marcus et al. 2004] linked LSI to concept location problem, where LSI was used to map the concepts that are expressed in natural

language change requests to relevant components in the source code. Poshyvanyk et al. [Denys et al. 2005] proposed a Visual Studio plug-in (IRiSS), based on an existing “find” feature that used LSI to search projects using natural language queries.

For more details on information-retrieval applications in software maintenance and evolution, readers referred to the survey by Binkley and Lawrie [Binkley and Lawrie 2010].

CHAPTER 3

Improving Feature Location by Enhancing Source Code with Stereotypes

This chapter presents a novel approach to improve feature location by enhancing the corpus (i.e., source code) with static information. An Information Retrieval method, namely Latent Semantic Indexing (LSI), is used herein for feature location.

When correcting a fault, adding a new feature, or adapting a system to conform to a new platform or API, software engineers must first find the relevant parts of the code that corresponds to the particular change. This is termed feature or concept location [Biggerstaff et al. 1994, Dit et al. 2011]. Feature location involves searching, exploring, reading, and understanding the source code. These types of comprehension activities make up a major portion of the costs in the evolution of modern software systems [Turver and Munro 1994, Binkley and Lawrie 2010].

A number of different techniques to support feature location have been suggested and involve approaches ranging from simple regular-expression matching to dynamic and static program analysis, and complex information-retrieval techniques. Regular-expression matching is often used by programmers but returns far too many false positives and has no ability to rank the results. Static and dynamic methods often suffer from the same types of problems [Eisenbarth et al. 2003, Dit et al. 2011] (too many false positives) or require very accurate test cases for the feature, which may not be available. Generally, the tools that deal with feature and concept location problem are mainly

classified into two categories, based on the way that such tools extract information from the source code; static and dynamic. Static (or interactive) approaches, collect their input without execution of the intended program, while in dynamic approaches, the input comes from investigating the execution traces or executing test cases [Wilde et al. 1992, Dit et al. 2011]. Neither category is optimal. The overlap between features cannot be distinguished in dynamic analysis, while static analyses often does not identify units contributing to particular execution scenario [Binkley and Lawrie 2010, Dit et al. 2011]. Both of dynamic and static methods are used as an input for hybrid approaches [Wilde et al. 1992]. Revelle and Poshyvanyk [Revelle and Poshyvanyk 2009] presented an investigative study of ten feature location techniques that used different combinations of textual, dynamic, and static analyses.

Over the past few years, IR methods have been used for feature location with encouraging results [Marcus et al. 2004, Poshyvanyk and Marcus 2007, Revelle et al. 2010, Dit et al. 2011]. IR methods move far beyond keyword matching and regular expressions and use advanced probability and information theory to derive relationships between documents based on the vocabulary and occurrences of words in each document. This is attractive because retrieval queries can be made in the language of the documents (i.e., programming language terms, identifiers, and natural language of comments). There are also means to rank the results from a query, much like the presentation of web search results.

While the use of IR methods has been successful for feature location, there is room for improvement. In particular, false positives are an issue and the most relevant

documents are not always ranked highly. This presents problems for software engineers using tools for feature location. Adoption is a problem because results are not good enough and searching through a long list of possible relevant documents is costly and time consuming.

To address this problem a number of researchers have applied and combined various static and dynamic analysis techniques to results from IR methods. For example, Formal Concept Analysis (FCA) has been used to help rank the results produced by IR methods, given the ranked list, the approach selects the most relevant attributes from the best ranked documents, clusters the results, and presents them as a concept lattice, generated using FCA [Poshyvanyk and Marcus 2007]. However, IR methods have been also used by researchers in a standard manner [Marcus et al. 2004, Poshyvanyk et al. 2005] for the problem of feature location.

In our approach, before applying the IR method, the corpus (i.e., source code) was enhanced through the addition of new information. This new information was derived automatically from the source code via static program analysis. Specifically, the source code was re-documented by adding stereotype [Dragan et al. 2006] information for each method/function in the system. After this was completed, the IR method was used to run queries for feature location. This type of up front enhancement of a corpus to improve results has not been investigated previously.

As mentioned earlier, augmenting source code with these new terms is a form of supervision added on top of an unsupervised method (i.e., LSI). The following are simple

examples that demonstrate the value of such added annotations. Suppose that there exist the following sentences:

1. Tom usually uses the plow and irrigation pipes when planting his land and his backyard with the tress and seeds. (Acting).
2. People buy compost and sterilizer when planting the trees and seeds in USA. (Acting).
3. Is the weather suitable to let our people clean the backyard and the land from party trees? (Predicting).

And there is also the following query that asks about the actions that are usually performed by a farmer when doing planting, Query:

“What do people use when farming the land by trees?”

Before adding the annotations that describe the main category of each sentence, the third sentence would be retrieved as the most relevant sentence, while clearly it is not relevant to the query at hand or the other sentences, thus, to the extent that the annotations represent accurate and useful associations between the sentences, adding the annotations (Acting and Predicting) increase the probability of retrieving similarly annotated category and sentences.

3.1 Approach Hypothesis

The hypothesis of the presented work is that the stereotype annotations are relevant and will improve the results in the context of feature location. The experimental study presented here, supports this hypothesis. The results demonstrate a significant

improvement in locating relevant methods pertaining to the feature being queried when stereotype information is included.

Stereotype information was chosen for a number of reasons. Stereotypes describe the abstract behavior and role of a method within a class. It was felt that this was relevant information and the previous work investigating the automatic detection of stereotypes [Dragan et al. 2006, Dragan et al. 2010], bears evidence that they support program comprehension. Moreover, it was found that distributions of method stereotypes could be used to derive class stereotypes. This evidence gave support to enhancing the information within the source code. Lastly, stereotype information is new information that did not previously exist in the source code (i.e., new vocabulary).

3.2 Related Work

The main research interest in this chapter is focused on feature location. Therefore, an overview of existing static feature location approaches is reviewed along with related work on feature location using LSI.

3.2.1 Previous Work on Feature Location

Historically, developers used the pattern matching techniques like *grep* to locate the features in the source code. Using pattern-matching techniques is simple; it performs an investigating through pattern matching on character strings. Nevertheless, it requires a lot of experience from the developer. If the technique failed, more advanced tools were required, especially when the system is large [Marcus et al. 2004, Poshyvanyk et al. 2007, Poshyvanyk 2009, Binkley and Lawrie 2010].

Biggerstaff et al. [Biggerstaff et al. 1994] referred to concept location as the concept location assignment problem. Their work was a preliminary point for a lot of efforts to facilitate and develop the process of concept location. Call graphs, program clustering graphs, etc. are used in their approach. Chen and Rajlich [Chen and Rajlich 2000] presented an approach based on looking through an Abstract System Dependencies Graph (ASDG). The ASDG can lead, guide, and help the users in the process of searching for a particular feature.

Wilde [Wilde and Scully 1995] developed the software-reconnaissance method , which utilizes dynamic information to locate features in existing systems. Wong et al. [Wong et al. 2000] analyzed the execution slices of test cases to the same end. Eisenbarth et al. [Eisenbarth et al. 2003] used dynamic information gathered from scenarios of invoking features in a system to locate the features in source code. The tools that deal with feature location are either static or dynamic. Overlap between features cannot be distinguished using dynamic analysis, while static analyses do not often identify units contributing to a particular execution scenario [Binkley and Lawrie 2010, Dit et al. 2011]. Revelle and Poshyvanyk [Revelle and Poshyvanyk 2009] presented an investigative study of ten feature location techniques that use different combinations of textual, dynamic, and static analyses. A survey of feature location techniques is presented in [Dit et al. 2011].

3.2.2 Previous Work on Feature Location Using IR

Recently, IR methods have been used successfully and effectively for feature location [Marcus et al. 2004, Poshyvanyk et al. 2005, Poshyvanyk et al. 2006, Poshyvanyk and Marcus 2007, Revelle et al. 2010, Mahmoud and Niu 2011, McMillan et

al. 2011]. For more details, we refer the readers to the survey by Binkley and Lawrie [Binkley and Lawrie 2010] about IR applications in software maintenance and evolution.

Marcus and Maletic [Maletic and Marcus 2001], were the first researchers to use LSI for applications to Software Engineering. They obtained similarity measures between source-code components in order to cluster and classify these components. And they define a number of metrics for comprehension. These metrics use the profile produced by the application of LSI to the matrix of source code. In [Marcus et al. 2004], Marcus et al., linked LSI to concept location, where they used LSI to map concepts expressed in change request that is described using natural language to the relevant components of the source code.

Many efforts have been presented to improve the use of LSI in feature location, by adding meaningful information to the whole process [Poshyvanyk and Marcus 2007]. For example, in [Liu et al. 2007], the authors combined LSI with user execution scenarios to improve the accuracy of feature location. Poshyvanyk et al. [Poshyvanyk et al. 2005], proposed a Visual Studio plugin called IRiSS, which based on the existing “find” feature uses LSI to search projects using natural-language queries. In [Poshyvanyk et al. 2006], Poshyvanyk et al., combined static and dynamic techniques they had developed before. They applied them individually to identify concepts and features in the source code in order to improve the accuracy of feature location and decrease the time needed to catch the first relevant method.

Poshyvanyk and Marcus [Poshyvanyk and Marcus 2007] proposed an approach that combines formal concept analysis (FCA) and latent semantic indexing (LSI). The

approach is evaluated in a case study on concept location in the source code of Eclipse. Their results showed that FCA is successful in terms of managing different concepts and in reducing the effort that the developers need.

In [Poshyvanyk et al. 2007], the authors, in order to improve the accuracy of feature location process, proposed a technique that combines information from an execution trace and from the comments and identifiers that extracted from the source code. M. Reville et al. [Reville et al. 2010] applied an advanced web mining algorithms (*Hyperlinked-Induced Topic Search* (HITS) and *PageRank*) to analyze the execution information during feature location. Their approach improved the effectiveness of existing approaches by as much as 62%. The ability of LSI in providing a straightforward language-independent method that recognizes relationships between documents is shown in SNIAFL [Zhao et al. 2004].

The dimensions of Singular Value Decomposition (SVD) when using LSI have been studied. The range of 200 to 300 dimensions has been proposed as a “golden standard” [Marcus et al. 2004]. In [Poshyvanyk et al. 2006], Poshyvanyk et al., looked at varying the number of dimensions when using LSI and compared the results. Their findings supposed that any larger factor could improve the results but it would generate too large a search space. Generally, the current approaches either use IR methods alone or in combination with other techniques, such as [Poshyvanyk et al. 2006, Poshyvanyk et al. 2007]. There is a need for improvements in recall and precision of feature location. None of these approaches augment the source code with new information. In our approach, the source code is augmented with method stereotypes, which is described next.

3.3 Method Stereotypes

Source code stereotypes are a type of source code annotation. They are abstraction of the basic behavior of a method or class. The programmer usually uses annotations mainly for source code documentation and comment expanding. Moreover, the behaviors of classes, methods, variables, parameters and packages can be annotated briefly. More details about the annotation we use (stereotypes) are presented in the following subsections.

3.3.1 Stereotypes Definition

Stereotypes are a concise abstraction of a method's role and responsibility in a class and system [Dragan et al. 2006]. They are widely used to informally describe methods. Stereotypes for classes are also used in the same manner to describe their role and responsibility within a system's design. Unified Modeling Language (UML) provides mechanisms for documenting class stereotypes.

Manually documenting method stereotypes is relatively easy for a small number of classes and methods however it is quite costly to do so for an entire system.

3.3.2 Method Stereotypes Taxonomy

A taxonomy of method stereotypes (see Table 3-1) and technique to automatically reverse engineer stereotypes for existing methods was presented by Dragan et al. in [Dragan et al. 2006].

Table 3-1. Taxonomy of method stereotypes as given in [Dragan et al. 2006]. The taxonomy is mainly focused on the C++ programming language. Methods may be labeled with one or more stereotypes.

Stereotype Category	Stereotype	Description
Structural Accessor	Get	Returns a data member.
	Predicate	Returns Boolean value which is not a data member.
	Property	Returns information about data members.
	void-accessor	Returns information through a parameter.
Structural Mutator	Set	Sets a data member.
	Command	Performs a complex change to the object's state.
	non-void-command	
Creational	constructor, copy-const, destructor, factory	Creates and/or destroys objects.
Collaborational	collaborator	Works with objects (parameter, local variable and return object).
	Controller	Changes an external object's state (not <i>this</i>).
Degenerate	Incidental	Does not read/change the object's state.
	Empty	Has no statements.

The taxonomy of method stereotypes given in Table 3-1 unifies and extends previous literature on stereotypes and addresses a number of gaps and deficiencies that were present. The taxonomy was developed primarily for C++ but many aspects of it can be applied to other programming languages. Based on this taxonomy, static program analysis is used to determine the stereotype for each method in an existing system.

As shown in the above table, the taxonomy is organized by the main role of a method while at the same time highlighting its creational, structural, and collaborational aspects with respect to a class's design as follow:

- Structural methods: support class structure. For example, accessors read an object's state, while mutators change it.
- Creational methods: create or destroy objects of the class. For example, constructor and destructor.
- Collaborational methods: describe the communication between objects (how objects are controlled in the system).
- Degenerate methods: are methods where the structural or collaborational stereotypes are limited.

The naming is based on the mathematical term for a case for which a stereotype cannot be any simpler. Also, a method may have more than one stereotype. This work was further extended to support the automatic identification of class stereotypes in [Dragan et al. 2010]. That work describes an approach to automatically identify method stereotypes that we use in this research. We refer the readers to those works for complete details on computing method stereotypes; however we present the main points here.

A tool [Dragan et al. 2006], *StereoCode*, was developed that analyzes and re-documents C++ source code with the stereotype information for each method. Re-documenting the source code is based on srcML (Source Code Markup Language) [Collard et al. 2011], an XML representation of source code that supports easy static analysis of the code.

In order to provide the method-stereotype identification, we translate the source code into srcML, and then, *StereoCode* takes over by leveraging XPath, an XML standard for addressing locations in XML. For details about the rules for identifying each method stereotype, we refer the readers to [Dragan et al. 2006]. Adding the comments (annotations) to source code is quite efficient in the context of srcML.

The XPath query gives us a location of the method and we can then do a simple transformation within the srcML document to add the necessary comments. This process is fully automated and very efficient/scalable. Running *StereoCode* on two systems used in the evaluation takes less than a minute each. Methods can be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have secondary stereotypes from the collaborational and degenerate categories. For example, a two-stereotype method `get-collaborator` returns a data member that is an object or uses an object as a parameter or a local variable.

Figure 3-1 presents an example of stereotype labeling for part of the class `DataSource` from the HippoDraw open source application (one of the systems used in the experiment). The class `DataSource` supplies one or more arrays of data. The evaluation of the taxonomy and approach demonstrated two things. First, the method-stereotype

taxonomy covered a very large percentage of the methods studied. That is, almost all methods can be labeled by the classification scheme. Second, the tool re-documented the systems according to the taxonomy with a very high accuracy in comparison to human evaluation.

```
class DataSource :public Observable
{
private:
    string m_ds_name;
    vector<string> m_labels;
    bool m_is_null;
protected:
    mutable vector<double> m_array;
    int m_rows;
public:
    /** @stereotype get */
    bool isNull() const;
    /** @stereotype predicate */
    bool isValidLabel(const string& label) const;
    /** @stereotype property */
    virtual double sum(int column) const;
    virtual int indexOfMinElement(int index)const;
    /** @stereotype set */
    void setLabels(const vector<string>& v);
    /** @stereotype command */
    virtual void reserve(int count );
};
```

Figure 3-1. A code snippet of the HippoDraw C++ Class DataSource after re-documenting with the method stereotypes.

3.4 Latent Semantic Indexing (LSI)

The LSI is a corpus-based statistical technique which is used for inducing and representing characteristics of the meanings of words and passages (of natural language) reflective in their usage [Deerwester et al. 1990, Marcus et al. 2004].

LSI method produced existent valued vector information for text documents. However, this valued vector can be employed efficiently to perform comparing and indexing for any text documents by using the similarity measures, in other words, it uses the similarity measures to compute the similarity between source code components. Moreover, the similarity is used to define the direct and indirect (hidden) relationships between components. Therefore, applying LSI to source code and its components (internal documentation i.e., comments) can allow the components to be compared and be investigated semantically and structurally. In literature, the results have shown [Berry 1992, Landauer and Dumais 1997] that LSI can define a significant quantity of the meaning of individual words and whole passages such as sentences or paragraphs in the text. The fundamental concept of LSI is that the information about word contexts in which a specific word appears or does not appear, provides a set of common restrictions so as to define and find the similarity between bags of word.

Theoretically, LSI relies on a Single Value Decomposition (SVD) [Deerwester et al. 1990] of a matrix (word \times context) derived from a corpus of natural text in the particular domain of interest, see Figure 3-2. SVD is a form of subject analysis and acts as a method for decreasing the vectors dimensionality of a feature space without any serious loss of specificity.

The number of dimensions can be reduced by using SVD without huge loss of descriptiveness. SVD is the underlying operation in a number of applications including statistical principal component analysis [Jolliffe 1986], text retrieval [Binkley and Lawrie 2010], pattern recognition and dimensionality reduction [Dit et al. 2011], and natural language understanding [Landauer and Dumais 1997]. For complete details of Latent Semantic Indexing see [Deerwester et al.].

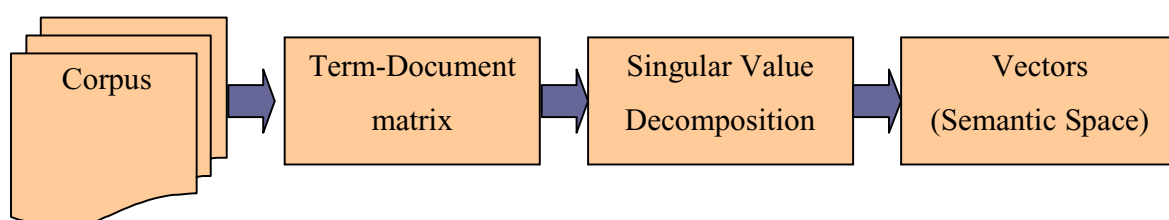


Figure 3-2. LSI Steps: The corpus is represented as a term-document matrix (term x document), then the matrix is then subject to SVD, computes the term and document vector spaces.

The resulting profile is that each word is represented as a vector in a d -dimensional space. The results mainly depend on the number of dimensions that are taken. As mentioned in [Marcus et al. 2004], the optimal number is usually around between 100 and 300 (golden set) and may differ from corpus to corpus, and from domain to domain. For more details, readers are referred to [Poshyvanyk et al. 2006].

The similarity of any two words, any two text passages, or any word and any text passage, are computed by measuring the similarity between their vectors. Often the cosine of the contained angle between the vectors in d -space is used to determine this similarity, and the length of vectors is also useful as a measure.

One of the criticisms of LSI method, when applied to natural language texts is that it does not make use of morphology, word order, or syntactic relations. Nevertheless,

very good results are derived from corpus without including this information [Marcus et al. 2004].

This characteristic is very well appropriate to the domain of source code. For the reason that much of informal concepts of the problem/task, are embodied in the names of operands and in the operators keys that assigned by the programmers in source code implementation. Moreover, word ordering has slight meaning.

3.4.1 Why LSI?

A major shortcoming of a number of IR methods is that they fail to treat synonymy and polysemy correctly. Synonymy is a term used to describe when there are many ways to refer to the same object. That is, developers in different contexts, with different domain knowledge, or linguistic behaviors will explain and describe the same information using different terms and different styles or symbols.

Polysemy refers to words that have more than one distinct meaning. LSI attempts to overcome this shortcoming by choosing linear combinations of terms as dimensions of the representation space. LSI explicitly represents terms and documents in high-dimensional space, which allow the searchers by using querying to discover and define the underlying semantic relationships between terms and documents.

As a conclusion, using LSI for extracting semantic similarity of source code documents provides precious information that can be used by the developers in the tasks of software maintenance and evolution. Moreover, it shows that concepts/features from the problem domain are often spread over multiple files, and files contain multiple concepts or features.

Among code-based feature-location techniques, LSI is considered one of the better techniques capable of recognizing terms in source code that are relevant to a user query [Binkley and Lawrie 2010]. Moreover, LSI is language independent and using it to preprocess and search the source code is more efficient than using a pattern-matching technique, especially as mentioned before, its capability in dealing with synonymy and polysemy. It is also simpler than using graph-based techniques [Binkley and Lawrie 2010].

3.4.2 LSI Processing Steps

The initial step of the IR process is to build the corpus for the software system. The corpus consists of a set of documents. In this work and in most all feature location works, documents in the corpus are methods or functions. These documents include the text of each method including all the identifier names, comments, etc.

3.4.2.1 Corpus Creation

Constructing the corpus is an important step for feature location using LSI. Five actions are taken to create the corpus:

1. Extraction of *identifiers*, and *comments*.
2. Extraction of *method stereotypes*.
3. *Identifier* (term) separations.
4. Removing *stop words*.
5. Divide into *documents* (method level).

A well-built corpus helps in locating the relevant methods (effectiveness measure). As mentioned in [Revelle et al. 2010], not all feature-location techniques can

locate all feature-relevant methods, One cause of failure is the preprocessing steps taken when enriching the corpus.

The approach proposed here uses srcML [Collard et al. 2011] to transform the C++ source code to XML format as a first preprocessing step. srcML is an XML representation that supports both document and data views of source code. The format supports lightweight static program analysis using standard XML tools, while at the same time preserving all original lexical information. A very usable and efficient tool to translate C/C++ to/from srcML is freely available⁴.

We developed an efficient corpus builder in C++ to extract these important elements from source code that in XML format. It takes less than 30 seconds to build both the corpus (corpora for the two systems we used in the experiments) with stereotypes and the corpus without stereotypes.

Names such as identifiers, function name, etc. are split according to the standard separators [Marcus et al. 2004, Revelle and Poshyanyk 2009]. An underscore, ‘_’, is used as a separator to split identifiers that contain more than one word, e.g., *feature_location* after splitting becomes *feature*, *location*, and *feature_location*. Camel casing is also used as a separator, e.g., *FeatureLocation* is split into *Feature*, *Location*, and *FeatureLocation*, and *FEATURELocation* is split into *FEATURE*, *Location*, and *FEATURELocation*.

⁴ See www.sdml.info for srcML downloads and documentation.

The final step of preprocessing is partitioning the code into documents. Each function is considered to be a separate document (i.e., level of granularity). Typically, a document in the corpus can be a file of source code or a program entity such as a class, function, interface, etc. When the preprocessing is completed the software system is represented by a set of documents, $S = \{d_1, d_2, \dots, d_n\}$, where d_i is any contiguous set of lines of source code and/or text.

Each document d_i contains the function name, identifiers that the function uses, internal comments, string literals, and the stereotype annotation for each the function. After these steps, the corpus is constructed.

3.4.2.2 Indexing

The next step is to index the corpus using LSI. After creating the LSI space (using SVD), each document d_i in system S will have a corresponding vector v_i . Reduction of dimensionality is done in this step and reflects the most important latent aspects of the corpus. The dimension of the vector is a parameter of the algorithm. It is normally between 100 and 300 [Marcus et al. 2004]. The typical manner to choose this value is to run experiments with different values (e.g., 100, 200, 300) and select the one that gives the best results with respect to evaluation measures as shown later [Marcus et al. 2004].

Measuring the similarities between any two documents $sim(d_i, d_j)$, can be done by measuring the similarities between their correspondents vectors. Here cosine similarities are used. By studying and analyzing these similarities, we can identify the semantic information regarding source code fragments, and the relations connecting them.

3.4.2.3 Queries Formulating and Documents Ranking

The user formulates a query by using natural language to describe a change request in the same manner as [Liu et al. 2007]. A user query (q) is converted into a document of LSI space (dc) and vector (vq) for it is constructed. Based on the similarity measure between vq and all documents in the corpus, the most relevant documents to vq are retrieved ranked list $\{P_1, P_2, \dots, P_n\}$.

Once LSI retrieves the relevant documents ranked by their similarities to the user query, then the user has the task of inspecting and investigating these documents to decide which of them are actually relevant to the query. The first ranked document (P_1) will be investigated first and then (P_2) and so on. The user decides when to stop investigating. If the user discovers a part of the feature, then the intended feature is located successfully. Otherwise, the user can reformulate the query taking into account these results.

At this point, the specialist developer with comprehensive understanding of the interested system should be the one who formulates the queries. In [Marcus et al. 2004], the authors exceeded this point by supporting a user query that is based on partially automated generated queries.

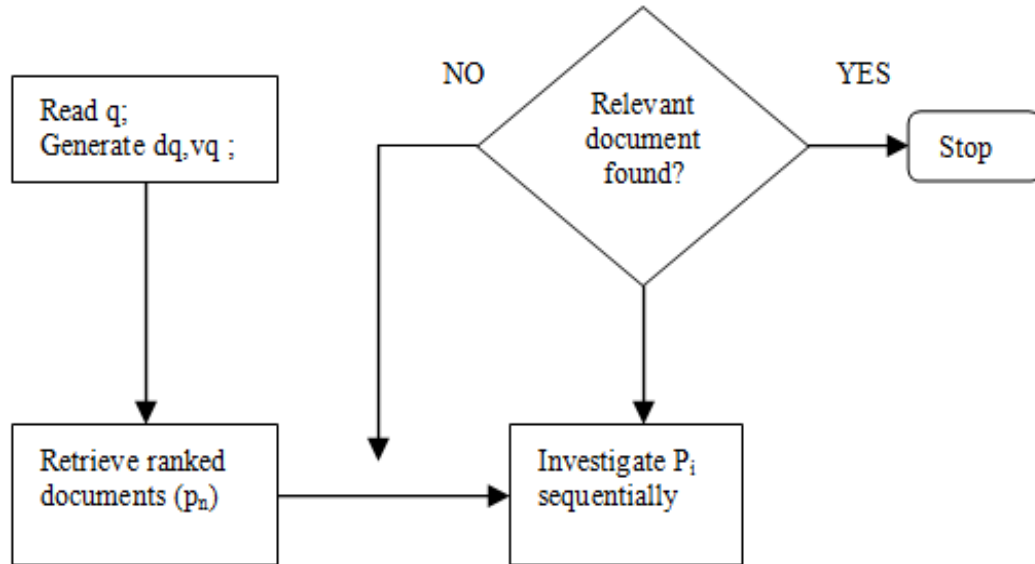


Figure 3-3. Retrieving the results for a query (q).

3.5 LSI+Stereotypes for Feature Location

Described in this section is the approach taken for feature location. The same approach as the one utilized in [Marcus et al. 2004] is used here. The IR method LSI [Deerwester et al. 1990, Binkley and Lawrie 2010], is the basis of the approach. Figure 3-4 presents an overview of the entire process. We term our approach LSI+S (LSI plus stereotypes) to differentiate it using with LSI without stereotypes.

The start is with the source code for a software system. As described in the previous section, the *StereoCode* tool is applied to automatically determine the stereotype of each method and re-document it with a comment stating its stereotype. Next preprocessing is done to the resultant re-documented source code to convert it into input for LSI. This is termed a corpus. It was described before how the corpus is generated.

At this point LSI is applied to the corpus. A co-occurrence matrix of vocabulary \times documents is computed and SVD [Salton and McGill 1983] is applied to reduce the dimensionality of this matrix by exploiting the co-occurrence of related terms. More details are in the next sub-section.

The result is a subspace that can be queried against to locate documents most similar to the query terms. Ranked documents will be retrieved based on their similarities to the query. The user then inspects the results. More details about these steps are covered separately on the following sub-sections.

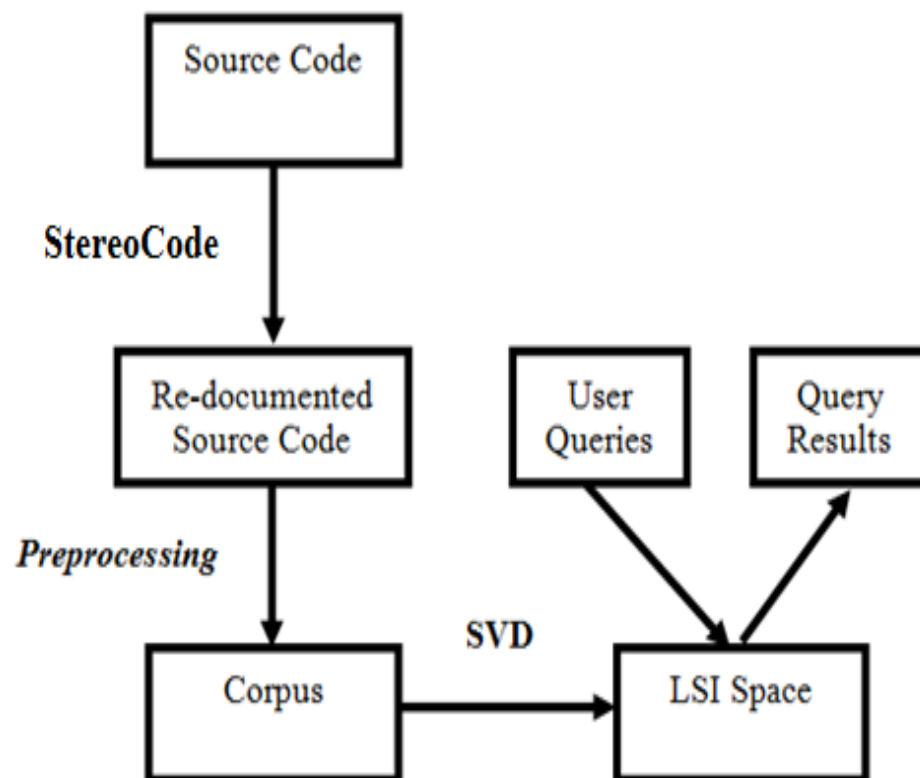


Figure 3-4. The feature location process used in this study. First, stereotypes are computed and added as comments in the source code. Next preprocessing is done to produce a corpus as input to Latent Semantic Indexing (LSI). LSI produces a vectorized representation of the corpus that queries can be made against.

3.6 Experimental Study

A feature-identification study, over two open source software systems was conducted to evaluate and compare the results of LSI and LSI+S. The study is designed based on recommendations from [Yin 2009]. Both techniques, LSI and LSI+S, are applied independently and then the results compared.

The only difference between the techniques is the inclusion of the stereotype information in LSI+S. Otherwise, the parameters used and the construction of the corpus is exactly the same. One large and one medium-size open-source system were selected to demonstrate the scalability/practicality of the proposed approach.

3.6.1 Design and Objective of the Experimental Study

The first system is HippoDraw⁵, an open-source application written in C++ that provides a data-analysis environment. It includes data-analysis processing and visualization with an application GUI interface, and can be used as a stand-alone application or as a python extension module.

HippoDraw source code is well written and follows a pretty consistent object-oriented style. Its library consists of approximately 50K LOC and over 300 classes. HippoDraw 1.21.3 release is used in our study since it's well documented.

⁵ See <http://www.slac.stanford.edu/grp/ek/hippodraw/> for more information on HippoDraw

The second system used is the open source cross-platform application and UI framework Qt⁶. It has extensive international support, as developers from Nokia, Digia, and other companies are involved in Qt's development. Qt is mainly written in C++ but has some language extensions with a special code generator (called the Meta Object Compiler) and special macros. It is cross platform for Windows, Linux, or Mac, and all of its editions support a wide range of compilers (e.g., gnu gcc, and MS Visual Studio). The Qt 4.4.3 release is used in our study. The major purpose of this particular release is to supply bug fixes and performance developments based on both internal testing and client feedback.

Table 3-2 describes the characteristics of HippoDraw and Qt in the context of their use for LSI. It is clear that Qt is a much larger system in all aspects. Both of LSI and LSI+S are applied separately to each system. This allows for comparing the results and assessing their quality relative to each other for the context of the added stereotype information. The method level of granularity is chosen in both studies. The same methodology which described in section 3.4 was used for ranking the relevant parts of source code with respect to user query, with different dimensionality reduction factors chosen for each study.

⁶ See <http://qt.nokia.com/products/> for more information on Qt.

Table 3-2. Details of the corpus used as input to LSI for each of the two systems used in the experimental study.

	HippoDraw 1.21.3	Qt 4.4.3
Vocabulary Size	6,803	91,187
Number of Parsed Documents/Methods	3,706	70,871
Dimensionality Used	200	300

3.6.2 Evaluation Measures

To evaluate the results of feature location, a number of studies [Poshyvanyk et al. 2006, Poshyvanyk et al. 2007, Revelle et al. 2010], use the position of first relevant method as an effort measure. Other studies [McMillan et al. 2011] use recall and precision measures. Additionally, computed is the total effort measurement and then the position of the last relevant method is used. All of these measures as well as p-value are used to evaluate the results of LSI and LSI+S approaches.

The standard IR measurements [Binkley and Lawrie 2010] *recall* and *precision* are used. Recall of 100% means that all the relevant documents are recovered, though there could be recovered documents that are not correct. Precision of 100% means that all the recovered documents are correct, though there could be correct documents that were not recovered.

Typically there is a tradeoff between precision and recall. If there is high recall, then precision normally is low. If there is high precision, then recall normally is low. In computing recall and precision we only include the first 100 ranked items retrieved for

the query. This is a standard approach to computing these values as anything more than 100 is beyond what a developer would normally investigate. Recall and precision are defined as follows:

- $\text{Recall} = |\text{relevant} \cap \text{retrieved}| \div |\text{relevant}|$
- $\text{Precision} = |\text{relevant} \cap \text{retrieved}| \div |\text{retrieved}|$

The main goal of all feature-location techniques is to reduce the effort of the developers in the location process. Therefore, in this evaluation we measure the effort that the developers need (maintenance-effort measurements) as the number of methods from the retrieved ranked list that they have to investigate until finding the first *relevant* method (PFR), the last *relevant* method (PLR), and all *relevant* methods ($\sum \text{EM}$) [Binkley and Lawrie 2010].

Typically, with respect to the maintenance effort measurements, lower values are preferred. These measures are defined as follows:

- $\sum \text{EM}$: Total Effort Measurement (number of methods we need to investigate to find *all* relevant documents).
- PFR: Position of first relevant document.
- PLR: Position of last relevant document.

For LSI and LSI+S, we compared the relevant documents rank side by side and we count the number of cases where LSI+S technique produces better ranks than LSI and vice-versa.

The Wilcoxon signed-rank test was used to examine whether the difference in terms of effectiveness for two approaches is statistically significant by computing the p-

value. Wilcoxon signed-rank test (One-Tail) is non-parametric test and it takes as an input two lists of ranks created from the two different feature location techniques, we assume that ranks implicitly contain the total efforts needed by developers when performing any maintenance activity. In our test, the significance level $\alpha = 0.05$ was designated, and the output of the test is a p-value, which can be understood as follows. If the p-value is less than α , then the difference in ranks produced by one feature location technique is statistically significantly lower than the ranks produced by the other technique. Otherwise, if the p-value is larger than α , then both of the two studied feature location techniques generate almost equivalent results.

The following are the null and alternative hypothesis that were formulated in order to test whether LSI+S has a higher effectiveness measure than LSI or not.

H_0 : *There is no statistical significant difference in the measure of effectiveness between LSI and LSI+S.*

H_1 : *LSI+S implied higher effectiveness than LSI.*

3.6.3 Experiments Feature Selection and Determining Relevant Methods

For the experiments, test features were selected for each system (see Table 3-3 and Table 3-5). The features were selected based on the bug reports present in the online system documentation for both HippoDraw and Qt. Compared to other studies on feature location [Liu et al. 2007] this choice represents a bit more rigorous set (i.e., previous studies have used as few as three queries), and some other studies use more. These 22 features were chosen because they were the most frequently changed based on the system documentation.

Both systems have extensive and very complete documentation. Developers maintain very detailed bug reports and descriptions of the modification to fix each. The set of relevant methods were manually determined for each feature using this documentation as described below.

For each feature the related bug reports and descriptions of the fixes were examined. Afterwards, all the methods were included which were modified in response to the bug fix. Two graduate students conducted a manual inspection of the code to determine all other methods relevant to that feature. We used systems websites, bug tracking reports, source code, etc. This collected data was then examined and any differences were resolved by additional inspection. This process took approximately 20 person/hours for HippoDraw and approximately 40 person/hours in the case of Qt, the difference here is due to the complexity and size of Qt.

3.6.4 Locating Features in HippoDraw System

For version 1.21.3 of HippoDraw the experiment was tested on the 11 features and queries described in Table 3-3. For the corpus that was re-documented, the stereotypes of relevant methods were inspected. It was found that all of the relevant methods for all features were labeled with at least one stereotype. That is, no relevant method was unclassified, which is a possible result from the re-documentation process. For overall distributions and details of the specific stereotyping of the HippoDraw system we refer the readers to Dragan et al. work [Dragan et al. 2006].

In order to examine the best user query that describes the intended feature accurately and completely, other researchers have used the process of formulating four

different user queries and then choosing the best one among them [Liu et al. 2007]. The same procedure is followed here. For each feature in Table 3-3, the given query, gives the best results of the four queries that were investigated. That is, the chosen query ranked the relevant documents more correctly than the other three queries for LSI and LSI+S.

Table 3-3 also presents the number of relevant documents for each feature. With respect to dimensionality reduction, the value of 200 was determined as the best value using the previously described method.

Table 3-4 summarizes the results obtained in identifying the features in the HippoDraw study. The first column indicates the feature number (from Table 3-3), the 2nd indicates the total effort measure, and the 3rd and the 4th columns indicate the positions of first and last relevant documents in the corpus respectively. As can be seen in Table 3-4, using stereotypes (LSI+S) improved all three measures comparing with the result of using no stereotypes (LSI).

The first relevant method (PFR) for LSI+S is equal or better to LSI. The precision and recall results are shown in Figure 3-5 and Figure 3-6, respectively. These figures show that LSI+S improves both recall and precision compared to LSI alone for most features. Specifically, the recall and precision are improved for 9 features using LSI+S, while for 2 features the recall and precision are equal using both approaches.

Table 3-3. HippoDraw Feature description, applied query, and the number of relevant methods for each feature.

Feature	Query	Number of Relevant Methods
1. change font size	<i>change font size weight set</i>	10
2. change font style	<i>change font style italic</i>	18
3. update zoom mode	<i>update zoom mode zoomin zoomout</i>	9
4. reset printer settings	<i>reset change printer settings</i>	8
5. add item	<i>insert add item canvas</i>	7
6. remove item	<i>Delete remove item canvas</i>	7
7. change mouse property	<i>Option change mouse property</i>	9
8. change cut color	<i>change cut color set</i>	7
9. change representation color	<i>change representation color set</i>	7
10. make new display	<i>make new display add make</i>	12
11. update axis modeling	<i>update axis modeling reset</i>	8

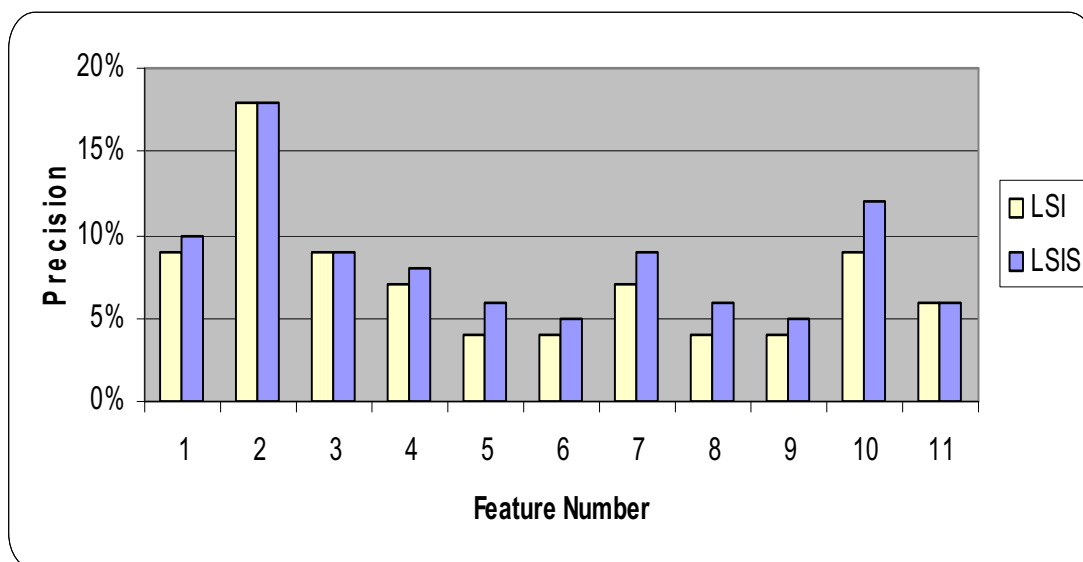


Figure 3-5. Precision results for the HippoDraw case study show that LSI+S (blue) had an equal or higher precision than LSI (yellow) alone.

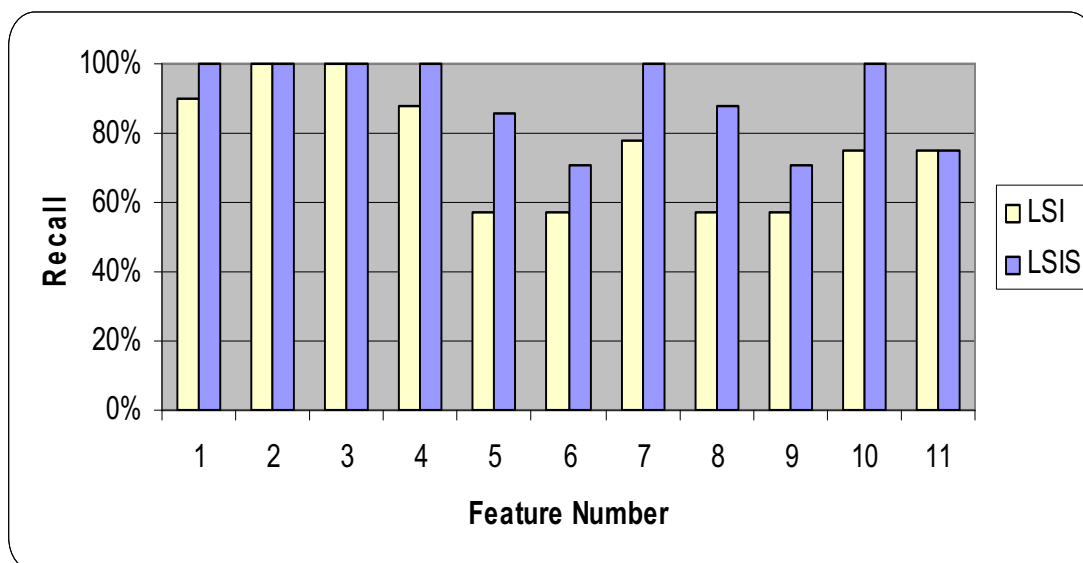


Figure 3-6. Recall results for the HippoDraw case study show that LSI+S (blue) had an equal or higher recall than LSI (yellow) alone.

Table 3-4. Result of HippoDraw system for three measurements; Total effort measurement (Σ EM), Position of first relevant document (PFR), and Position of last relevant document (PLR).

Feature	Total Effort Measurement(Σ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	208	103	8	1	109	32
2	466	362	9	3	70	54
3	172	98	6	1	36	22
4	328	231	3	2	210	100
5	455	339	1	1	216	183
6	648	484	12	10	238	138
7	834	544	4	1	121	67
8	1595	764	2	2	1290	534
9	602	471	1	1	250	174
10	503	387	2	1	125	94
11	1721	843	3	1	1200	388

3.6.5 Locating Features in Qt System

For version 4.4.3 of Qt the experiment was run on the 11 features and queries described in Table 3-5. The same steps taken on the first system were also done here. Again, four different queries were chosen, and then the best one among them was chosen. Experiments with different dimensionality reduction values showed that 300 gave the best results.

Table 3-5 presents the summarization for all investigated features and the best queries used to locate these features. Table 3-6 summarizes the results obtained in identifying the features in the Qt study. As can be seen LSI+S results in better values for all three measures compared with LSI alone. For this study, the precision and recall results are also shown in Figure 3-7 and Figure 3-8 respectively. Again, LSI+S improves recall and precision.

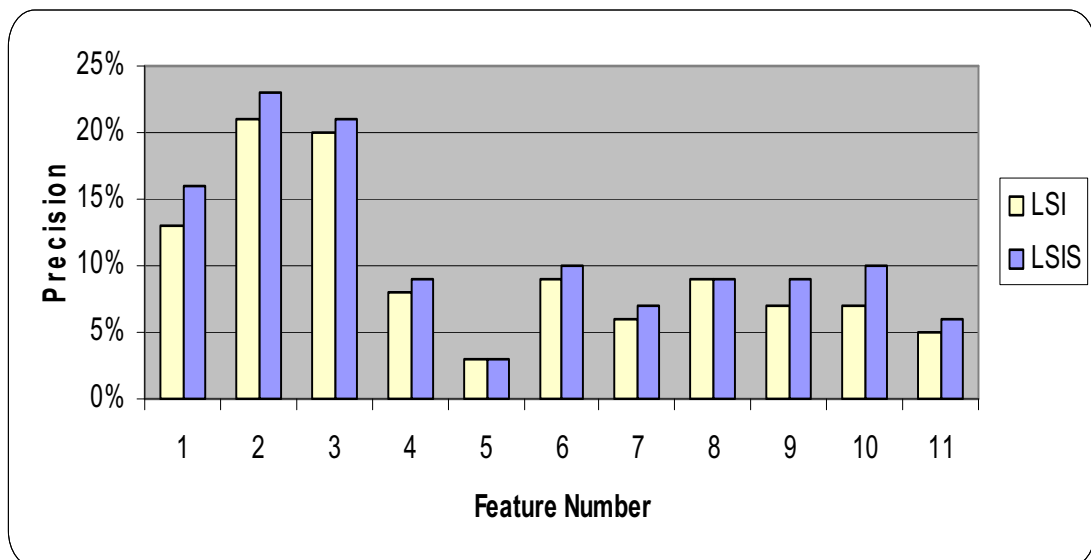


Figure 3-7. Precision results for the Qt case study show that LSI+S had better precision than LSI in almost all cases.

Table 3-5. Qt Features descriptions; feature name, query used, and number of relevant methods to each feature.

Feature	Query	Number of Relevant Methods
1. update font settings	<i>font update options settings reset</i>	21
2. create new font	<i>create new font</i>	24
3. change font size	<i>size font change</i>	23
4. set password	<i>set password change</i>	12
5. set RGB	<i>update RGB color RGBA RGBF</i>	7
6. add menu	<i>add create new menu insert menubar</i>	15
7. remove menu	<i>menu remove delete</i>	7
8. add action	<i>insert action add new</i>	11
9. remove action	<i>action delete remove</i>	9
10. search	<i>index search searching searcher indexing find</i>	12
11. draw polygon	<i>points polygon draw lines polyline</i>	7

Table 3-6. Result of Qt system for three measurements; Total effort measurement (EM), Position of first relevant document (PFR), and Position of last relevant document (PLR).

Feature	Total Effort Measurement (Σ EM)		First Relevant Document (PFR)		Last Relevant Document (PLR)	
	LSI	LSI+S	LSI	LSI+S	LSI	LSI+S
1	2208	1846	2	1	1054	332
2	1900	928	1	1	520	467
3	1668	1192	4	1	684	443
4	1760	996	4	1	710	359
5	112	100	19	8	59	40
6	2792	1667	2	1	830	451
7	251	149	1	1	101	94
8	1239	701	3	1	815	456
9	359	185	1	1	153	100
10	1078	599	2	1	184	150
11	1641	566	1	1	1321	450

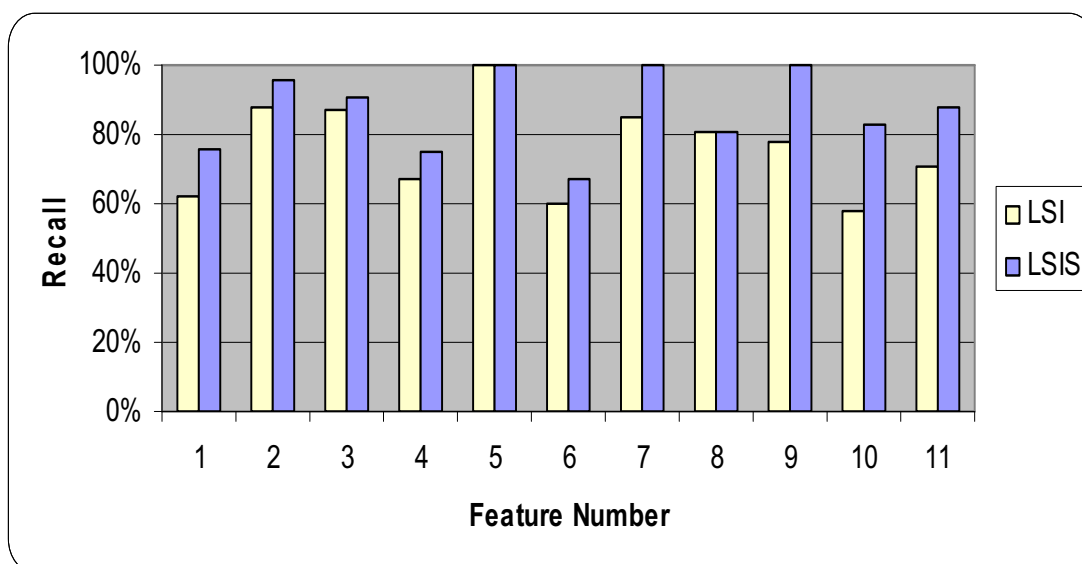


Figure 3-8. Recall results for the Qt case study show that LSI+S had better recall than LSI in almost all cases.

3.7 Discussion

The hypothesis tested was that adding stereotype information to the corpus (source code) would improve the results of LSI in the context of the feature-location problem. It is quite clear from the data that the addition of the stereotype information does improve the results of feature location using LSI for the presented queries in the context of these two systems. In all cases, and for all measures, LSI+S has equal or better values.

When examining the results of the studies, given in Table 3-4 and Table 3-6, it can be noticed that the position of the first relevant method improved with LSI+S in approximately 75% of the queries. The remaining 25% produced the same value.

Moreover, the position of the first relevant method for LSI+S is in the first position in 7 of the 11 features for HippoDraw and 10 of the 11 features in Qt. Using LSI

alone produced first positions of 2 of 11 for HippoDraw and 4 of 11 for Qt. This is a particularly nice improvement in the context of usability for the developer. They need not look far into the list for something relevant to their query.

Furthermore, the position of the last relevant method has been improved for all features in all cases with LSI+S. The improvement in this measure is also much more drastic (approximately one half on average). In Table 3-7 summarized is the difference between the first and last relevant method positions for the two approaches for HippoDraw and Qt respectively. Obviously there is an average improvement of 43% for HippoDraw and 36% for Qt in the distance from the first relevant method to the last relevant method.

The total effort measure is examined in Table 3-4 and Table 3-6. LSI+S again has better values for all queries. The average improvement is 46% with a range of 11% to 66% for both HippoDraw and Qt. From a usability standpoint this means that a developer would need to wade through far fewer methods on average to find all relevant methods.

With respect to the standard IR evaluation measurements (recall and precision), as described before, there is a tradeoff. The tradeoff depends on the list size used for ranked documents [Binkley and Lawrie 2010]. Likewise, we take the top 100 ranked methods.

The results for recall and precision for both studies are shown in Figure 3-5, Figure 3-6, Figure 3-7, and Figure 3-8. For both systems LSI+S has equal or better precision and recall values. Other studies that have used LSI alone [Marcus et al. 2004] or combined with other analysis [Eisenbarth et al. 2003, Dit et al. 2011] approaches

produce comparable precision and recall values. This improvement appears to be on the same order as what has previously been observed.

The Wilcoxon signed-rank test was performed to investigate whether the difference in terms of effectiveness for the two approaches is statistically significant. We computed it based on the total effort measure (Σ EM) dependent variable. The null hypothesis is that there is no statistical significant difference in terms of effectiveness between LSI and LSI+S.

The alternative hypothesis is that LSI+S has statistically significantly higher effectiveness than LSI. Our results were found to be statistically significant. The p-value is lower than $\alpha = 0.05$, it was actually less than 0.0001. This allows for rejecting the null hypothesis.

All the data from the three experimental studies supports the hypothesis that the addition of the stereotype annotations improves the results of querying in the context of feature location. This lays the foundation to generalize the results further. However the question why this particular type of information helps?, needs to be explained. Beyond the abstract information-theoretic explanation (i.e., more information will give you better results) it would be prudent to understand some of the specific reasons improvements are seen.

It has been found that when using LSI, methods with small bodies and small numbers of identifiers are not ranked correctly [Poshyvanyk et al. 2006] because there is not enough terms to properly build an accurate vector representation. However, the addition of stereotypes seems to mitigate this problem to some degree.

Table 3-7. The difference between the positions of the first relevant and the last relevant method for each query result in Hippodraw and Qt. The last column is the percentage improvement using LSI+S.

Feature	HippoDraw			Qt		
	LSI	LSI+S	%	LSI	LSI+S	%
1	101	31	69%	1052	331	69%
2	61	51	16%	519	466	10%
3	30	21	30%	680	442	35%
4	207	98	53%	706	358	49%
5	215	182	15%	40	32	20%
6	226	128	43%	828	450	46%
7	117	66	44%	100	93	7%
8	1288	532	59%	812	455	44%
9	249	173	31%	152	99	35%
10	123	93	34%	182	149	18%
11	1197	387	78%	1320	449	66%
Average			43%			36%

That is, small methods appear to be ranked more correctly with the extra stereotype information. For example, in HippoDraw feature 3 “update zoom” using LSI resulted in the first relevant function *getZoomMode()* being ranked in the 6th position, while using LSI+S it is ranked first. We investigated this further and made some

interesting observations. LSI ranked the function *hasZoomY()* in the first position, which is not relevant to the feature. However, *hasZoomY()* is small with only a couple lines of code. When re-documented, it is labeled with the *predicate* stereotype. This additional information changed the similarity between it and the query. We observed this same type of situation happening elsewhere. That is, small methods being ranked high by LSI but after being labeled with stereotypes, such as *predicate* or *get*, receiving a much lower ranking.

Later on, additional 14 features were examined, which were derived by investigating eight new bug reports in Qt. These bug reports are given in Table 3-8. These 14 features were chosen because they were the most frequently changed. LSI+S improved or preserved the position of the most relevant method in each case. For instance, the bug 24685 affected versions 4.7.4 and 4.8.0, and was fixed in version 4.8.3.

Based on the bug description, it occurs when the method *QPainter::drawText()* is called from a thread. A memory leak occurs if the text contains Russian characters (i.e., "Время"). For this bug to be fixed the three functions *painter()*, *setFont()*, and *drawText()* all need to be modified. For the query we used the bug title "*memory leak in drawText()*". Using LSI these three methods were ranked 47, 65, and 11 respectively, while using LSI+S they were ranked 28, 31, and 1. An explanation for this result is that the function *drawText()* is overloaded 18 times, 9 of which have only one line of code in the body of the function, and were labeled with *predicate* or *void-accessor*. The others have different and more complex behavior, and were labeled as *command-collaborator* or *void-accessor*.

In the context of our query the most relevant *drawText()* function is labeled with command-collaborator like the other two relevant methods *painter()* and *setFont()*. This function is ranked in the first position using LSI+S, while it is ranked in the 11th position using LSI alone. Another example is the bug 11204, which impacts version 4.6.2, and is fixed version 4.7.1. Based on the description of this bug, this bug involves two features “*direction of text*” and “*alignment of text*”.

Table 3-9 gives the relevant methods for this bug, and how they were ranked using both techniques. In this experiment we used the bug title “*direction change no longer implies alignment change*” as a query. The total effort measure for those new 14 features is examined, LSI+S has better values for all features with 38% average improvement. Moreover, the position of the most relevant method is improved using LSI+S for 10 out of 14 features, where for the remaining 4 features, LSI+S gives the same ranks as shown in Table 3-8.

It is believed that using the stereotype information acts as a type of filtering mechanism when building the LSI subspace. That is, simple methods such as *get* and *set*, are superficially related to a feature, as they rarely impact the actual behavior and often play little part in the actual maintenance task. However, this belief is speculative in part and further investigation is needed to substantiate or generalize this hypothesis.

Stereotypes, by nature, increase the similarities between any two methods that have the same category. Since stereotypes are an abstract summary of a method’s role and behavior, therefore, this implies that methods with similar roles will be made more similar (within the LSI subspace).

Table 3-8. Description of eight bugs (which corresponding to 14 features) from Qt bug reports. The table cloumn's show the bug number, followed by the number of features that relate to each bug, the component name, and the number of relevanr methods.

Bug Number	Component	Number of Relevant Methods	Rank of Most Relevant	
			LSI	LSI+S
24685 (1)	GUI: Font handling	3	11	1
15754 (3)	GUI: Font handling	7	3	3
11204 (2)	GUI: Text handling	4	3	1
5002 (2)	GUI: OpenGL	10	5	3
4210 (2)	GUI: Painting	9	7	4
2276 (1)	Widgets: Itemviews	13	11	9
1868 (2)	GUI: Text handling	8	1	1
935 (1)	GUI: Workspace	7	25	14

Table 3-9. Comparison results for locating the relevant methods for bug 11204.

Rank _{LSI+S}	Relevant Methods	Rank _{LSI}
1	direction()	43
262	setTextDirection()	285
5	setAlignment()	5
17	fixedAlignment()	21

Table 3-10. Distribution of stereotypes for the relevant methods over both studies. The other 15 were a variety of different stereotypes with no one category making up more than 2%.

Stereotype	Number of Methods	Ratio (%)
<i>Command-Collaborator</i>	221	71%
<i>Command</i>	453	17%
<i>Predicate</i>	20	6%
Others	17	6%
Total	311	100%

Table 3-10 presents an overview of how the relevant methods were stereotyped. This is for both systems across all the 36 features. There were 311 relevant methods. We see that the vast majority (almost 90%) are labeled with the *command* and/or *collaborator* stereotypes. Approximately 6% are predicates and the remaining is a variety with no single stereotype category making up more than 4%. In short, the most relevant methods, in these two studies, are almost always some type of *command* or *collaborator* method.

We observed this distribution after running the studies while attempting to better understand the results.

Command and *collaborator* methods do the majority of the logic within a class. They model the behavior of a class and hence provide most of behavior of observable system features. Thus, it makes sense that the most relevant methods for any system feature would most likely be of the *command* stereotype.

Table 3-11. Stereotypes types for the relevant methods of the feature “remove item”.

Method #	Relevant Function Name	Stereotype Type
1	removeSelected()	Command Collaborator
2	removeSelectedItem()	Command Collaborator
3	removeFromItemList()	Command Collaborator
4	deleteSelectedItem()	Command Collaborator
5	deleteSelected()	Command Collaborator
6	reTile()	Command Collaborator
7	deleteSelectedItem()	Command Collaborator

3.8 Threats to Validity

A number of issues could affect the results of the study we conducted and so may limit the generalizability of the results. We attempted to minimize factors so to decrease their effect. Feature selection is an issue. The features that were picked were commonly modified in the systems based on the documentation. Needed were also features for which all relevant methods could be identified. As such they were selected with no preconceived notion of how well either LSI or LSI+S would perform on them.

The number of queries used could also be too few for a rigorous comparison. Compared to other studies on feature location [Liu et al. 2007] the number we used, 30 queries over 36 features, represents a bit larger set (i.e., previous studies have used as few as three queries). However, other studies [Poshyvanyk et al. 2013] have used more but they depend on bug reports titles or descriptions directly as a query without filtering or preprocessing. They also only include items that were changed due to the bug report. This may not include all relevant items, but only relevant items that were changed. Another issue is if the features used in this study are representative to those used in practice. Taking features directly from active open-source systems minimizes this to a degree. Also, these features were involved in actual maintenance tasks. We also minimized this threat by selecting two different systems from two different domains. Expanding the study to other systems could further minimize this issue. Another issue is that query selection depends on the knowledge of the user. We attempted to minimize this by selecting the best query for LSI and LSI+S from the set of four queries.

Lastly, we may not have found all relevant methods or may have labeled methods as relevant that actually were not. This was addressed by a careful manual inspection of the systems and associated documentation.

3.9 Summary

A novel technique to improve the results of using LSI on the problem of feature location is introduced. The technique involves adding new information to the source code before applying LSI. In this case, the new information added is method stereotypes, which were derived via static program analysis from the source code.

The results of using LSI on the original code base were compared with that of a version re-documented with stereotype information. This experimental study on two open-source systems demonstrated that the added stereotype information improved the query results for the feature-location process. We saw substantial average improvements in the results for all measures. For each individual query we saw equal or better results in all cases when using the stereotype information. The results were compared using recall, precision, position of first and last relevant document, and a total effort measure.

The implications of these results are important for a number of reasons. The results confirm that adding information to a corpus (here source code) will improve the results for extracting and querying that corpus. The results provide evidence that the addition of other information than stereotypes, gained via static or dynamic analysis of the code, could also improve the results. The results also imply that stereotype information is relevant for feature location, which supports our previous studies on stereotypes. This last issue could give rise to a new means for evaluating techniques to support comprehension. If we claim that adding or deriving particular information from source code supports comprehension, then it should in theory also improve the results of IR methods such as LSI.

CHAPTER 4

Source Code Indexing for Feature Location

The main contribution in this chapter is to study and examine the effects of excluding certain textual information (e.g., comments and function calls) from being included when performing source code indexing for feature and concept location purposes. In Software Engineering, the developers in order to identify which parts or fragments of source code that implement a specific task or functionality, they employ Information Retrieval (IR) methods to automatically identify source code that implements them. A key step in this process is indexing all important, valuable, and helpful information from the software artifacts, which is extracted and converted into a suitable representation (corpus) that is compatible with the underlying IR model.

Textual information has the advantage of being commonly obtainable and accessible from the source code, but unfortunately it is exceedingly subjective. The terms may have several meanings. Moreover, functions names from source code are often ambiguous if taken out of the context. And comments are frequently out of date, meaningless, and not well written [Anquetil and Lethbridge 1998]. As mentioned in [Mahmoud and Niu 2011], naming style, and comments are considered as characteristics of source code that make the process of indexing a real challenging task.

This chapter is divided into two main sections, in the first one; we introduce a study about the effect of comments over feature location process. Two experiments for

feature location are conducted; the first one includes the comment, where the second one ignores the comments when indexing the source code. In the second section, we introduce the results for comparing two feature location experiments that were conducted, one with including function calls, and the other with ignoring them when indexing source code.

4.1 A Case Study of Feature Location with and without Comments

Commenting source code is considered as one of the attributes of a great code. Well-documented software components are easily comprehensible and therefore, maintainable and reusable. Studies have shown that the effective use of well written comments can significantly increase a program's comprehension [Dit et al. 2011].

“Comments as well as the structure of the source code aid in program understanding and therefore reduce maintenance costs.” - [Elshoff and Marcotty 1982].

Comments have a very effective and broad range of potential uses, from enriching program source code with meaningful descriptions, to producing the external documentation. Comments are generally written in an easy, readable, and clear form of the human natural language⁷.

The main contribution in this section is to examine and investigate the effect of comments on feature location process. Moreover, we study and analyze the commenting

⁷ <http://www.icsharpcode.net/technotes/>

styles that are being followed by the developers when they assigned the internal documentations (comments) for the three systems we investigated.

4.1.1 Code Comments Overview

Commenting of source code is an important part of the coding style to make the code understandable to the next person who comes along or even for a later usage by the programmers. In other words, comments are usually added with the purpose of making the source code easier to read, understand, and modify [Maletic and Marcus 2000, Binkley and Lawrie 2010]. The flexibility given by comments often permits for a wide level of variability and potentially non-useful information inside the source code of any system. Sometimes, a comment just simply doesn't mean anything. These kinds of comments appear to be making an attempt at explanation, but do it so poorly and they might as well not be there [Cleary et al. 2009] . Comments that are too tiny are too enigmatic. On the other side, comments that are too extended may contain extra, repeated, and meaningless information.

In source code indexing, as shown in Figure 4-1, the comments are considered optional linguistic information that can be extracted from any system source code [Mahmoud and Niu 2011].

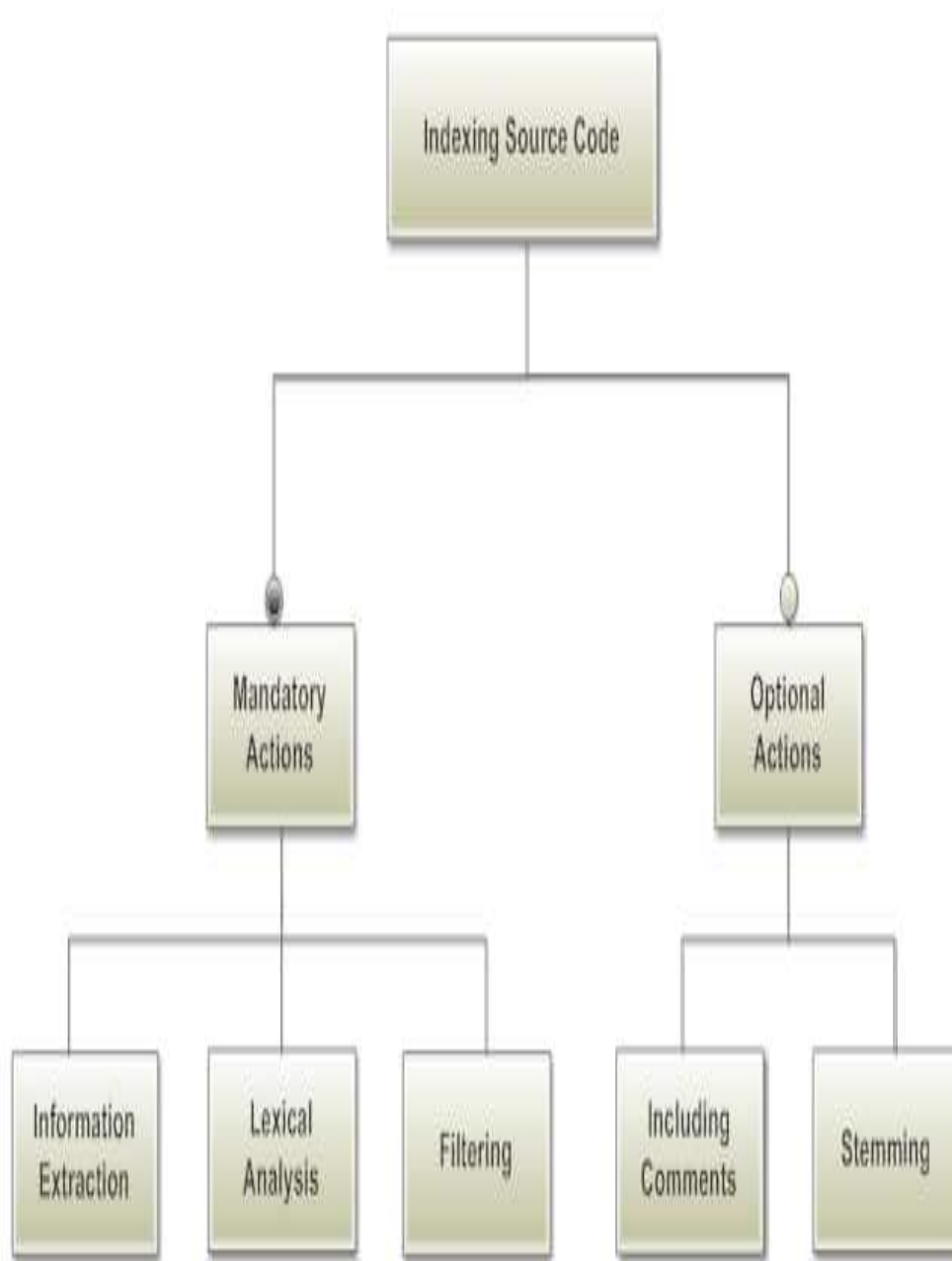


Figure 4-1. A feature diagram for source code indexing.

In [Woodfield et al. 1981], the authors conducted a user study on forty-eight experienced programmers and confirmed that source code with well written comments

can be easily enhanced and improved to be understood and maintained by programmers. When we attempt to perform source code indexing, we must perform *information extraction*, *lexical analysis*, and *filtering*; those components are basics in source code indexing process. Including or excluding *comments* and performing *stemming* in source code indexing process, are recognized as optional steps, since they have produced some discussion in the literature.

Every so often developers include some tokens in their code comments that are used throughout the project task as references, such as (*Author*, *Param*, *Date*, *Copyright notice* or *license terms*).

A substantial amount of research has been done on the topic of studying and evaluating code comments. In [Dit et al. 2011], the authors study has shown that the effective use of well written comments can drastically increase a program's comprehension. However, the amount of research centered towards the quality evaluation of in-line documentation is limited [Padioleau et al. 2009].

Moreover, In [Khamis et al. 2010], the authors present an automated approach for assessing the quality of inline documentation. They applied their tool (JavadocMiner) to the different modules of two open source applications (ArgoUML and Eclipse).

In [Mahmoud and Niu], the authors studied the effectiveness of including comments and making stemming over traceability links , they concluded that considering comments in the indexing process helps in improving the traceability link quality significantly.

The authors in [Schreck et al. 2007], studied the comments importance and one of their recommendations, was that in order to differentiate between source code and documentation, a specific documentation or programming syntax has to be used.

In [Tan et al. 2007], the authors studied the feasibility and the benefits of automatically analyzing comments, their goal was to detect software bugs and bad comments in the source code documentations.

In order to find the bugs that were caused by wrong assumptions made by the programmers, the authors in [Howden 1990], built a tool for comments analysis. And they concluded that not all of programmer's comments are useful or helpful.

4.1.2 Code Comments Categorizations

Based on [Spuida 2002], there are three main classes (styles) or categories for source comments; these classes are categorized based on the purpose of the comment to documentary comments, functional comments, and descriptive comments. The following gives a brief description about each style.

- **Functional Comments**

The main usage of this kind of comments is adding new features to the source code. These comments when added by the programmer, only describe added features. In other words, they do not describe the whole program/project or the history. Such an examples of functional comments are *feature addition*, *bug description*, and *to do*. For code comprehension, this kind of comments should be added in a standard way and assigned reasonably to the fragments of code [Howden 1990].

- **Documentary Comments**

This type of comments is called documentary since it is used to document the development of the software project and the history of that project. These comments contain information about the project components as we see in Figure 4-2, such as filename, version number, author's name, and project or program purposes, etc.,

The main goal of this kind of comments is to keep the program/project maintenance or updating easy. Moreover, this kind of comments can contain a good description for the hardware needed. In other words, it gives the programmers, especially the new ones, a summarization about the program before changing or marinating it [Tan et al. 2007].

- **Descriptive Comments**

When the programmers write the code in a very well way this kind of comments shows up a lot. However, this comment does not need to be added for each line of code or for each statement. *Sub routines* and *methods* (functions), the *starting up code*, and *regular expression* are the most popular examples where the descriptive (explanatory) comments should be added. Figure 4-3, is an example for this kind of comments. For instance, as we can see in the figure, each regular expression has a descriptive comment that describes it briefly [Tan et al. 2007].

File:	PCMBOAT5.PAS
Author:	B. Spuida
Date:	1.5.1999
Revision:	1.1
	PCM-DAS08 and -16S/12 are supported. Sorting routine inserted. Set-files are read in and card as well as amplification factor are parsed.
1.1.1	Standard deviation is calculated.
1.1.2	Median is output. Modal value is output.
1.1.4	Sign in Set-file is evaluated. Individual values are no longer output. (For tests with raw data use PCMRW.EXE)
To do:	outliers routine to be revised. Statistics routines need reworking. Existing Datafile is backed up.
Purpose:	Used for measurement of profiles using the Water-SP-probes using the amplifier and

Figure 4-2. A snippet for an example about documentary comments [Spuida 2002].

```

void DumpHrefsClean(String inputString) //same as above, commented
{
    Regex r;
    Match m;

    r = new Regex("href          #This looks for the string 'href'
    \\s*=\\s                 #followed by whitespaces, '=', ws
    (?:\\\"(?<1>[^\"]*)\\\"      #a ':', + a group in '\"', no '\"' in it
    |                        #or
    (?<1>\\S+))",             #a group followed by non-spaces
    RegexOptions.IgnoreWhiteSpace|RegexOptions.IgnoreCase|
    RegexOptions.Compiled);
    for (m = r.Match(inputString); m.Success; m = m.NextMatch())
    {
        Console.WriteLine("Found href " + m.Groups[1] + " at "
            + m.Groups[1].Index);
    }
}

```

Figure 4-3. A snippet for an example about descriptive comments [Spuida 2002].

4.1.3 Case Study Comments Samples

In this section, we introduce samples for the taken systems code comments.

- **Qt- Comments Sample**

Here, we give four comments of four different functions from Qt code.

1. "setOpenFileName "

```
"! options selectedFilter fileName openFileNameLabel selectedFilter
options filename."
```

2. "blendComponent "

```
"! shadow gets a color inversely proportional to the alpha value then do
standard blending."
```

3. " findFiles "

```
"! filePattern fileNameComboBox directory directoryComboBox allFiles
directory matchingFiles file."
```

4. "createLayout"

```
"! fileLayout QHBoxLayout directoryLayout QHBoxLayout mainLayout
QVBoxLayout ."
```

- **HippoDraw- Comments Sample**

Here, we give also two comments of two different functions from HippoDraw system source code.

1. "setCutRange “

"setCutRange projector * @bug @@@@@" This needs fixing for two dimension functions.”

2. "mousePressEvent “

“m_plotter. “

As we see in the above samples, that the comment for HippoDraw system are less standardized than Qt, the second comment is too short and meaningless for the developer.

- **KOffice- Comments Sample**

1. "createShape”

“factory shape factory path reset tranformation that might come from the default shape / creates a shape from the given shape id.“

2. "saveImage”

“format NULL ret pixmap Save the image.“

4.1.4 Evaluation Strategy and Results Discussion

This section tries to answer the following question:

“Should comments always be considered when indexing source code for feature and concept location?”

To answer this question, two experiments for feature location using LSI were conducted; the first experiment is done with including the comments when performing

source code indexing and the other one with excluding the comments. The stop-list removal and stemming were performed with the two experiments.

For evaluation, we use the same data set from chapter 3 with addition to a new system (KOffice). The results analysis as shown in Figure 4-4, Figure 4-5, and Figure 4-6, show that considering comments in the indexing process has a significant effect on the retrieval effectiveness for some systems. For example as shown in Figure 4-4, including the comments when experimenting Qt System improved the results, and the main reason behind that is the developers of Qt followed a standard style when commenting the source code.

However, it has a negative effect on the other two systems, as shown in Figure 4-5 and Figure 4-6 for HippoDraw and KOffice systems respectively, the reason behind this result are the contents of comments in both systems; there are a lot of meaningless comments in both system source codes. In other words, some systems are well commented by the developers, while other systems have no standardization in writing the comments. Our findings match the fact “a useful comment always follows some basic rules of style.” which was presented in [Spuida 2002].

Moreover, including or excluding the comments depends on the contents of the comments. The results show that some comments contain invaluable information (*copyright notice* or *license terms*), even after removing the stop list words, some terms stay indexed and negatively affect the feature location results for some systems.

Table 4-1, shows the comments density for the three systems, the density is measured as what is the percentage of all comments line compared with all source code

lines for each system separately. The table shows that Qt system has the largest percentage of comments, which means that the developers commented the source code enough, and this was reflected positively on the results of feature location when including them in indexing process.

Table 4-1. Comments Density for the three systems, computed based on the number of lines of code of each system separately.

Systems	Comments-Density (%) LOC
Qt	18
KOffice	12
HippoDraw	11

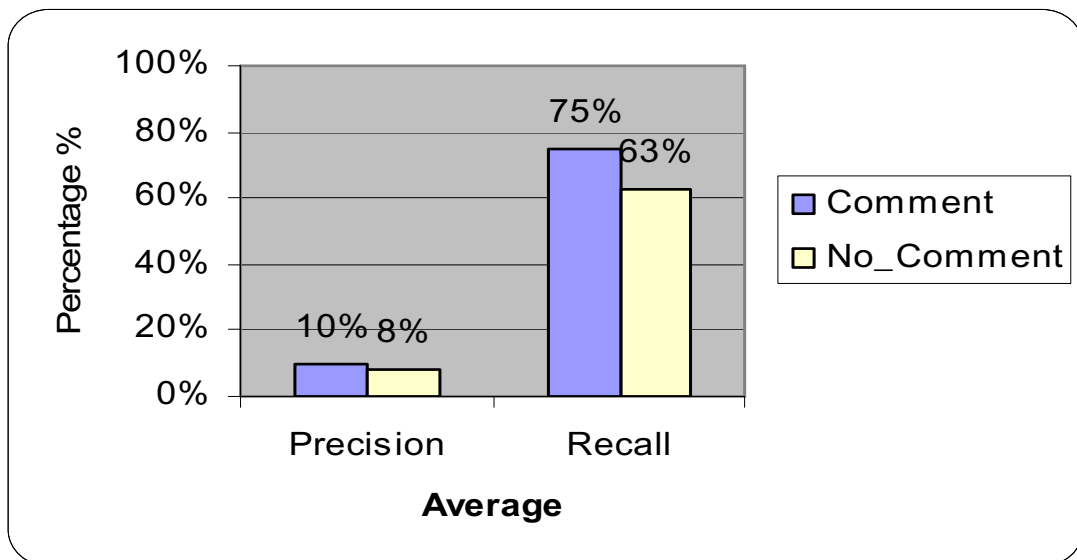


Figure 4-4. Qt-system experiments results average.

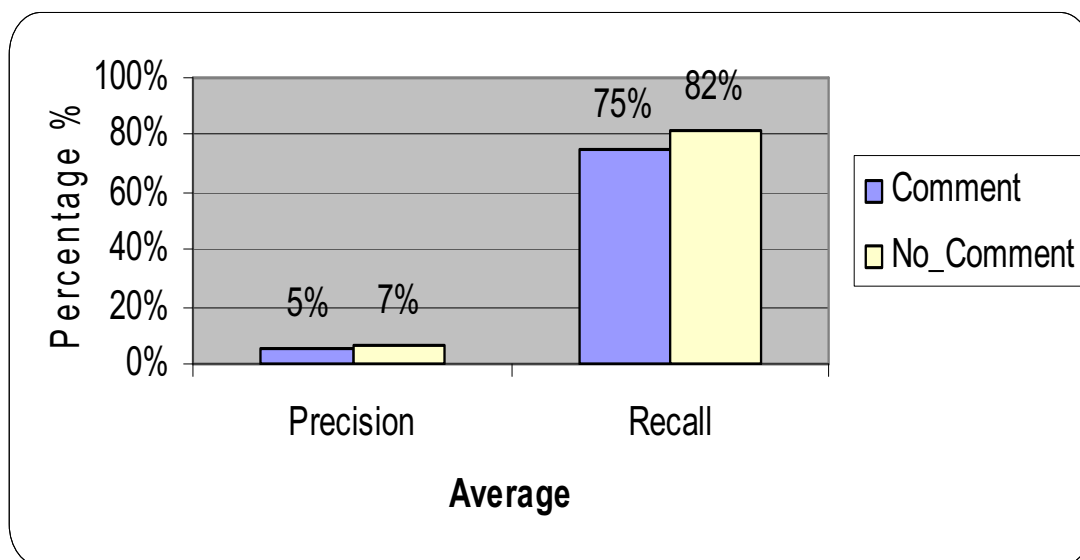


Figure 4-5. HippoDraw-system experiments results average.

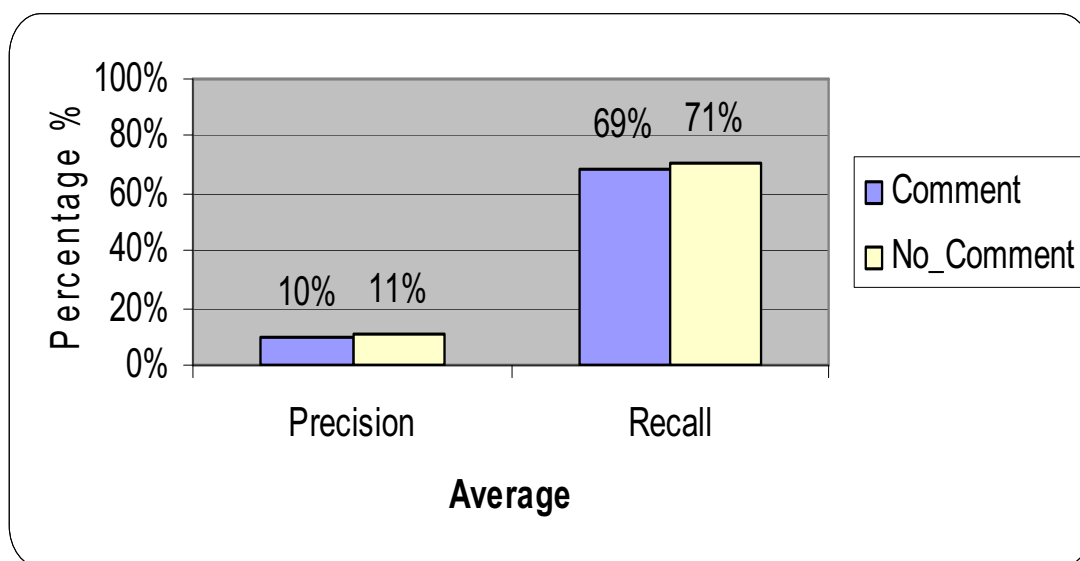


Figure 4-6. KOffice-system experiments results average.

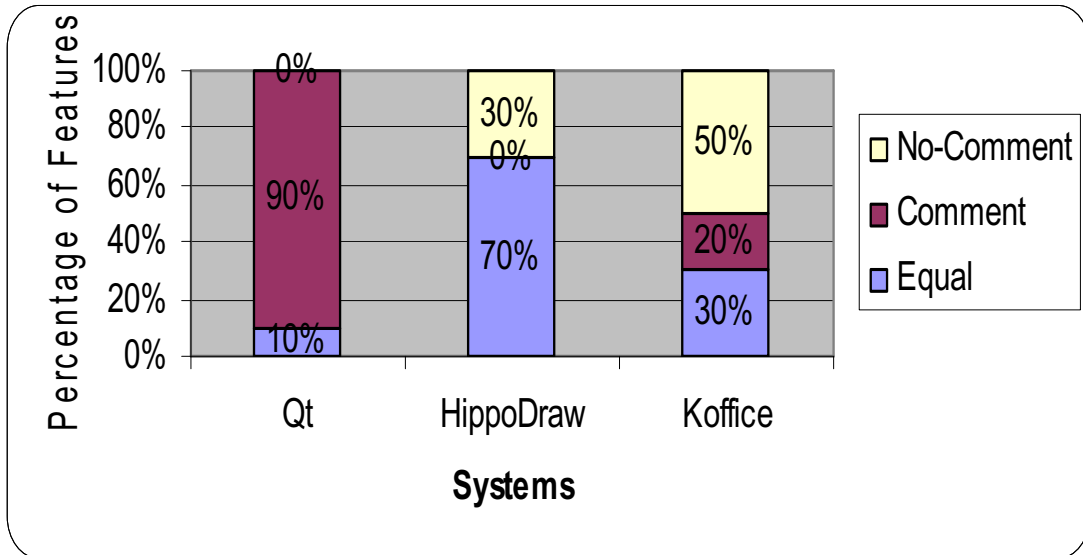


Figure 4-7. Ranking comparison for all relevant methods of all taken systems queries. Three cases taken, the red color shows the percentage of relevant methods that best answered when including the comments. The yellow color shows the percentage when excluding the comments, and finally the blue color shows the percentage when including and excluding the comments do the same.

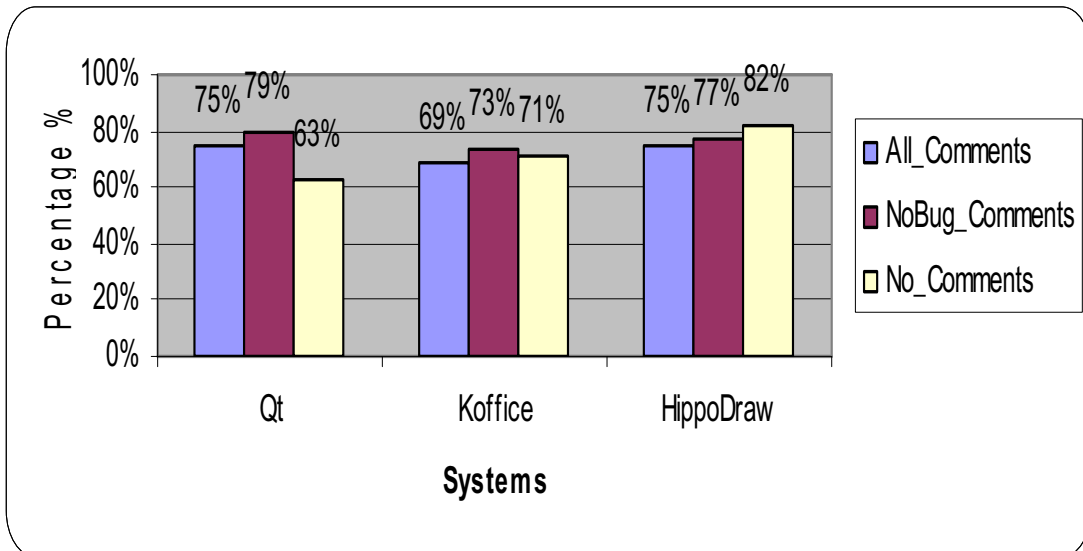


Figure 4-8. Comparison results (Recall) for the relevant methods of all queries. Three cases taken, one with including all comments, and one without including any comments, and the finally one, is when including the comments except the bug comments.

Figure 4-7 shows comparing results between including and excluding the comments from the corpus of each system. For instance, for the Qt system, as shown in the figure, for the taken experiment (looked features/query), 90% of the queries were best answered (best ranked) when including the comments, where the rest 10% gave the same ranks for the relevant methods, either when including or excluding the comment from the corpus. As mentioned and shown before, the comments of the Qt system are almost done in a standard manner.

For the HippoDraw system, as shown in the same figure, the results are different than those for the Qt; 70% of the queries were not affected by including the comments. In other words, the comments did not affect the location process positively nor negatively for those queries (features), the rest 30% of the queries got improved when including the comments. This means that the developers of HippoDraw didn't follow a standard way when commenting the code, moreover, the comments of HippoDraw itself as shown before, doesn't contain a lot of meaningful/helpful information with respect to location process.

For the KOffice system, the results are little bit different. As shown in the above figure, excluding the comments improved 50% of the queries while including the comments just improved 20%. On the other hand, 30% of the queries are not affected by excluding or including the comments. This means that the developers didn't comment on the fragments of source code that are related to those features or queries well. Moreover, recall and precision are compared for the three cases (with comment, without comments,

with comments except bug comments) as shown in Figure 4-8 and Figure 4-9 respectively.

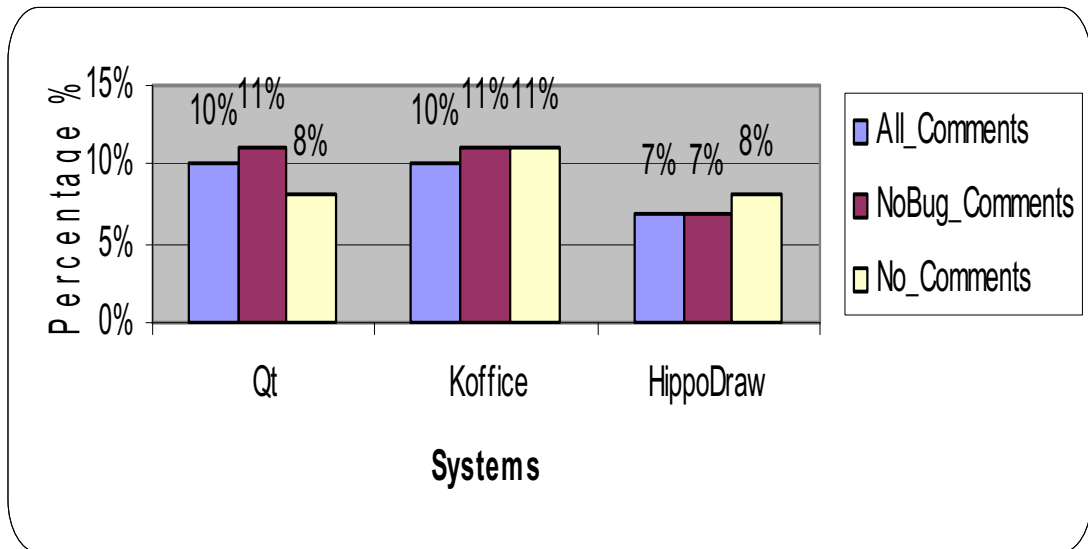


Figure 4-9. Comparison results (Precision) for the relevant methods of all queries. Three cases taken, one with including all comments, and one without including any comments, and the finally one, is when including the comments except the bug comments.

4.1.5 Study Recommendations

Here, based on the results, we present the study recommendations for developers when attempting to comment their code. These findings will definitely help in improving program comprehension activity.

1. Comments should say differently what happens in the source code block, rather than being a pure literal translation into human language.
2. Comments should be placed according to their related code blocks (front of the related code).

3. Comments should be short and should assist readers in understanding the next stage of aspects in the program (perform a bridge between the reader and the code). Computers do not at all read the comments, while programmers tend to read comments rather than codes.
4. Comments should be avoided to be in-line comments within the body of the component itself.
5. Comments should be written by the programmer in a consistent standard way for the whole program as much as possible.
6. There are many forms to comment such as including design and discovery docs (e.g. UML, Logos, diagrams, and flowcharts) and changing comments (e.g. this change fixes bug 2938). Therefore, describing these different forms in the human language would help in code comprehension.
7. Comments should be written professionally to answer why, not what.
8. Comments that are well written shouldn't be repeated a lot across the code.
9. Comments should be processed separately before indexing the source code.
10. Comments should cover all the core code. That is, the developers should describe the central parts of source code enough.

4.1.6 Summary

The main objective in this section is to investigate an empirical answer for the question: Should comments be considered always when indexing source code for feature and concept location? To answer this question, we conducted experiments over three open systems, named Qt, HippoDraw, and KOffice. These systems provide a variety of

applications, domains, programming languages, development practices, sizes, and commenting styles.

We used the data set from chapter 3 in the evaluation. The results show that for indexing source code for feature location purposes, not all comments should be included or considered. For instance, for the Qt system, the comments are written in a more standardized style than those for HippoDraw and Koffic. Moreover, for HippoDraw system, the results show that comments play a minor role in improving the results of feature location.

Therefore, including or excluding the comments when indexing a source code is mainly dependent on how much the comments of any system are written in a standard way, whether the comments are up to date or not, and how much these comments are meaningful and helpful.

4.2 A Case Study of Feature Location with and without Function Calls

This section tries to answer the following question:

“Should Function calls be considered always when indexing source code for feature and concept location?”

To answer this question, two experiments for feature location using LSI were conducted; the first experiment was done with including the function calls when performing source code indexing and the other one with excluding them. The stop-list removal and stemming were performed in both experiments.

For evaluation, the work presented here uses the same data set from chapter 3, with addition of a new system (KOffice).

4.2.1 Function Calls Overview

Mainly there are two types of code functions: *built-in functions* and *user-defined functions*⁸. A built-in function is pre-constructed and is accessible for use in any program. The user-defined function must be constructed by the programmer. The user defined functions contain the functional behaviors of the program. Generally, function insides code represents a unit that performs specific tasks.

One of the excellent recommendations with respect to great coding is to divide the program into as many functions as possible, even if doing this requires more coding. Moreover, breaking the program into manageable fragments help in future in re-implementing and testing these fragments independently.

The use of functions keeps away from a lot of problems [Anquetil and Lethbridge 1998]. Therefore, the programmers while coding, they document the functions they created internally (comments) or externally by describing what the goals behind each added function.

The developers frequently study and analyze program function calls when attempting to understand any large and complex program for maintenance purposes. Function calls show how source code fragments interact, moreover, it shows the locations of source code where a specific feature or concept is implemented. In other words,

⁸ <http://www.cplusplus.com>

function calls work as a navigator for source code components relationships and for the flow of code behavior.

Moreover, analyzing function calls can help the developers in discovering and mapping unknown source code for enhancement or maintenance tasks or activities [Anquetil and Lethbridge 1998]. That is, function calls express the relationships and the dependencies between source code fragments.

4.2.2 Function Calls in Code Comprehension

It is easier and more accurate to think of functions rather than writing the whole program as one large unit. Instead of writing the code within the main program, make a function call in main and code the function separately across the source code as needed.

The researchers focused a lot on the idea of using functional abstractions and function calls to improve code searching [Stylos and Myers 2006, Chatterjee et al. 2009, Ossher et al. 2009] . In [Livshits and Zimmermann 2005], the authors have applied data-mining techniques, explicitly frequent-pattern mining algorithms to the problem of uncovering/discovering call-usage patterns from large systems code.

In [Sim et al. 1998], the authors found that the search goals that mentioned frequently by developers were code reuse, defect repair, program understanding, feature addition and impact analysis. Moreover, they found that programmers were most frequently looking for function definitions, variable definitions, all uses of a function and all uses of a variable. However, several works show that when the programmers attempt to search, analyze, and understand the source code, they are most likely interested in

finding definitions of functions and chains of function calls than code variables, statements, or random fragments of source code [Sillito et al. 2008].

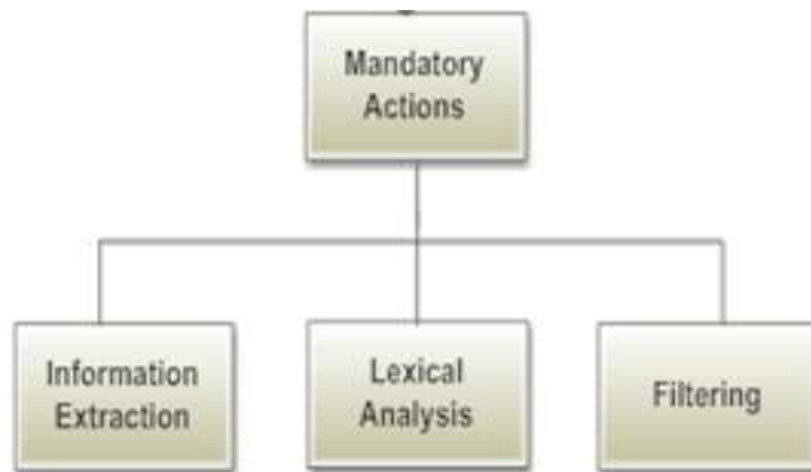


Figure 4-10. The mandatory actions that must be considered when indexing source code.

A lot of researches have been conducted about usage of function calls in Software Engineering tasks (e.g., code comprehension, discovering data dependency, and expressing program data flow). Here we mention some of them. In [Bohnet and Döllner 2006], the authors present a prototype tool for analyzing feature implementation of large software systems by building and visualizing function call graph.

Call graph is defined as a directed graph that stands for relationships of calling between fragments in the source code. Specifically, each node represents a function and each edge (f,g) indicates that function f calls function g . *gprof*, *KCachegrind*, and *CodeAnalyst* are examples of tools that generate and build program call graph [Ryder 1979]. In feature and concept location, as shown in Figure 4-10, function calls extraction

is considered as a mandatory action when indexing source code [Marcus et al. 2004, Liu et al. 2007, Mahmoud and Niu 2011].

Moreover, the researchers have developed a lot of tools to help in code comprehension, these tools stand mainly on extracting function calls from source code. For example, *Brilliant* source code browser, it can import sources in many different languages, and split them down into classes/methods/functions, *Exploration Tools*: it is a command-line based set of tools for examining functions and the structure of C source code, it allows the user to scan and analyze source code to build function call hierarchy and data structure relations, and *Source Navigator* tool; it is known as source code comprehension and documentation tool, it allows the developers to perform source browsing, showing relationships (call/callby/include/includeby/etc.) between the various parts of the program. In [Padioleau et al. 2009], the authors presented call-extraction tool, namely *callextractor*, their tool can perform ordered-pattern extraction.

In [Laski and Korel 1983], the authors used the function calls for source code directed testing of functional programs. In [Berg 1995] the authors use call graphs in the context of software measurement for functional programs. They consider function calls as atomic operations and are produced for each function independently.

In [McMillan et al. 2011], the authors introduce a code search system called *Portfolio*. This tool supports and helps programmers in identifying the relevant functions or fragments of source code that implement a specific concept that are reflected in developer query expression, and determining how these functions are well relevant to the query, moreover, the tool also make visualizing dependencies of the retrieved functions

to show their flows. In [Holzmann 2002], the authors use function calls as a guide in order to do local and global analysis in source code by finding paths in the control-flow graphs of functions. The author concluded that identifying the list of functions that called from a given function, can help in better understanding of source code specially for large and complex programs.

At this point, for code indexing purposes, we study the function call with depth equal 1, the next step to do in the future, is to study the feature location with different depth (2, 3, or 4) of function calls.

4.2.3 Evaluation Strategy and Discussion

This section discusses the results of the conducted experiments, as shown in all figures below; including function calls with depth one to the indexing process has a significant effect on the process of feature location. As we see that for the three experimented systems, the results have been improved significantly, and the queries are better answered when function calls are included. For the three systems taken, the recall and precision results have been improved for all quires.

As we can see in all results figures, including function calls improve feature location process for all systems we studied. In other words, including the function calls in the document of each method (function) in the corpus is enriching the corpus with helpful information. This information improves the searching process. That is, when two documents share the same function calls, there should be a structured relationship between these two documents (functions) [Maletic and Marcus 2001, Binkley and Lawrie 2010].

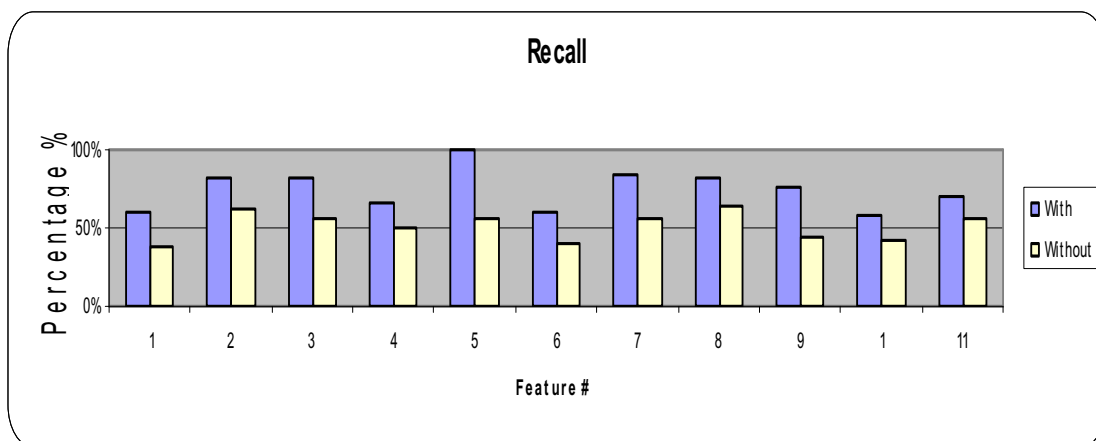


Figure 4-11. Recall results for Qt system experiment.

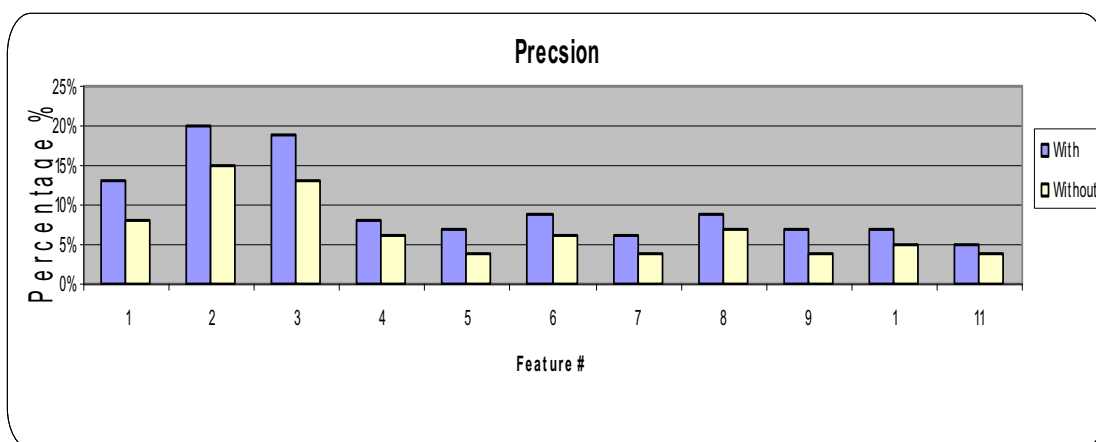


Figure 4-12. Precision results for Qt system experiment.

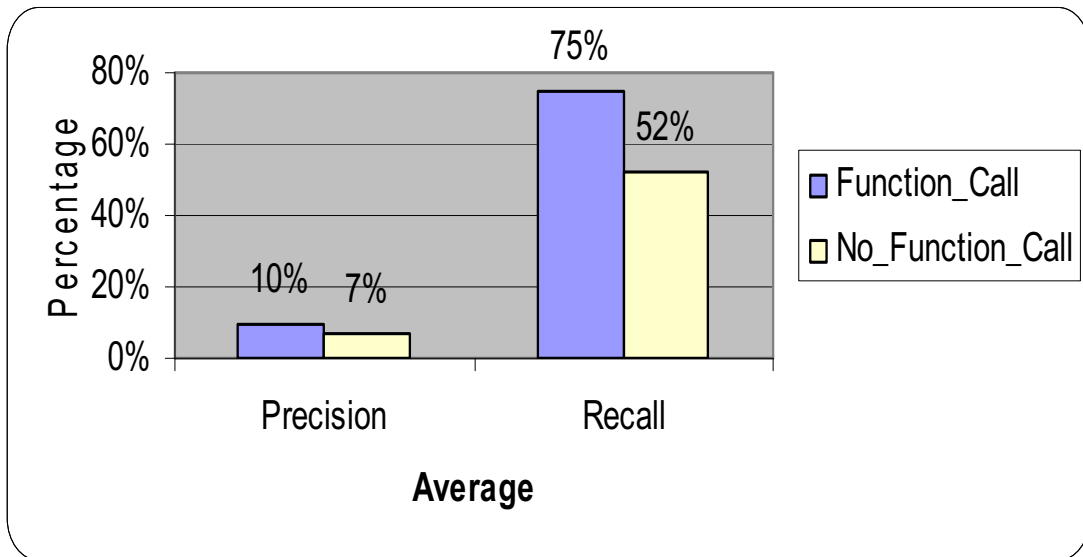


Figure 4-13. Average of recall and precision for Qt system experiment results.

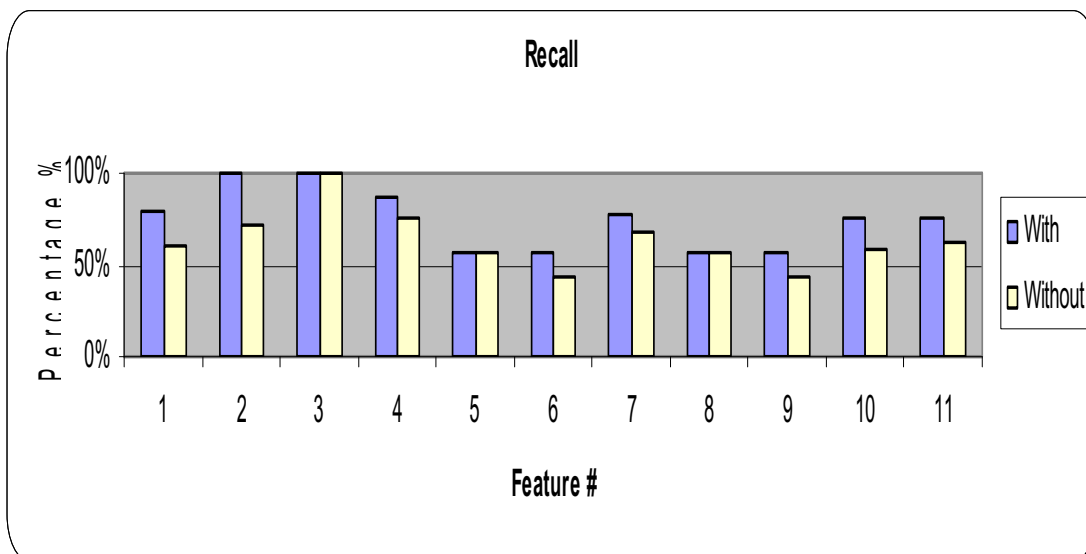


Figure 4-14. Recall results for HippoDraw system experiment.

Therefore, function calls play a major role in enriching the source code corpus with helpful textual information that is reflected positively on the results of computing

the similarities (cosine) between corpus documents vectors, as the results show, adding function calls increase it for these similar vectors.

For Qt system, the percentage of improvement when including function call with respect to the recall measurement, is equal to 23%, see Figure 4-13, while it is 12% for HippoDraw system as shown in Figure 4-16, and 7% for KOffice, as shown also in Figure 4-17 . One of the reasons behind the big improvement in Qt system is that the naming style that Qt developers follow is consistence and standard. For HippoDraw and KOffice, their developers are following less consistence style when naming the identifiers (variables + functions).

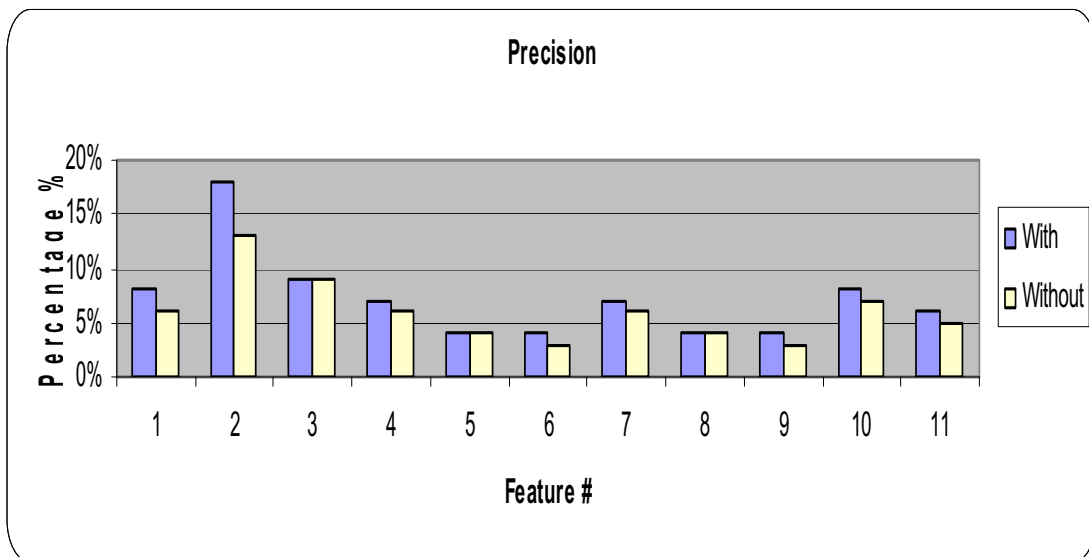


Figure 4-15. Precision results for HippoDraw system experiment.

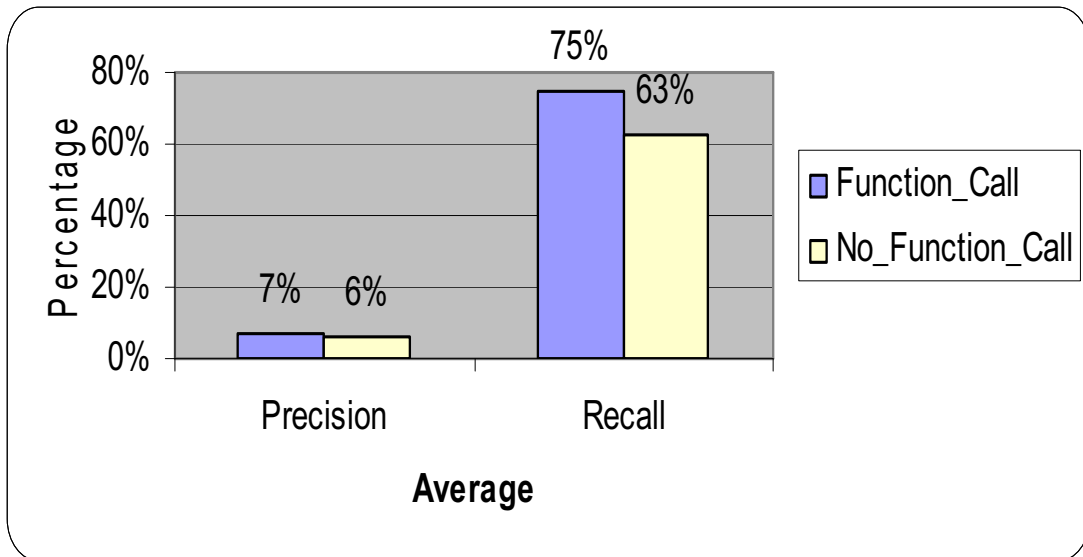


Figure 4-16. Average of recall and precision for HippoDraw system experiment results.

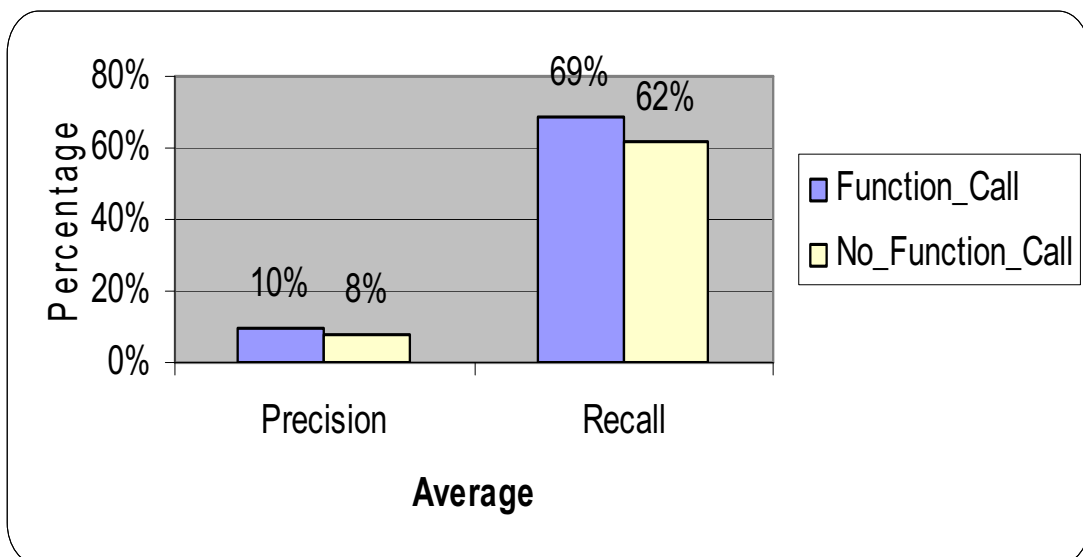


Figure 4-17. Average of recall and precision for KOffice system experiment results.

4.2.4 Summary

The main objective in this section is to investigate an empirical answer for the question:

“Should the developers always consider function calls when performing source code indexing for feature and concept location purposes?”

To answer this question, we conducted an experiment over three open systems, namely Qt, HippoDraw, and KOffice. As we mentioned in the previous section, the systems we chose provide a variety of applications, domains, programming languages, development practices, sizes, and commenting styles.

For the experiments that were conducted for this section, the same data set from chapter 3 is used. The results show that including function calls when indexing source code for feature location purposes, improves significantly the process of feature location.

Therefore, our findings match the results in [Mahmoud and Niu 2011], that including function calls must be considered when indexing the code. Moreover, the results show that the more the developers use standard identifiers (variables and function) naming style, the more the result would be improved.

CHAPTER 5

LSI-Based Solution for Categorizing Software Repository Commits for Maintenance

This chapter presents a novel approach to automatically categorize repository commits based on maintenance types into adaptive, corrective, perfective, and preventive. The approach is currently evaluated by identifying the adaptive commits changes over three open source systems. The next step to do in the future is to investigate and identify all other types of maintenance.

Typically, open source systems evolve during years of development history, where millions of lines of code are maintained by a set of expert developers. Evolution of a software system is normally documented as commits, for the entire period of a project, in version control systems such as *subversion* or *CVS*. The documented dataset includes metadata about the accomplished changes. Such data include why the change was made, when the change was applied, and who makes changes to the necessary files to implement the maintenance request. The developers with purposes of improving system maintenance activities and save the time and efforts needed, they extracted, studied, and analyzed those commits.

The proposed approach uses an advanced IR technique, LSI [Deerwester et al. 1990], to locate for each type of maintenance the corresponding commits in the software

repository. The approach simply builds a corpus for log messages and then makes a topic modeling for the corpus.

The work presented here has two main contributions as shown in Figure 5-1, the first one is enable developers to gain an overview of the past maintenance activities applied to any software system by semantically extracting natural language topics (clustering) using the commits, and the other one is to query the corpus in order to identify the maintenance type that each commits belong to. The approach uses two different techniques for querying, one is using all the terms of each topic as a separate query, and the other one is based on generating a query by choosing specific terms from each topic using Term Average Model (TAM). The approach is evaluated using a collection of commits for three popular large open source projects (Extargear/Graphics, KOffice, and OSG).

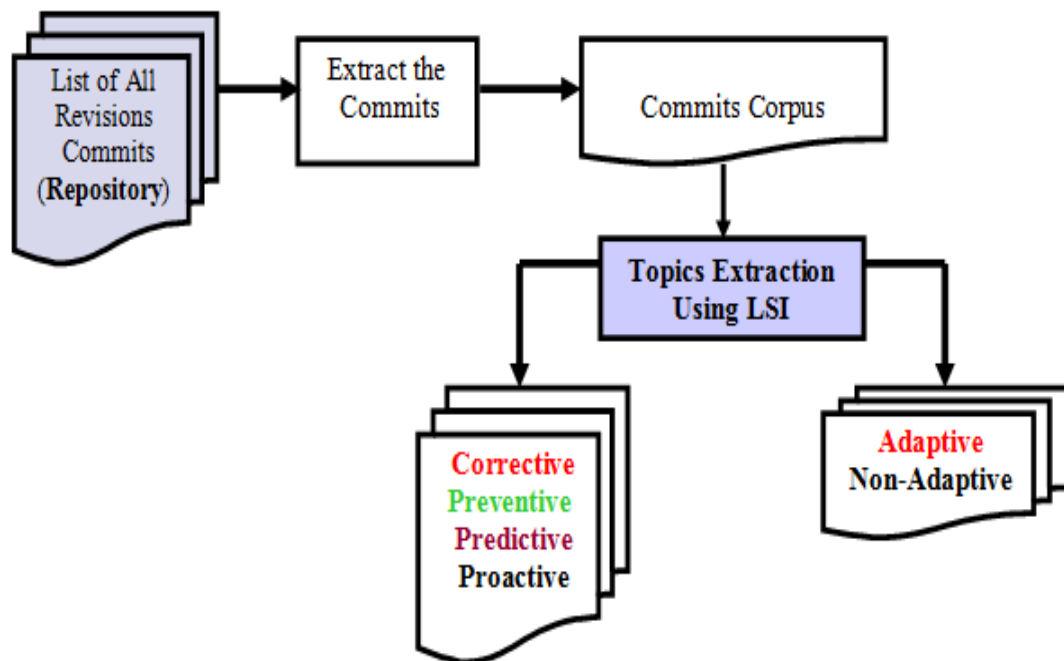


Figure 5-1. Repository commits categorization steps.

The current experiments results show that the proposed approach is able to classify adaptive commits which are derived from associate tasks that support undertaken adaptive maintenance. In other words, the approach accurately retrieves relevant adaptive commits when querying the commits corpus. Upon querying the commits available in a version control system, it achieves nearly 90% recall.

5.1 Repository Commits Overview

Generally, software repository goal is to support software evolution by managing the lifecycle of software revolutionize. Software revolutionize can be defined as performing any task for of any software artifacts (e.g., addition, deletion, replacing, or updating).

Software repository consists of what is called a metadata; this kind of data encloses and includes important information, either explicitly or implicitly. The developers employ this information efficiently to derive high-level semantic information in the context of software revolutionizes. Moreover, this information can be analyzed separately and combined with other sources of information (e.g., source code) [Kagdi et al. 2007]. Furthermore, Researchers used this information to extract relevant information and to discover the relationships or styles about a particular evolutionary characteristic.

Typically, each commit has a commit message to sign and show the main purpose behind the changes that were applied and to displays the information to other developers. The majority of large systems developers have developed a standard way of formatting commit messages that everyone is expected to follow.

When the commit is checked out; it does not offer much information. Therefore, a lot of tools were produced to enable developers to see the same information in a much more useful way (e.g., `git log -g`) which is in a normal log output form.

Moreover, numerous tools were produced in the last decades in order to control and deal with systems repositories, for instance, source-control systems which are used for recording and organizing changes to source code artifacts, defect-tracking systems which are used for managing the reporting process of any source code enhancements (e.g., bugs, and feature), and archived communications which keep and archive the discussion between developers. Moreover, Researchers studied and supported software repository in multiple ways with the goal of utilizing the history of software projects in order to improve future evolution of the subject software system. For instance, a number of approaches have been proposed to combine the various software repository into a regular universal information source [Alonso et al. 2004, Gasser et al. 2004, German 2004, Robles et al. 2004, Zimmermann et al. 2004, Conklin et al. 2005]. A combination of information in the *CVS* log file (change deltas) and *Bugzilla* is used to study *fix-inducing* changes by Sliwerski et al. [Sliwerski et al. 2005]. The information that were extracted from *CVS* log files, are presented using a graph provided by Hindle and German [Hindle and German 2005].

As a conclusion, the proposed approach in this chapter addresses a very significant issue faced by approximately all organizations that depend on large software systems repository. The ultimate objective here is to decrease the cost and increase the

quality of discovering and identifying maintenance types during large software systems development and evolution.

5.2 Version Control Systems

Version Control Systems are standard tools that conserve changes to source code artifacts during the development and maintenance of software systems. Therefore, any change is checked into repository using such control systems, and each check-in is called a commit. Version number assignment and metadata are associated at the change-set level, and recorded as a log entry. Figure 5-2 shows a log entry from the *Subversion* repository of KOffice (a part of KDE repository).

A log entry corresponds to a single commit operation. This commit log information can be readily obtained by using the command-line client SVN log and a number of APIs (e.g., pysvn). *Subversion's* log entries include the dimensions, author, date, and paths involved in a change-set. As an example, Figure 5-2, where the revision number 545547 is assigned to the entire change-set (and not to each file that is changed as is in the case with some version control systems such as *CVS*). The changes in the files `editor.cpp` and `test.cpp` are committed together by the developer *adridg* on the date/time 2006-05-27T18:47:40.125692Z.

Additionally, a text message describing the change entered by the developer is also documented. That is, the purpose of applied change can be clarified by this message terms. It should be noted that the order in which the files appear in the log entry is not necessarily the order in which they were changed.

```
<logentry
    revision="545547">
  <author>adridg</author>
  <date>2006-05-27T18:47:40.125692Z</date>
  <paths>
    <path action="M">/trunk/koffice/libs/koproperty/editor.cpp
  </path>
    <path action="M">/trunk/koffice/libs/koproperty/test/test.cpp
  </path>
  </paths>
  <msg>Qt4 porting</msg>
</logentry>
```

Figure 5-2. A Snippet of KOffice subversion log.

5.3 Commits Identification

Since the purposes of the primary maintenance activities (adaptive, corrective, preventive, and perfective) are well known, commits can be classified according to the purpose of the implemented change.

The research interest is in uncovering maintenance labeling based on commits. given that the submitted commit generally does not keep a tag that would identify and make clear the purpose of undergone change [Mockus and Votta 2000],

Accordingly, the upcoming solution would be for developers to manually extract the maintenance type for all changes. This manual approach should disclose the intention of investigation in the commit messages, where these commit messages clarify what the programmer did and what the intended purpose of the undergone change was.

5.4 Related Works

An overview of existing software repository classification approaches is reviewed in the upcoming sections along with related work on using IR for software repository.

5.4.1 Previous Work on Software Repository Classification

Historically, there is a wide range of research investigations that concern categorizing maintenance commits based on the type of undertaken changes. A number of methodologies have been proposed on utilizing the commit log information stored in repository for change classification purpose. Automatic classification of large changes in software systems into various categories of maintenance tasks using machine learning techniques is given in [Hindle et al. 2009]. The proposed classification is based on word distribution of a commit message, commit author, and modules modified. The authors reported various results that show the usefulness generated by using commit message to provide valuable information about the maintenance class of a commit, where words of this message can identify the reason for the accomplished maintenance activities.

Hattori and Lanza [Hattori and Lanza 2008] proposed a commit classification through concerning the commit size, which is derived from the number of touched files.

Additionally, they have classified commits by the types of development and maintenance activities based on the content of their textual message.

In [Eick et al. 1992], Stephen proposed an approach to automatically discover the structure of textual repositories; the approach utilizes statistical topic models. Moreover, with the purpose of categorizing software systems, Kawaguchi et al. in [Kawaguchi et al. 2003], presented an algorithm to help in automatically finding similar software systems in software archive. Furthermore, in [Kawaguchi et al. 2006], Kawaguchi et al. presented MUDABlue tool that also efficiently and automatically can categorizes software systems.

In the context of maintenance tasks, the authors in [McMillan et al. 2011] proposed an approach that can automatically categorize the applications of software. The approach suggested by the author includes singling out the APIs used by applications and employing them as elements for categorization.

5.4.2 Previous Work on the use of IR in Software Repository

The researchers in Software Engineering field in the last decade started studying and planning how to use IR methods (LSI, LDA, Lucia and VSM) to mine software repository. These studies focus on concept mining [Ohba and Gondow 2005], constructing source code search engines [Chen et al. 2001], or recovering traceability links between artifacts [Antoniol et al. 2002] etc. Generally, the textual commits of *CVS* make IR an attractive option; therefore, IR is used for utilizing the usage of software repository successfully [Kagdi et al. 2007].

An approach that stands on LSI and machine learning methods to recommend software development artifacts based on analyzing the software repository was proposed by David in [Shepherd et al. 2007].

An IR-based automatic keyword clustering and classification was presented by Mockus and Votta [Mockus and Votta 2000]. They derived a heuristic based algorithm to classify modification requests and commits based on the keywords in the textual abstract of the change. For example, if keywords like ‘add’ and ‘new’ were present, the change was classified as pertaining to adaptive maintenance. However, there were numerous cases when changes could not be correctly classified using the appearance of specific keywords. Moreover, the commit terms depend in large part on the developer’s subjective style, and this in turn results in a discrepancy from system to another.

Canfora and Cerulo [Canfora and Cerulo 2005] used the bug descriptions and the *CVS* commit messages for the purpose of change predictions. They proposed an IR method to index the changed files in the *CVS* repository with the textual description of past bug reports in the Bugzilla repository and the *CVS* commit messages. In their approach, they link each bug report with associated *CVS* commit using explicit bug identifier. Consequently, a list of relevant files that are likely to change because of a given bug report can be determined using query generated from the textual description of that report.

An IR method (vector-based) was presented to sketch the similarity between artifacts in [Cubranic and Murphy 2003]. Using this similarity, modification requests in Bugzilla can be related to the files in *CVS* by matching bug-ids in the commit messages.

Their work helps developers to retrieve the relevant artifacts to their maintenance task explicitly in the form of an explicit query.

Hindle et al. [Hindle et al. 2009] applied LDA to commit messages to determine topics that are being accomplished by developers at any given period of time. They proposed topic similarity scores, based on common terms, to link successive time periods that share same activities.

The authors in [Grant et al. 2011] presented an approach to inspect the relationship among co-maintenance record and concept location. Within this approach, the authors visualize the allocation of changes based on concepts to clarify how these methods are capable of being used in forecasting co-maintenance of system's software.

There have been a number of efforts to develop LSI-based approaches for a broad class of investigations with the goal of simplifying the task of understanding software development and evolution. Maletic et al. [Maletic and Valluri 1999] were the first to use LSI to cluster source code documents. Marcus et al. [Marcus and Maletic 2003] proposed an LSI-based method to recover traceability links between source code and documentation, such as requirements documents. Measuring the cohesion of the content of a bug report using LSI is offered in [Dit et al. 2008]. The authors applied LSI to a group of bug reports and after that, they set about calculating a similarity measure on each comment within a single bug report.

For more details about understanding software repository using IR approaches, interested readers are referred to Kagdi's survey [Kagdi et al. 2007]. With respect to capturing the adaptive changes, Collard et al. in [Collard et al. 2010] proposed an

efficient approach to locate the source code statements that are in need adaptive modification. They developed a lightweight transformation approach to automate adaptive maintenance changes on large-scale software systems.

To the best of our knowledge, this is the first work in using LSI to help in the process of semantically classifying software repository commits based on maintenance types. In other words, there is no other work in the literature to cluster version history commits of large scale systems using LSI based on change type especially for adaptive changes.

5.5 Case Study: Adaptive Commits Identification

This case study had previously undergone manual identification process by Software Development and Maintenance Laboratory⁹ (SDML) members; more details regarding this matter are in the evaluation section.

Based on the results from the manual investigation, there are specific identifiable vocabularies (terms) in the adaptive commit log messages; however those terms differ from one system to another, and from one maintenance type to another. Commit classification and clustering can be offered by means of similarity concepts that are associated between commits of the same type in version control system. This intuition is also derived from the work done by Mockus and Votta [Mockus and Votta 2000]. The

⁹ <http://www.sdml.info/>

commit clustering, in turn, will be helpful in enhancing the required adaptive maintenance identification process.

IR methods such as VSM or LSI, as mentioned before, are valuable methodologies that are used for the categorization and clustering textual units based on various similarity concepts [Kagdi et al. 2007]. Here, an automatic keyword clustering using LSI approach will be applied on the textual description of the text messages of the associated commits of version control system.

The hypothesis, on which this case study is built, revolves around the factor hypothesis that the resulting LSI clusters contain at least one topic which is associated with the undertaken adaptive maintenance during the evolution of open source systems. For this to be a sound hypothesis, the basic prerequisite is to ask for the relevant adaptive commits explicitly in the form of an explicit query, which is formed from the terms of resultant clusters.

5.5.1 Latent Semantic Indexing (LSI) for Adaptive Commits

As described before, LSI is a corpus based statistical technique which is used for inducing and representing characteristics of the meanings of words and passages (of natural language) reflective in their usage [Deerwester et al. 1990, Marcus et al. 2004]. Among the IR techniques, LSI is considered one of the better techniques [Binkley and Lawrie 2010] that is capable of recognizing the relevant data that are relevant to a user query. Moreover, LSI is language independent, and deals with synonymy and polysemy. More details were presented previously in chapter 3.

To begin the IR process, the corpus for the system that would be inquired must be built as an initial step. The corpus consists of a set of documents.

Now is presented a description of the approach taken at the moment for identifying the adaptive commits. The IR method, LSI is the basis of the approach. Figure 5-3 presents an overview of the entire process.

Firstly, inspected commits were extracted from the examined system repository. A straightforward approach to extract the log entries from a *subversion* repository is to use the client command SVN log from that repository. This command takes a repository URL, a start date, and an end date of a history, and extracts the commits from the repository logs for a specified period.

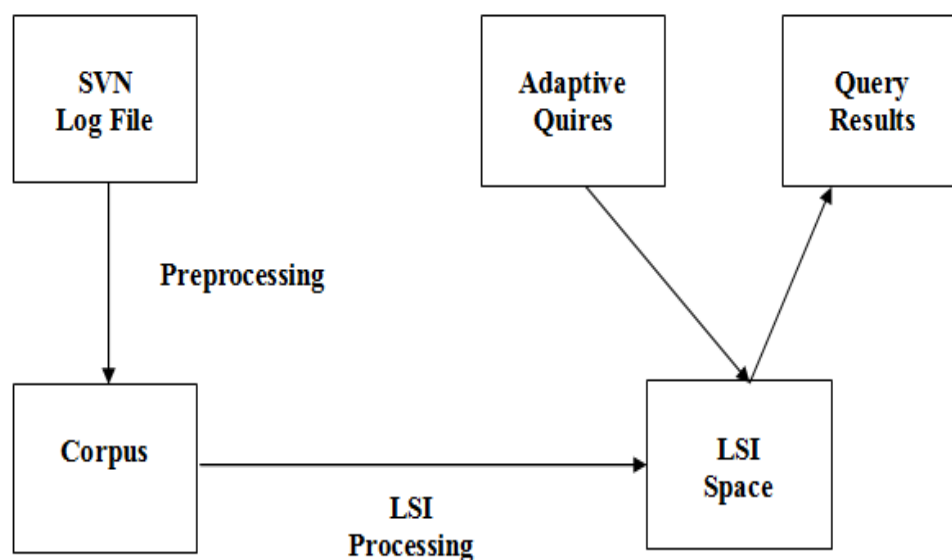


Figure 5-3. Adaptive commits identifying approach.

Subsequently, a corpus was built for those commits; each document in the corpus represented a commit message, and the author of that commit. Afterwards, preprocessing

was applied to the resultant commits to convert them into an input for LSI. This is termed a corpus. Later in this section, how the corpus is generated will be described.

Ranked documents will be retrieved based on their similarities to the query. The user then inspects the results. More details about these steps are covered next.

Corpus Creation

Five actions are taken to create the corpus, the first step is the extraction of commits, the second step is extracting the author name for each commit, the third step is separating the identifiers (terms). The Fourth step is removing the stop words, and finally the corpus is divided into documents (commit level).

An efficient corpus builder was developed in C++ to extract these important elements from SVN log file. Terms are split according to the standard separators [Maletic and Marcus 2000, Marcus et al. 2004, Reville and Poshyvanyk 2009]. An underscore, ‘_’, is used as a separator to split terms that contain more than one word. For example, Adaptive_Commits after splitting becomes Adaptive,Commits, and Adaptive_Commits . Camel casing is also used as a separator, e.g., AdaptiveCommits is split into Adaptive,Commits, and Adaptive_Commits , and ADAPTIVECommits is split into ADAPTIVE, Commits, and ADAPTIVECommits .

The final step of preprocessing is partitioning the commits log documents. Each commit is considered to be a separate document (level of granularity). When the preprocessing is completed the system commits history is represented by a set of documents, $S = \{d_1, d_2, \dots, d_n\}$, where d_i is a commit message and the author name. After these steps, the corpus is built.

Corpus Indexing

Subsequent to creating the LSI space using SVD, each document d_i in system S will have a corresponding vector v_i . Reduction of dimensionality is done in this step and reflects the most important latent aspects of the corpus. The dimension of the vector is a parameter of the algorithm. It is normally between 100 and 300 [Marcus et al. 2004] . The typical manner to choose this value is to run experiments with different values (e.g., 100, 200, and 300) and then select the one that gives the best results. Measuring the similarities between any two documents $\text{sim}(d_i, d_j)$, can be done by measuring the similarities between their correspondents vectors.

Queries Formulation

Typically the user formulates a query by using natural language to describe the change request. This query (q) is converted into a document of LSI space (d_c) and a corresponding vector (v_q) is constructed. Based on the similarity measure between v_q and all documents vectors, the most relevant documents to v_q are retrieved (P_n) ranked according to their relevancy value to the query vector.

Generally, once the LSI retrieves the relevant documents ranked by their similarities based on user query, then the user has the task of inspecting these documents to make a decision of which of them are in fact relevant to the query. The first ranked document (P_1) will be investigated first and then (P_2) and so on. The user decides when to stop the investigation. In this work, we use a threshold of $\delta = 0.65$ when investigating the retrieved ranked list, so, if (P_i) relevancy is greater than the threshold, then it would be investigated to the query of interest.

In [Kuhn et al. 2007], the authors and based on the relevancy of terms, decided which terms must be added to each topic, they concluded that the term to be added to a specific topic must be strongly relevant (high relevancy value) only to that topic. Moreover, the authors after performing documents clustering, they use documents in a cluster as search query to find the most similar terms, and to label a cluster, they take the *top-n* most similar terms.

In this approach, the IR is promoted and employed for different querying purposes. The main contribution of this work, as mentioned before, is to categorize the repository commits based on maintenance types. The approach presented herein performs topics modeling for the commits corpus, more specifically, 10 topics were used. As a next step, two automatic styles were used for formulating the query.

The first formula used is to work by including and considering all terms of each topic as a separate query. This sort of query style is referred to as TopicAllTerms (TAT). The second formula is based upon looking and choosing suitable terms (based on specific criteria's) to be included within the query of each topic. A TermAverageModel(TAM) formulating model was developed. TAM generates the query terms, from topic words, by selecting the suitable terms, which are strongly related to this topic rather than the remaining topics. The following formula (1) ranks high the terms that are very relevant to the current topic but not common to all other topics

$$GAvg_i = \left[\frac{\sum T_{ijr}}{n} \right]$$

- n =number of topics contains term i .
- ΣT_{ijr} = Sum of all term i relevancy across all topics.
- $GAvg_i$: Total average of term i relevancy.
- LT_{ijr} : local relevancy for term i in topic j .

In TAM, Term t_i is included in the query (j) that is related to topic (j) based on the following condition:

if ($LT_{ijr} \geq GAvg_i$)

Add (T_{ij})

else

Discard (T_{ij})

5.5.2 Case Study Evaluation

The main goal of the current evaluation is to assess the accuracy of the suggested approach in correctly examining version histories to identify adaptive commits and directly increasing the productivity of repository mining approaches through clustering the repository commits based solely on maintenance types.

To undertake this evaluation process, two directions are outlined. The first one is to test if the approach is able to construct clusters containing at least one topic associated with adaptive maintenance. The second one is to automatically generate a query using resultant topics that is able to sign and identify a large portion of adaptive commits.

To validate the results, the outcomes of the manual investigation study that was conducted by two PhD students from SDML, where the adaptive commits were identified as a result of this investigation.

Manual Investigation

We examined two main KDE (K Desktop Environment) packages namely KOffice, an office-applications suite, and Extragear/Graphics package, collection of graphical applications that are associated with the KDE project, in the time period of 06/28/2005 to 12/31/2010. Additionally, we studied the OpenSceneGraph (OSG) project, which is a high performance 3D graphics toolkit, in the time period between 08/11/2008 and 03/11/2010.

We manually searched for adaptive commits in order to distinguish between the adaptive and non-adaptive changes. Adaptive commits were identified by searching through the commit log messages for changes in the usages of a specific framework, such as Qt, features and interfaces that were changed to the new features and interfaces found in the new release of that framework. Subsequently, we went about reading over and inspecting the actual commits to make sure they were in fact an adaptive change.

The vast majority of the commits during that time period did not have anything to do with adaptive changes. The other commits addressed corrective maintenance issues or were involved in the adding of a new functionality or features to the examined systems. A summary of this is given in Table 5-1.

Table 5-1. Adaptive and non-adaptive commits for the examined systems.

	KOffice	Extragear/ Graphics	OSG
# Commits in the Log File	38980	26336	4310
Adaptive Changing Starting-Date	03/29/2006	11/07/2006	09/18/2008
# Adaptive Commits	131	219	79
# Non-Adaptive Commits	38849	26117	4231

After identifying the commits involved in the adaptive changes, we examined the vocabulary of the associated commit messages. There are a number of distinguishing techniques that are commonly used in these messages that support possible efforts to identify such commit using IR methods as shown in Table 5-2. More specifically, the terms port, support, add, remove, replace were all used in high frequency within the associated commit log messages. This manual investigation study concludes that a large portion of adaptive changes can be characterized as:

- Involving known API's or language features
- Being system wide and on average large
- Having specific identifiable vocabulary in the commit log messages

5.5.3 Experiments Findings

The input data of this evaluation consists of version history commits. In order to construct a corpus that suits LSI, many preprocessing steps for the input (commits) were undertaken as described before in sub-section 5.5.1. Table 5-3 describes the

characteristics of KOffice, Extragera/Graphics and OSG systems in the context of their use for LSI.

Table 5-2. Frequency of the top 12 average terms in the adaptive commits and their frequency in non-adaptive commits.

Term	Average Rank	
	<i>Adaptive Change Commits</i>	<i>Non-Adaptive Change Commits</i>
Port	45.10%	3.05%
Replace	19.90%	2.80%
Fix	18.70%	22.25%
Remove	16.80%	6.60%
Add	14.60%	19.45%
Test	11.15%	6.90%
Bug	8.90%	10.10%
Compile	6.55%	3.90%
Support	6.30%	2.45%
Cleanup	3.20%	1.60%
Update	1.80%	8.60%
Patch	0.85%	1.20%

Table 5-3. Details of the used corpora. total number of terms for each system, vocabulary size (number of terms after stop list), number of parsed documents, and the dimensionality used for each system.

Properties	KOffice	Extragear/Graphics	OSG
Total # of Terms	281260	164992	48722
Vocabulary Size	14111	10087	5639
Number of Parsed Documents/Commits	38981	26337	4310
Dimensionality Used	300	300	200

The first experiment was conducted to perform clustering the commit corpus. The focus here was on the clustering or grouping of related maintenance commits based on the similarity measure produced by LSI. The topics produced by LSI represent an abstraction of the commits/documents based on a semantic similarity [Maletic and Valluri 1999, Kuhn et al. 2007]. The grouping produced in this automated fashion reflects the reality. Commits that had large amounts of semantic similarity were grouped together and commits with no relation to others remained apart. Once discovered, commits can be in a few words expressed in terms of this structure, queried for topical similarity and so on.

Table 5-4, Table 5-5, and Table 5-6 present the topics for KOffice, Extragear/Graphics, and OSG systems respectively. As shown in these tables, LSI extracted 10 topics (clusters) numbered from 1 to 10. For each topic, the tables show the words that relate to that topic ordered by their relevancies. For example, in Table 5-4, for

topic 1, the word fix is the most relevant word to this topic with 0.705 relevancy, in other words, it can be said that the activity “fix” is the main subject/activity here, followed by compile task with 0.684 relevancy and so on. The topics in the LSI library for the commits corpus seem to reflect the maintenance categories as groups of related commits which address same maintenance problems (reflected and represented by the terms of each topic).

For instance, the topic starting with the term “port”, in Table 5-4, addresses the problem of porting to Qt4 by adding, removing and replacing old functions by the suitable Qt4 methods. By grouping similar commits together, a broader understanding of the system maintenance may be achieved. Additionally, the LSI links related tasks in one topic as what developers do in the reality. An example of such linking is the terms “compile” and “fix” appear in several topics together. Understanding one of the components (activity) in a specific topic implies and gives some basic understanding about which other activities relate to that topic and sometimes for the other topics.

As can be noticed in these tables, there is a gap between the relevancies of each topic terms, this gap is small almost between the top two terms while it increases with respect to the rest of that topic terms. When considering Table 5-4, in topic 1 as shown, fix and compile are the most relevant terms to the topic, while the terms error, crash and warnings are less relevant, that means that the main work/activity covered by this topic is about fixing and compiling. For some topics this gap is small between all topic terms, for example topics 7, 8, 9 and 10 in the same table. This means that there are many tasks or works being done frequently together and being covered in this topic. For instance, topic

8 terms, have small gaps between their relevancies, which mean that the whole terms arise in the original documents frequently together. In other words, it deals with the same task or activity as topic 8 concerning adaptive maintenance, as we will see later.

Table 5-4. KOffice topics and the related terms for each topic.

Topic #	Topics Terms				
1	fix 0.705	compile 0.684	error 0.089	crash 0.057	warnings 0.044
2	compile 0.726	fix 0.652	crash 0.087	add 0.065	warnings 0.063
3	update 0.832	add 0.465	fix 0.121	remove 0.102	api 0.073
4	add 0.780	update 0.542	remove 0.108	test 0.103	fix 0.087
5	warnings 0.972	wemove 0.138	add 0.094	deprecated 0.094	cleanup 0.084
6	remove 0.620	cleanup 0.578	add 0.254	code 0.201	warnings 0.190
7	cleanup 0.772	remove 0.466	fix 0.354	support 0.332	debug 0.307
8	port 0.625	replace 0.5101	remove 0.320	add 0.202	qt4 0.191
9	api 0.734	port 0.383	support 0.301	new 0.147	cleanup 0.103
10	crash 0.702	error 0.624	test 0.508	bug 0.501	add 0.101

Table 5-5. Extragear/Graphics topics and the related terms for each topic.

Topic #	Topics Terms				
1	update	changelog	screenshots	version	messages
	1.000	0.004	0.003	0.002	0.002
2	polish	code	api	layout	header
	1.000	0.011	0.005	0.002	0.001
3	desktop	file	messages	svn_silent	compile
	0.502	0.502	0.501	0.495	0.009
4	compile	fix	layout	crash	error
	0.902	0.815	0.052	0.051	0.037
5	fix	compile	layout	header	typo
	0.874	0.829	0.121	0.117	0.088
6	port	qt4	digikam	replace	remove
	0.888	0.520	0.380	0.370	0.101
7	use	il8n	code	add	trunk
	0.412	0.371	0.291	0.269	0.238
8	typo	layout	header	fix	add
	0.988	0.079	0.078	0.069	0.031
9	add	digikam	new	missing	image
	0.580	0.521	0.233	0.202	0.183
10	digikam	layout	optimize	missing	add
	0.726	0.663	0.106	0.091	0.058

Table 5-6. OSG topics and the related terms for each topic.

Topic #	Topics Terms				
1	wrappers	updated	changelog	release	authors
	0.782	0.618	0.051	0.042	0.020
2	warnings	fix	typo	test	build
	0.732	0.658	0.125	0.087	0.041
3	release	wrappers	dev	changelog	authors
	0.970	0.119	0.105	0.104	0.083
4	osg_info	osg::notify	converted	redundant	spaces
	0.592	0.590	0.547	0.033	0.019
5	support	remove	build	fix	huber
	0.795	0.561	0.375	0.253	0.107
6	typo	warnings	fix	test	handling
	0.671	0.523	0.470	0.184	0.058
7	changelog	updated	wrappers	release	huber
	0.754	0.352	0.337	0.183	0.166
8	stephan	changelog	huber	xcode	add
	0.351	0.349	0.346	0.325	0.306
9	remove	build	huber	xcode	stephan
	0.353	0.343	0.333	0.297	0.291
10	compile	build	fix	remove	debug
	0.703	0.469	0.329	0.186	0.162

5.5.4 Discussion

Mainly, this evaluation seeks to demonstrate that:

1) The resulting topics of the current experiments contain at least one topic which is associated with the undertaken adaptive maintenance.

2) The approach proposed supports expressive classifying adaptive commits based on associate activities, such as removing warnings, compiling new code, and cleaning up the code, that were accomplished simultaneously with the main adaptive changes.

To show that this approach is accurately able to identify and label maintenance types topics, the retrieved commits of queries issues are investigated against the commits corpus through conducting two experiments. In the first experiment, and through the use of TAT model, the corpus was queried with 10 queries, where each query was specified in terms of one topic terms, as shown in Table 5-4, Table 5-5, and Table 5-6.

In the second experiment, the 10 queries mentioned previously were fixed using TAM model. By using TAM model, no domain knowledge is needed to formulate those queries. In both experiments, the most common measure in experiments with IR methods was used which is: recall. For a given query q , N_i documents will be inspected in step i . Among these N_i documents the user will identify that $C_i \leq N_i$ of them are actually related to the concept expressed by the query. There are R_i documents considered relevant to the concept. With these numbers the recall for q is defined as follows:

$$Recall = \#of\ correct\ \&\ retrieved\ documents\ (C_i) / \#\ of\ correct\ documents\ (R_i)$$

In these experiments, the main interest is the adaptive commits, and R_i represents the total adaptive commits that were manually discovered, see Table 5-1. If recall is 100%, it means that all the adaptive commits are recovered, though there could be recovered commits that are not adaptive.

In both experiments, the recall was computed for each of the 10 queries, as this measure will help in identifying the topics associated with adaptive maintenance. Figure 5-4, Figure 5-5, and Figure 5-6 show these measures and the comparison between the two experiments queries for all investigated systems. Based on these results, the majority of adaptive commits, more than 70% are given by one specific query in both experiments, namely query (8) related to topic 8 for KOffice, query (6) related to topic 6 for Extragear/Graphics, and query (5) related to topic 5 for OSG. Hence, these results demonstrate the utility of the LSI-based approach in grouping a vast majority of adaptive commits in one cluster. This grouping, in turn, provides strong evidence of the semantic similarity between undergoing adaptive changes. One important thing to remember is that the three mentioned queries contain terms (i.e. port and support) commonly used in the adaptive commits as founded by the previous manual examination. For example, 45.1% of adaptive commits in KOffice contain the term “port”, and 37.3% of adaptive commits in OSG have the term “support”.

Based on term similarities of the best results queries, as shown in Table 5-4, Table 5-5, and Table 5-6, it came into notice that some terms, such as digikam, fix, add, and hubers, return rather poor results since they are strongly similar to other topics and generate many correlations. As well, the frequency of those terms appears to be greater in

non-adaptive commits when compared with adaptive commits as exposed by the manual investigation. For example, the frequency of the term “add” in KOffice adaptive commits is 14.6%, while the frequency is 19.45% in non-adaptive commits. Therefore, this observation can explain why using TAM model helps enhancing the recall values slightly than using all terms in each topic as a query.

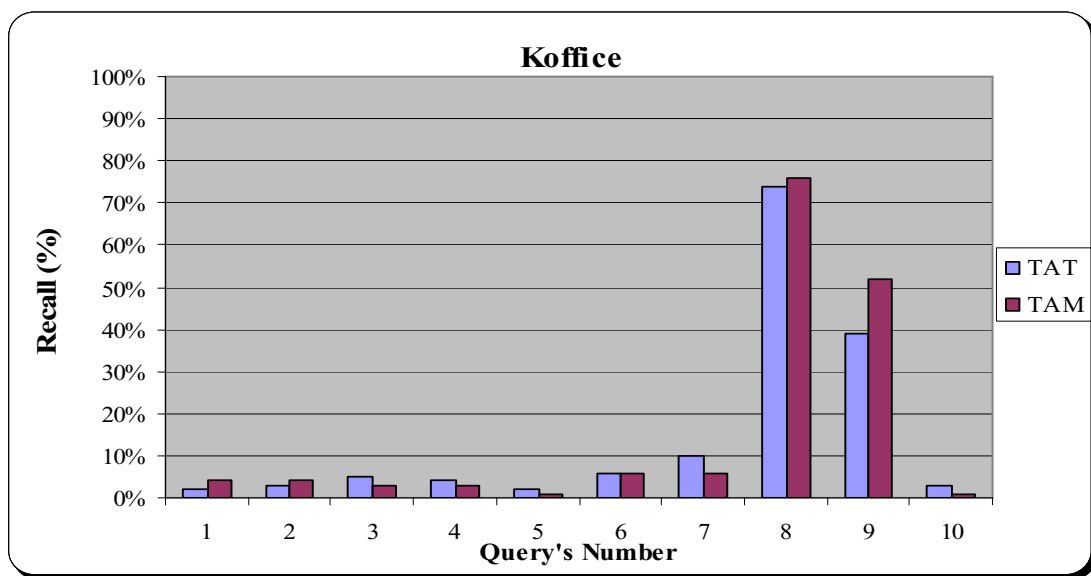


Figure 5-4. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for KOffice.

Furthermore, a set was created that resembles the union of all relevant adaptive commits from the all queries produced by the TAM model. Interestingly, as shown in Table 5-7, this set consists of nearly 90% of adaptive commits that were manually harvested for every investigated system. Thus, applying the LSI in commit corpus is a precious approach as an infrastructure to automate the identification of adaptive maintenance changes. The main point here is to lower the cost and save the developer’s time when identifying the adaptive commits for large scale systems. Using this approach,

to identify adaptive commits, the developer will only need to search the commits most relevant to the executed queries instead of investigating all commits in the log file. An example of such time saving is: when looking for adaptive commits of KOffice, developers search about 2000 commits instead of searching nearly 36,000 commits.

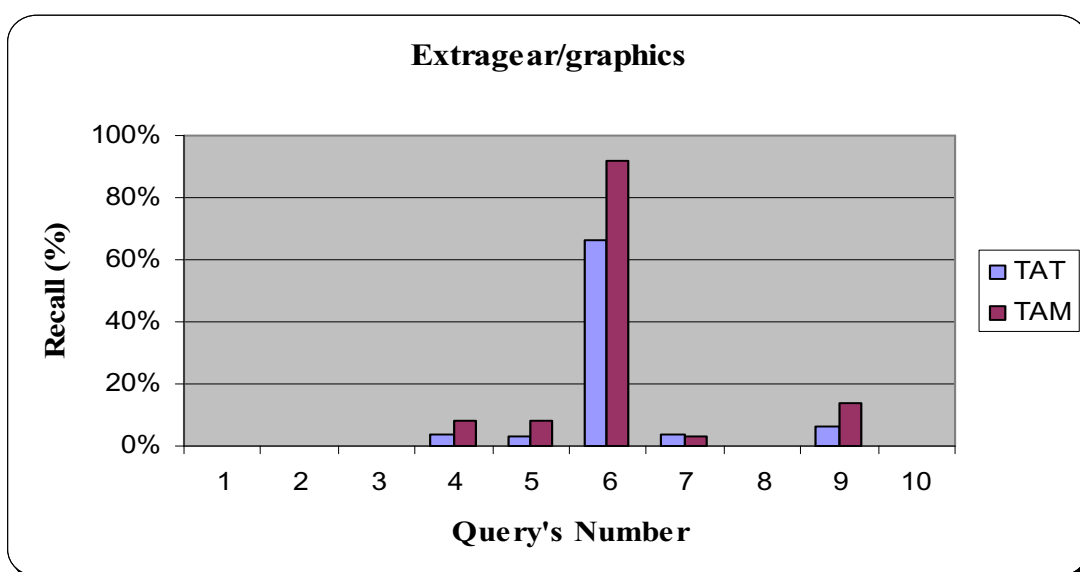


Figure 5-5. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for Extragear/Graphics.

In an attempt to get a zoomed-in picture, interesting commits returned by the other queries were examined. Here, the undertaken tasks that supported the adaptive changes maintained by each of these commits were inspected.

The inspection results are hardly surprising. Commits, which share same supported tasks, were returned by the same query. An example of this is in Extragear/Graphics experiment where all adaptive commits that were retrieved as relevant commits for query number four (in topic 4, “compile” is the most relevant term), mainly did compiling the maintained code against Qt4.

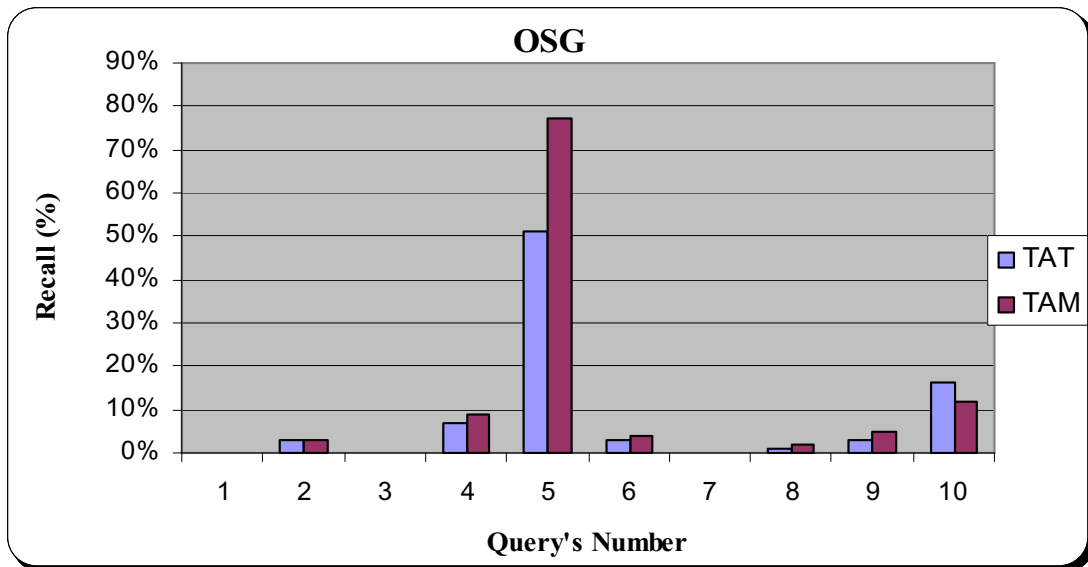


Figure 5-6. Recall(%) of each query, where query number (i) is formatted from topic number(i), using TAT and TAM models for OSG.

This inspection is viewed as a very important and positive result. As a conclusion and based on the results, LSI-based approach is able to significantly, reasonably, and accurately classify adaptive commits based on associate activities, which support the undertaken adaptive changes.

Table 5-7. The size of the union set reported as a ratio of the total discovered adaptive commits.

Systems	Size of Union Set (%)
KOffice	90.1%
Extragear/Graphics	93.7%
OSG	87.3%

5.5.5 Threats to Validity

There are some threats that may affect the validity of this work's results and the ability to generalize obtained results for the current experiments. One of these issues is the style of performing committing, that is, most of developers don't follow a standard way when committing their modifications. Efforts were done to minimize this issue by selecting open source systems that followed good practices of version control and commits.

Moreover, the same commit may relate to many different types of maintenance, and some adaptive maintenance tasks are accomplished via several commits. Another issue which affects the results is the contents of commits, some developers describe poorly what they modified, therefore, some commits contain ambiguous information which adversely affects the categorization process, and hence, systems for the conducted experiments were chosen that are prime examples of well committed open-source. For the future, it is planned to conduct user studies that would aim at statistically evaluating key features of this approach.

5.6 Summary

This chapter presented a new technique for categorizing repository commits based on maintenance types using an IR method, latent semantic indexing (LSI). The proposed approach up till now is employed to recognize the adaptive commits interest in the change log file, and to answer the following questions:

1. What are the main tasks of Software Engineering being accomplished in adaptive commits topics?

2. Which Software Engineering tasks are being accomplished in each topic produced?
3. How can the extracted clusters help in software evolution?

Two variants of the commit location technique using LSI are presented. One, based on all terms of each topic (TAT model) as a separate query and the other based on generating query from specific terms of each topic using Term Average Model (TAM).

The LSI-based approach was evaluated with regard to a collection of commits from popular large open source projects. The evaluation results illustrate the ability of LSI to construct commit clusters containing at least one topic related to adaptive maintenance. The results show that the approach accurately retrieves relevant adaptive commits. When querying the commits available in a version control system, it achieves nearly 90% recall.

By analyzing the retrieved adaptive commits, the use of LSI for adaptive commits identification presents several advantages. The approach is able to classify adaptive commits derived from associate tasks that support undertaken adaptive maintenance. The method is almost as easy and flexible to save time and increase the quality of recognizing adaptive commits for large-scale systems.

CHAPTER 6

Source Code Query Assistant Builder

In Software Engineering, the performance of using IR methods for searching source code artifacts depends significantly on the textual query, and its correlation to the text enclosed in the software artifacts [Smart et al. 2008, Haiduc 2011, Haiduc et al. 2013]. Determining what the best words that must be used in a specific query is a nontrivial difficulty and it requires a full knowledge of the vocabulary of the software artifacts to be searched.

During searching source code artifacts, the developer initiates a query based on his understanding of the current task. Subsequently, he investigates the retrieved results and decides whether they are relevant or not. When the retrieved list contents are not relevant to the task, then typically, it is reformulated.

The reformulation query process is often as hard and time consuming as writing the first query. However, developers could benefit from retrieval tools particularly when such knowledge is missing, or when the experienced developers are missed, or when the system is large and complex [Haiduc et al. 2010].

This way of querying suffers from two main limitations. The first one is that it requires developers with a-priori well knowledge regarding the intended software artifacts. The second limitation is that no significant attention has been paid to the dependencies among the query terms and source code artifacts terms [Smart et al. 2008].

Generally, solutions for software searching techniques have been studied broadly by researchers. They have devised and experimented with an extensive spectrum of approaches in order to efficiently search and extract important, significant, and meaningful information from software artifacts. Traditionally, the search process was performed through the use of text or expression matching [Marcus et al. 2004]. Later on, developers used complex techniques that involved the use of IR approaches. However, the approaches and tools that discover semantic relations between words in the English language [Gyongyi and Garcia-Molina 2005, Gleich et al. 2010], have a somewhat limited capacity in terms of identifying semantically related words in software (synonymy and polysemy) [Sridhara et al. 2008].

Most often, developers tend to use IR (Text based search) to help in facilitating their tasks when they are looking into studying and understanding the artifacts of software for maintenance purposes. The authors in [Castro-Herrera et al. 2009], employed a text retrieval approach for software requirements analysis. On the other hand, in [Marcus et al. 2004], the authors used text retrieval for the process of concept/feature location. The text retrieval in [Marcus and Maletic 2003], was for this case used to recover the traceability link. For more details, the reader is referred to the publications by Haiduc [Haiduc et al. 2010, Haiduc et al. 2010, Haiduc et al. 2013] and Marcus [Marcus et al. 2004].

As mentioned earlier, searching software artifacts mainly depends on the quality of the query [Haiduc et al. 2013]. That is, the queries that will be used to search a code for a specific concept or feature must be formulated accurately and precisely in order to

be able to describe the intended searching goal. Moreover, Searching software artifacts is time consuming for developers; this is due to the fact that choosing the terms that would best describe a certain query consumes a great amount of developer's time. Subsequently, the query is run, and then the retrieved ranked list is investigated to decide on its level of relevance to the task on hand. In the event, that the list is judged not be relevant, and then this mandates the reformulation of the first query, and afterwards for that to be run again.

The main goal is to overcome this issue by producing a tool to assist with the creation of queries for any software artifacts. Introduced is an efficient tool that has the capability to semi-automatically generate a query that best describes the feature or concept that needs to be updated based on the code.

The Query Builder Assistant (QueBA) is a tool that is intended to assist with the creation of queries for a corpus. The Query Assistant leverages the names of documents, in most cases functions names, and synonyms, provided by WordNet¹⁰. This tool attempts to utilize information from both the problem domain and solution domain in order to provide better information about words/terms in the corpus.

6.1 Preprocessing Steps

The algorithm used by the QueBA tool is very simple and is executed over two steps:

- a. Preprocessing
 - b. Term look up
-

¹⁰ <http://wordnet.princeton.edu>

Preprocessing is initiated by reading a document containing all the names of the documents (functions names) used to build the corpus. The document names are then split into multiple words based upon camel casing and/or underscoring separation, and acronym capitalizations. Each word is stored inside a set containing all of the words from the split document name, and that set is then placed within a list for use at a later time.

The second step takes place in two phases, however; both occur when the user enters a term or terms that are contained within the names of the documents. The two phases are:

1. Word document co-occurring terms.
2. Synonyms looking up.

The terms entered by the user, are then added into a set, the term or set of terms are then compared to each document name within the list of documents. All other terms which co-occur with all terms in the users entered are then added to a list of possibly related terms, and those terms with additional information are then displayed for the user.

Synonyms looking up are done using WordNet to provide synonym associations. Each of the words entered by the user is looked up individually within WordNet, consequently separate lists of synonyms are provided for each term entered. A point of concern here, is the fact that the synonyms are not limited to those within the names of the documents, which may be a surprise factor for some of the users

6.2 Algorithm Pseudo-Code

Preprocessing

- Read in all functions names.
- Split functions names into sets of individual words and acronyms and store each into a list *L*.

Look Up Algorithm

- Given set *S* of user selected terms within the corpus, a list of term sets *I*, a set of terms *T* and a map *M* of terms and a list of terms.
- For-Each *splitFunctionSet* in *L*:
 - If *splitFunctionSet* contains *S*
 - Append *splitFunctionSet* into *I*.
 - For-Each *term* in *T*:
 - insert *term* into *T*
- For-Each *term* in *S*:
 - Query WordNet for synonyms of *term* set
 - Insert *term* and the list of synonyms into map *M* where *term* is the key and synonyms are value.

Figure 6-1. QueBA algorithm pseudo-code

6.3 Tool Program Setup

Requirements for compilation are:

- *Qt Creator 2.5.2 or later.*
- *Qt Version 4.8.1 or later.*
- *Change the location within the source code SQLite3 version of WordNet (file name: wordnet30.sqlite) which is with the provided source code, and located within file: WordNetManager.cpp and uses the variable defaultLocationOfWordNet to the location of the SQLite3 WordNet on your computer.*

QueBA is applicable to any Software Engineering tasks that rely on code search, and it is made up of several steps that are described below.

6.4 Tool Usage Instructions

Data input format is expected to be a *text file* containing one function/document name per line as shown in Figure 6-2.

An entire document could be given as a single line and the tool should have the same exact pattern of performance. Nonetheless, given that this tool is meant to handle function's names, it has yet to be tested. For example, the following is a function line from a JEdit4.3 corpus:

```
org.gjt.sp.jedit.gui.AbbrevEditor.AbbrevEditor()
```

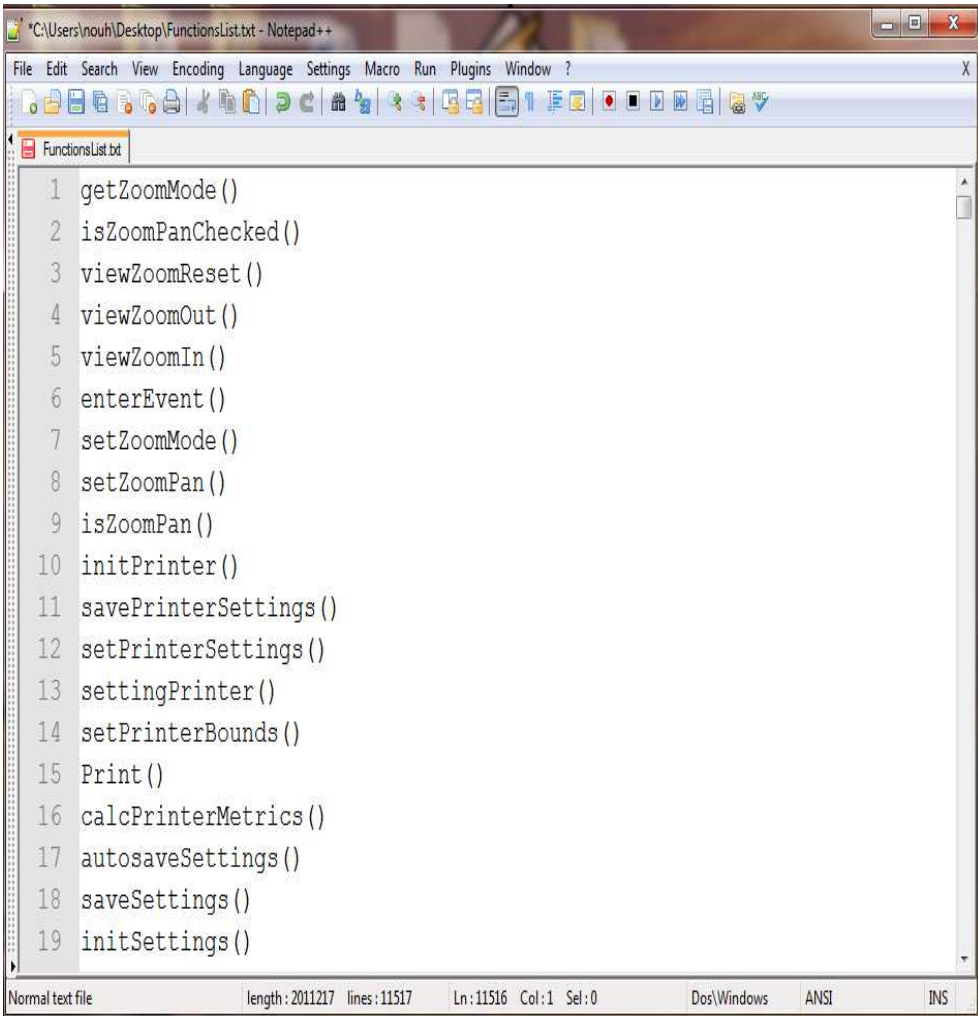
In the line shown above, the following words will be extracted from the function name: abbrev editor. Every after the last '.' but before the first '(' ,or end of line is

considered to be the function name. The same holds true for C++ functions, however; rather than using ‘.’ the scope operator ‘::’ is used instead.

If either a dot or a double colon not present is in the line every word is hence considered to be part of the function name. For example:

- `void MyAbbrevEditor::AbbrevEditor()`

Will yield the terms: abbrev editor, and skip the other terms.



```
1 getZoomMode()  
2 isZoomPanChecked()  
3 viewZoomReset()  
4 viewZoomOut()  
5 viewZoomIn()  
6 enterEvent()  
7 setZoomMode()  
8 setZoomPan()  
9 isZoomPan()  
10 initPrinter()  
11 savePrinterSettings()  
12 setPrinterSettings()  
13 settingPrinter()  
14 setPrinterBounds()  
15 Print()  
16 calcPrinterMetrics()  
17 autosaveSettings()  
18 saveSettings()  
19 initSettings()
```

Figure 6-2. A snapshot of an input text file for a list of code function’s names.

6.5 Tool Interface Components Description

After compiling and launching the application, the field's display looked as illustrated in Figure 6-3, (minus the descriptive text). Moreover, QueBA, displays a friendly user interface that contains the following components:

6.5.1 File Menu

The file menu is comprised of the menu option “Load Info...”, which allows the user to select the data to be loaded onto the system. Loading more than one document filled with functions names will result in failure, as it has yet to be implemented. Therefore, it is necessary to close and re-launch the application in order to use the tool for more than one system.

6.5.2 User Word Entries

This is where once the terms are loaded the user can enter his or her terms. If the term appears in a bold and red format, it indicates that the term is not within the corpus. In the case that the term is in black, this would indicate that the term is within the corpus and the rest of the fields will be updated accordingly.

Upon running the tool for multiple trials, a bug was found. When in the process of entering terms, if the terms are all on a single line, then everything will work correctly and smoothly. In contrast however; if the terms are divided on multiple lines, then only the terms on the currently selected line will be displayed. Hence, the best advice is keep entries on a single line.

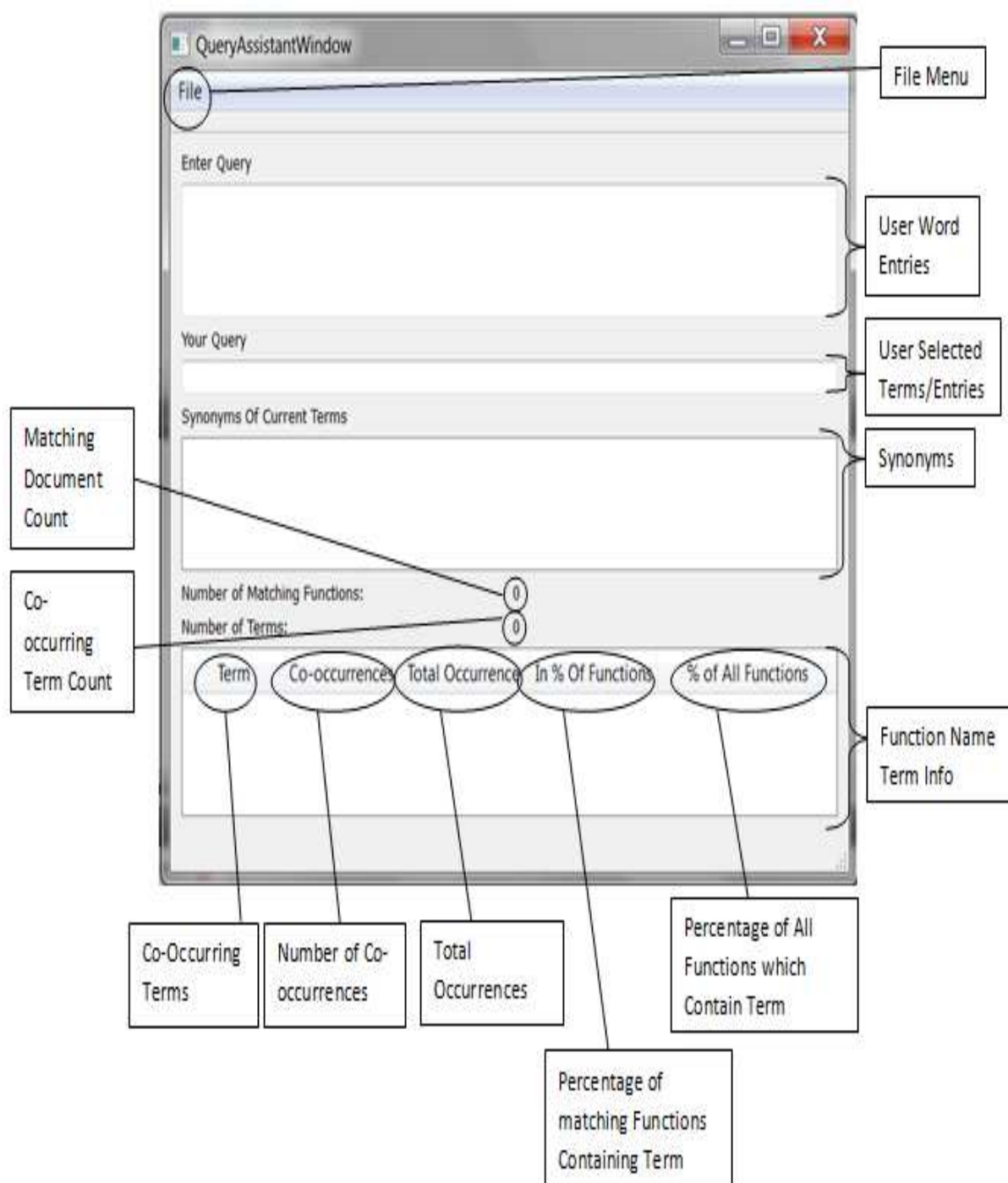


Figure 6-3. Tool interface components.

When this field changes (or one of the multiple lines are selected - see previous note) the following updates occur. The User Selected Terms/Entries will show all terms that are entered into the current User Word Entries and remove any of those which were previously within it.

Later on, the synonyms will show lists of synonyms for each term and the Function Name Term Info will be updated with new information.

6.5.3 User Selected Terms/Entries

This component contains terms/synonyms which are double clicked on by the user or entered into the User Word Entries. This field cannot be edited directly, but it can be copied from. Moreover, this field cannot be cleared.

6.5.4 Synonyms List

Integrated into the synonyms list are the synonyms of words entered into the User Word Entries field. The terms entered are not guaranteed to be in the list of words in the split functions names.

Also, there is no stemming or word morphing to compensate for issues like plural words, so those words will not display synonyms.

6.5.5 Matching Document Count

This component illustrates the number of matching documents that contain all words entered by the user. This box shows the relevant methods that contain the entered term/word as a part of it. In other words, this component demonstrates the methods that

are relevant to the entered term. The retrieved methods definitely deal, implement, or use the entered user term.

6.5.6 Co-occurring Term Count

The number of unique terms which co-occur with the user entered terms in document names. Therefore, the user can accurately pinpoint which methods are named with a composite name (composed of a unique term with the entered term).

6.5.7 Function Name Terms Info

It is a table which displays information about particular terms within the corpus. When information is first loaded into Query Assistant, all terms are displayed and all values have a zero value, and they are not updated until the user enters information.

6.5.8 Co-occurring Terms

It displays a term which co-occurs in a document name with the entire user supplied terms. Each field within this column when double clicked will add its value into the User Selected Terms/Entries field.

6.5.9 Number of Co-occurrences and Total Occurrences

The number of occurrences box shows the total number of times a term occurs within all document names (including the case where if there is a function name that consists of the entered term only as its name). Total Occurrences means the number of times a particular term co-occurs with all terms in the document names.

6.5.10 Percentage of Matching Functions Containing Term

This value can be calculated according to the following formula:

$$\text{Percentage of matching} = ((\text{Number of Co-occurrences/ Matching Documents Count}) * 100).$$

6.5.11 Percentage of All Functions Containing Term

This percentage is computed according to the following formula:

$$\text{Percentage of All Functions} = ((\text{Total Occurrences / Matching Documents Loaded}) * 100).$$

6.6 Related Work

Marcus et al. [Marcus et al.] have used LSI in order to find out the terms of most relevance to the query from the source code corpus, and include them in the query. They have utilized different formats for each query, starting from choosing a single word or phrase.

In [Shepherd et al. 2007], the authors enlarge and expand search queries with terms that are semantically related (e.g., synonyms and abbreviations). In [Holmes and Murphy 2005], the authors utilize and make use of the context in which query words are found in the source code to extract synonyms, antonyms, abbreviations and related words. A code search tool that expands search queries with alternative words learned from verb-direct object pairs was presented in [Gyongyi and Garcia-Molina].

Other approaches make use of exterior sources of information in order to determine the related words that should be included in the query. Algorithms from web

mining are employed in [Reiss 2009, Haiduc 2011] to identify web documents relevant to the query. In order to improve query accuracy, researchers have used query reformulation, either by query reduction [Mandelin et al. 2005], or query expansion [Carpineto and Romano 2012] approaches.

In [Haiduc 2011], the authors present a tool that can automatically detect and measure the quality of the query, along with its implications in IR-based concept location. The authors extended their work in [Haiduc et al. 2013], and were able to present an automatic query reformulation approach (*Refoqus*). This approach focuses on the employment of the various strategies, so that eventually the best one for each query is selected. The goal of the *Refoqus* reformulation tool, is to define a new query starting from the initial one until discovering the best one.

In [Haiduc et al. 2012], the authors present a novel pre-retrieval metric, which is used as a sign of the quality of a query, the metric does what was previously mentioned by measuring the specificity of user query terms. The authors used different measurements in order to classify each query based on its terms quality. In their evaluation, they have conducted an empirical study about their metric, and they have concluded that their proposed metric can accurately predict the effort for text retrieval-based concept location, as well as it being able to outperform all other techniques from the field of natural language document retrieval.

In [Starke et al. 2009], the authors have studied how developers search source code when performing corrective tasks on an unfamiliar system. Their findings indicate

that after several reformulations, some developers were still unable to locate the information they needed.

On the other hand, in [Holmes and Murphy 2005], the authors presented a semi-automated (i.e., interactive) approach for reformulating the queries. This approach requires developers to choose and select the terms from the retrieved list after running the first query. The tool allows developers to re-run the query automatically. In other words, the developer investigates the feedback taken from the retrieved list with the ultimate goal of formulating a further meaningful query that is closer to the relevant documents.

6.7 Tool Evaluation

The proposed approach was evaluated in the context of IR-based feature location in the source code. The results on two systems show that the approach presented herein, is able to correctly suggest relevant terms that are positively associated with the task on hand, and must be considered and added to the query.

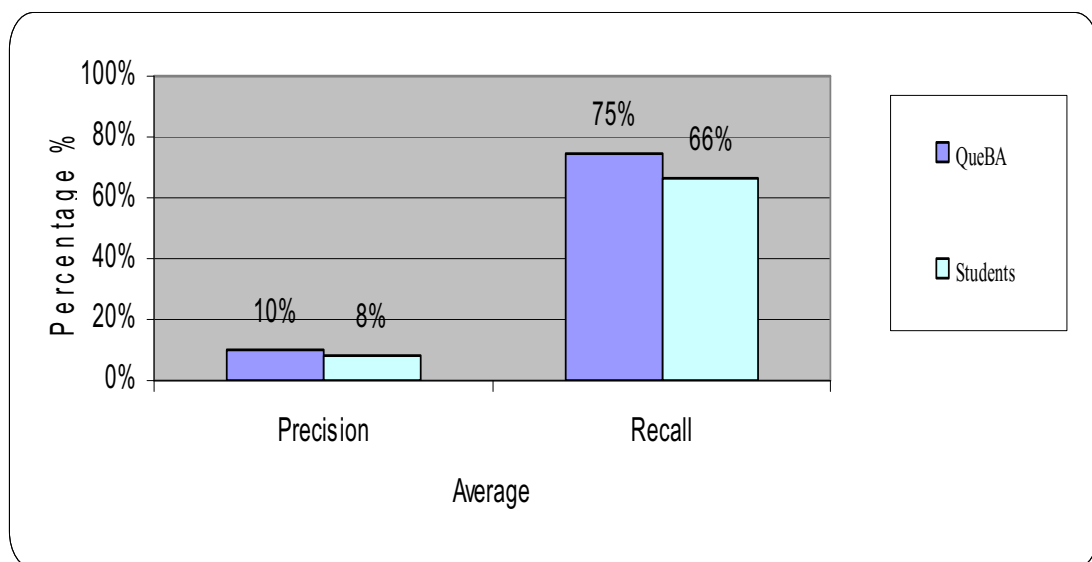
Two main measures were utilized for evaluating the effectiveness of retrieval, precision, and recall. Two PhD students were chosen to formulate a query for 36 features; 25 features for a Qt system, and 11 for a HippoDraw system.

Then, for each feature, the QueBA was applied to specify which terms must be added to the query that best describes it. Afterwards, LSI was used to run the query over the corpus. Finally, the retrieved ranked list of each feature was compared against the list that was retrieved using the student's query. The results, as illustrated in Figure 6-4 and Figure 6-5, show that the usage of QueBA improved on average 75% of all queries results.

Table 6-1. Details of the corpora that were used in the experimental study.

Systems	Number of Parsed Documents/Methods	Number of Investigated features	Vocabulary Size
HippoDraw 1.21.3	3,706	11	6,803
Qt 4.4.3	70,871	25	91,187

Table 6-1, describes the characteristics of HippoDraw and Qt in the context of their usage for the purpose of this experiment. It is clear that Qt is a much larger system in all aspects. The method level of granularity is chosen in both studies. Here, the same methodology described in chapter 3 was adopted in ranking the relevant parts of the source code with respect to the user query, with different dimensionality reduction factors chosen for each study.

**Figure 6-4. Average of recall results for the Qt experiments.**

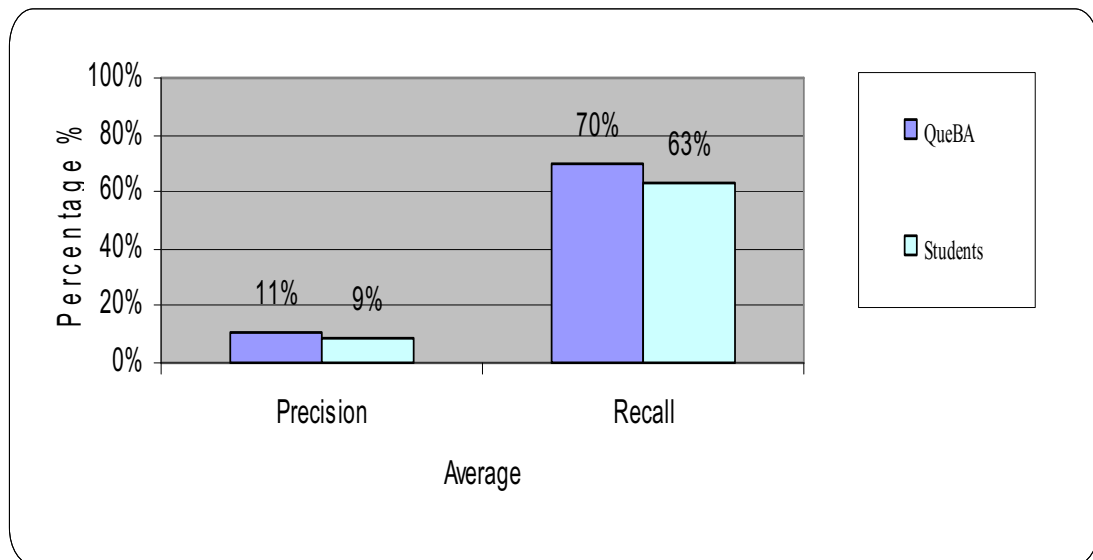


Figure 6-5. Average of recall results for the HippoDraw experiments.

Table 6-2, shows the 9 relevant methods for the query “update zoom mode”, that describe the feature update mode as shown.

Experiments were conducted for the feature using two queries. The first of these queries was using the student’s query which had the terms “set zoom mode change modify update”. The second one was with using the suggested terms from QueBA, which were “reset checked zoom mode pan view”. Both of the two retrieved ranked lists were inspected, and QueBA proved itself to have the capability to give more accurate suggested terms, that being reflected on the recall and precision of the query results.

Moreover, the usage of the query that was generated using QueBA decreases the total effort needed from developers to find all relevant methods given that it ranks the relevant methods for the current task (feature) higher than when the student’s query is used.

Table 6-2. List of all relevant methods/functions for modify mode feature.

Functions List
getZoomMode()
isZoomPanChecked()
viewZoomReset()
viewZoomOut()
viewZoomIn()
enterEvent()
setZoomMode()
setZoomPan()
isZoomPan()

QueBA displays for the user the entered terms, the total number of time that the entered term was mentioned, or used within function's names all across the code. That is, for example for the feature "update zoom", and upon entering the term zoom for example, the QueBA retrieve the terms; get, pan, view, set, etc.

For the term set, which was suggested by the students query, the QueBA shows that this term is commonly used as a part of a lot of functions names. Therefore, including it in the query will definitely rank irrelevant methods in a high position. This is clearly shown and reflected on the results. When the developer realizes the common usage of such a term, he/ she must exclude such a term from being a part of his/ her query.

6.8 Summary

In this chapter, a novel tool was presented for the aim of generating based code queries (QueBA). The main goal of the tool is to help, assist, and support with the creation of queries for a corpus. The QueBA expresses the names of documents, in most cases these are functions names, and synonyms, provided by WordNet. The tool employs the information extracted from both the problem domain and solution domain efficiently in order to provide better information about words/terms in the corpus.

QueBA was evaluated in the context of IR-based feature location over source code. Two PhD students were used to formulate queries for 36 features. The results for two open source systems (Qt and HippoDraw), indicate that the proposed tool is able to correctly suggest relevant terms that are accurately relevant to the current task and describe it precisely and perfectly.

QueBA improved 75% of all queries results on average. Moreover, the results show that with using QueBA, developers save a lot of the time needed to formulate any query. Furthermore, with using the query that was generated by QueBA, the relevant methods for any current task (feature), are ranked more correctly and in relatively higher positions. This is translated to the fact that the amount of effort required by developers to searching and investigating the ranked retrieved list is decreased.

CHAPTER 7

An Environment to Conduct Experiments in Information Retrieval for Software Engineering

In Software Engineering, it is hard to use IR methods for conducting, reproducing, comparing, and generalizing the results of case studies involving feature and concept location, detection of duplicate bug reports, and traceability links retrieval, etc. The main reasons behind that are due to issues such as, lack or inappropriateness of different datasets, lack of freely available implementation, etc.

To address these issues, we propose a solution for creating, conducting, and sharing experiments in feature and concept location, detection of duplicate bug reports, traceability links uncovering, etc., based on TraceLab framework.

In this chapter, we present a new component namely LSI, that we implemented and added to TraceLab environment. This new component allows and facilitates rapid advancements in feature/concept location and traceability research, etc., LSI component enable researchers to create new experiments or conduct new researches in the form of TraceLab templates, and compare them with existing ones using the same datasets and the same metrics [Alhindawi et al. 2013]. More details about TraceLab and about LSI component are presented in the following sub-sections.

7.1 TraceLab Overview

TraceLab [Dit et al. 2012, Keenan et al. 2012] is an environment where traceability, feature and concept locations experiments can be easily constructed, and reproduced all while reusable components are being used. It uses a visual modeling environment to set up experiments with the components. TraceLab also has the ability to allow for repeating experiments by other researchers with ease. TraceLab was created at DePaul University with collaborating partners at Kent State University, University of Kentucky, and the College of William and Mary.

The work presented herein demonstrates the usage of TraceLab components in running experiments, similar to those found in [Antoniol et al. 2002, Marcus and Maletic 2003, Dit et al. 2012], of using the IR method, more specifically, LSI (as plug-in component), for enhancing source code searching, feature/concept location, and traceability links uncovering, etc.

The objective of this discussion is to show how TraceLab components and namely LSI component can be used, and how the reusable component can be utilized to build, create, and share experiments.

Furthermore, this chapter presents and explains the preprocessing steps that taken to generate a set of traceability data in a semi-automated manner via frequent itemset mining. SDML¹¹ members utilized some manually generated information along with frequent itemset mining to uncover a set of traceability links for a specific type of

¹¹ <http://www.sdml.info/>

software maintenance task. What is investigated here is a particular adaptive maintenance task that involves the migration of an API.

The results of the experiment align well with the original findings and form a basis for running a variety of experiments to test the proposed hypothesis on different parameters.

7.2 TraceLab Features

Several features come together to form TraceLab; in this chapter a brief description is offered about the features found to be most vital. This chapter will also discuss the components of TraceLab, the method for working with these components, and the features of running an experiment.

7.2.1 Components

TraceLab components are high-quality software components. Therefore, the developers of TraceLab designed and implemented the components in such a way that can be used in many different programs repeatedly. Furthermore, component-based usability testing was taken into consideration.

To make the reuse of components easier and simpler, TraceLab's components library provides a hierarchy that is based around user defined categories. Moreover, the TraceLab components library explorer permits users to search for a specific component (filtering). Figure 7-1 illustrates the component's library in TraceLab.

Generally, A TraceLab components can be written using any programming language that based on memory-managed. Examples include C#, Visual Basic, or Java.

Strings, integers, arrays, lists, and community defined data structures (trace matrices, artifact lists, and terms dictionaries), are all examples of the datatypes that TraceLab supports.

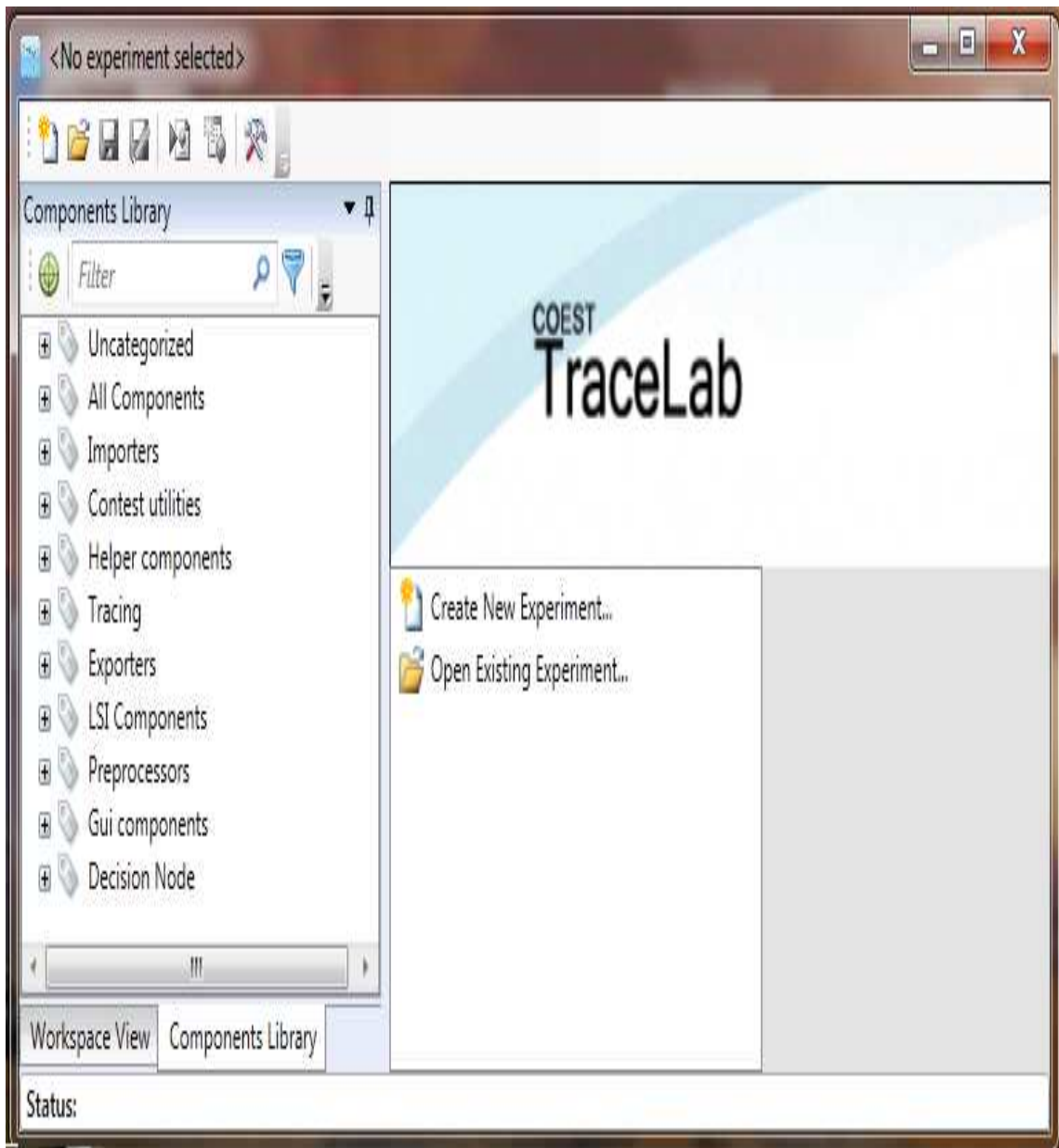


Figure 7-1. Home page for TraceLab showing the component's library.

7.2.2 Working with Components

Data is swapped between TraceLab components during the experiments via the workspaces which designated for each one, Figure 7-8 shows the different workspaces allocated. Each component that is added to the experiment has specific configurations that must be defined and chosen by the users prior to use, as shown in Figure 7-2.

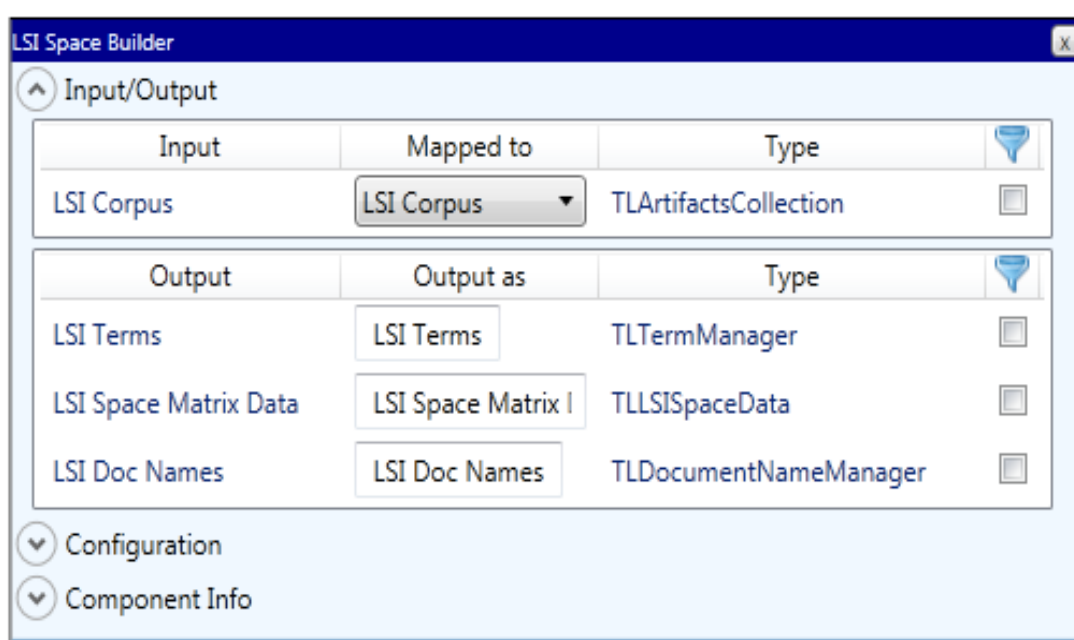


Figure 7-2. LSI Space Builder component.

The user-defined components can be integrated easily into TraceLab. All what is required by the developer is to add meta-data to the main class of the component, then map any imported or exported TraceLab datatypes to the internal data structures. Afterwards, the project is compiled into a .NET assembly, and finally the assembly is copied to a TraceLab component directory.

7.2.3 Running an Experiment

Running the experiment in TraceLab is easy, clear, and comprehensible. Each component in the user's experiment is highlighted by TraceLab at runtime. Moreover, the information (logging) assigned for each component by the user, is output to the screen. This would be in addition to the fact that the present state of the workspace is also restructured and modified dynamically.

7.3 TraceLab Components

For the experiment illustrated within this context, several components were created in order to use LSI [Deerwester et al. 1990], as shown in Figure 7-1 and Figure 7-4. Each of the created components was designed to use the existing TraceLab types. This was done with the intention of facilitating the integration of these components into other experiments as needed.

The components discussed here were written using C# and C++. Following is a brief description of the components that were created.

7.3.1 LSI Space Builder

This component is used to construct the LSI space for a given corpus, as shown in Figure 7-2. For an input, it functions by taking a set of document names and documents, which make up a corpus. The output is the LSI space up to a specified rank, a dictionary of document titles, and a dictionary of vocabulary. The corpus input is a TraceLab type that allows easy preprocessing, such as stop word removal, word splitting, and many others. The LSI Space builder computes TF/IDF implicitly, and it uses LAPACK's

dgesdd [Anderson et al. 1999] function to compute the Singular Value Decomposition (SVD) of the resulting matrix.

7.3.2 LSI Querier

This component is responsible for executing queries on a given LSI space, as can be seen in Figure 7-3. The LSI Querier takes multiple inputs including; the LSI space to query, the dictionary of document titles, the dictionary of vocabulary, and a set of queries to execute against the corpus. The queries input are in the same form as the corpus. Therefore, that preprocessing can be also used for the queries.

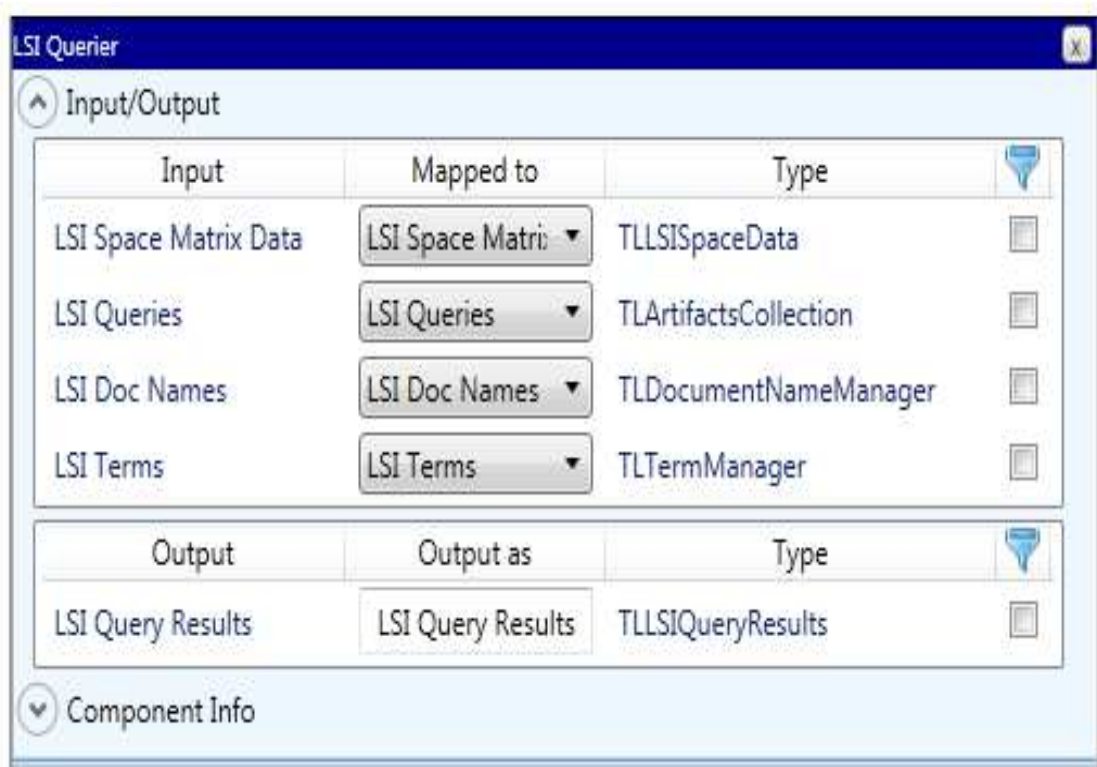


Figure 7-3. LSI Querier component.

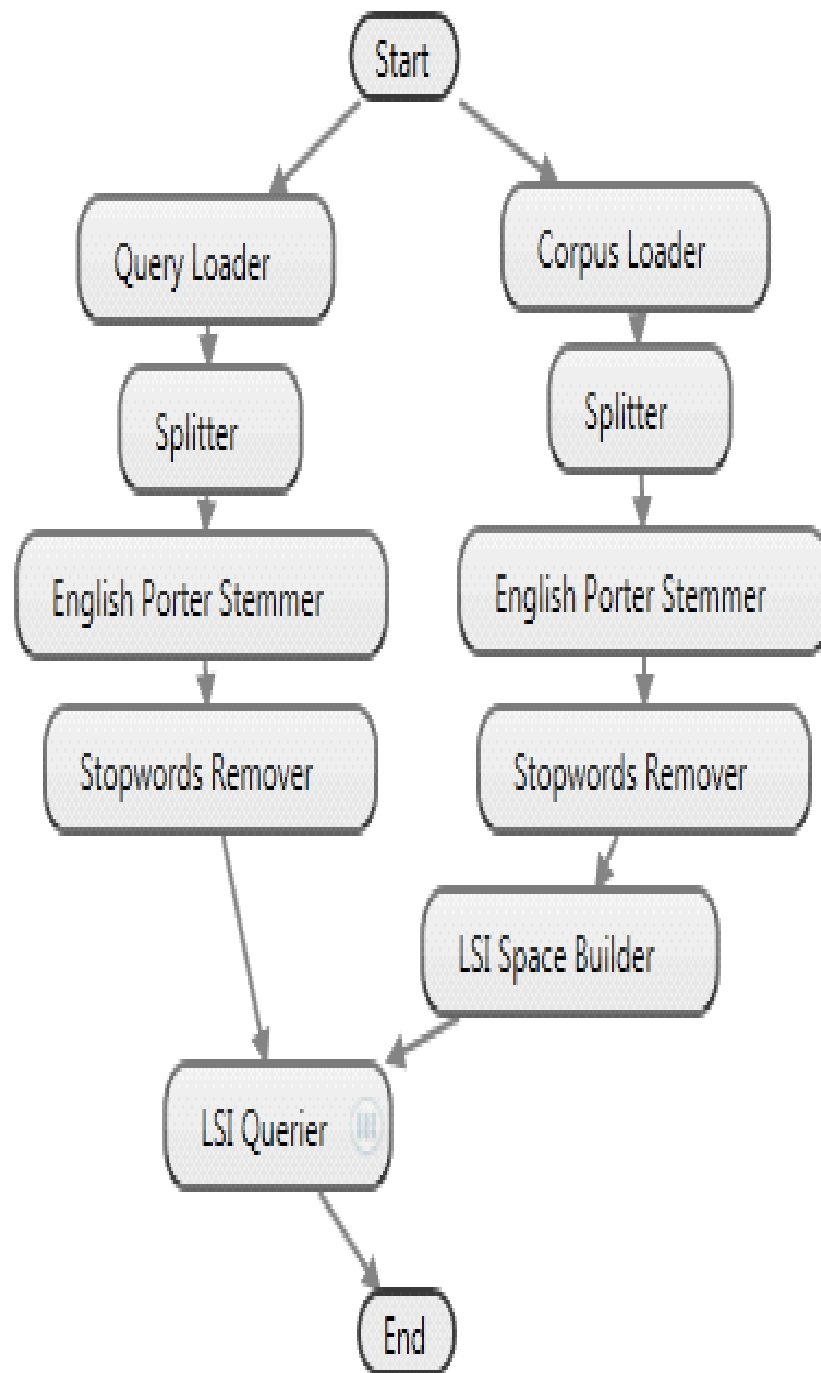


Figure 7-4. An example of experiment set up of how to preprocess a loaded corpus and set of queries to the LSI Space Builder and LSI Querier respectively.

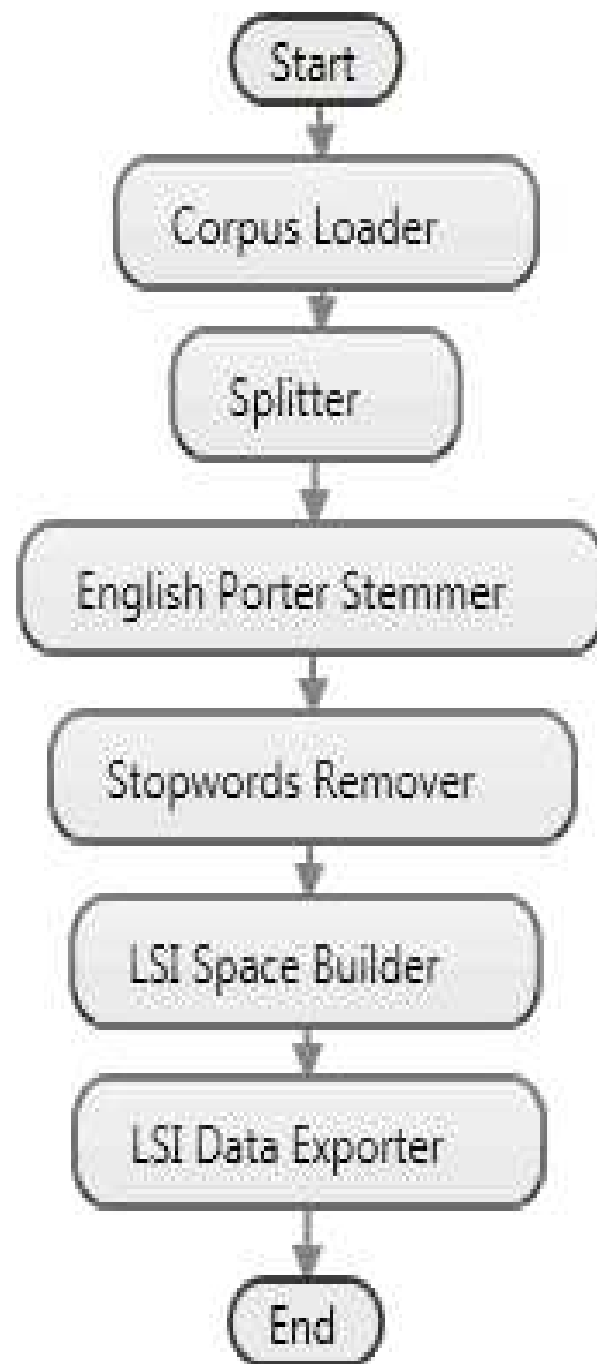


Figure 7-5. An example of experiment set up of how to use the LSI Space Builder with the LSI Data Exporter.

7.3.3 LSI Data Importer

This component imports the LSI space, the dictionary of document titles, and the dictionary of vocabulary from the file system as shown in Figure 7-6. Therefore, the data may conveniently be available for reuse. This allows for multiple different sets of experiments to be run on the same corpus without necessarily having to rebuild the LSI space every time.

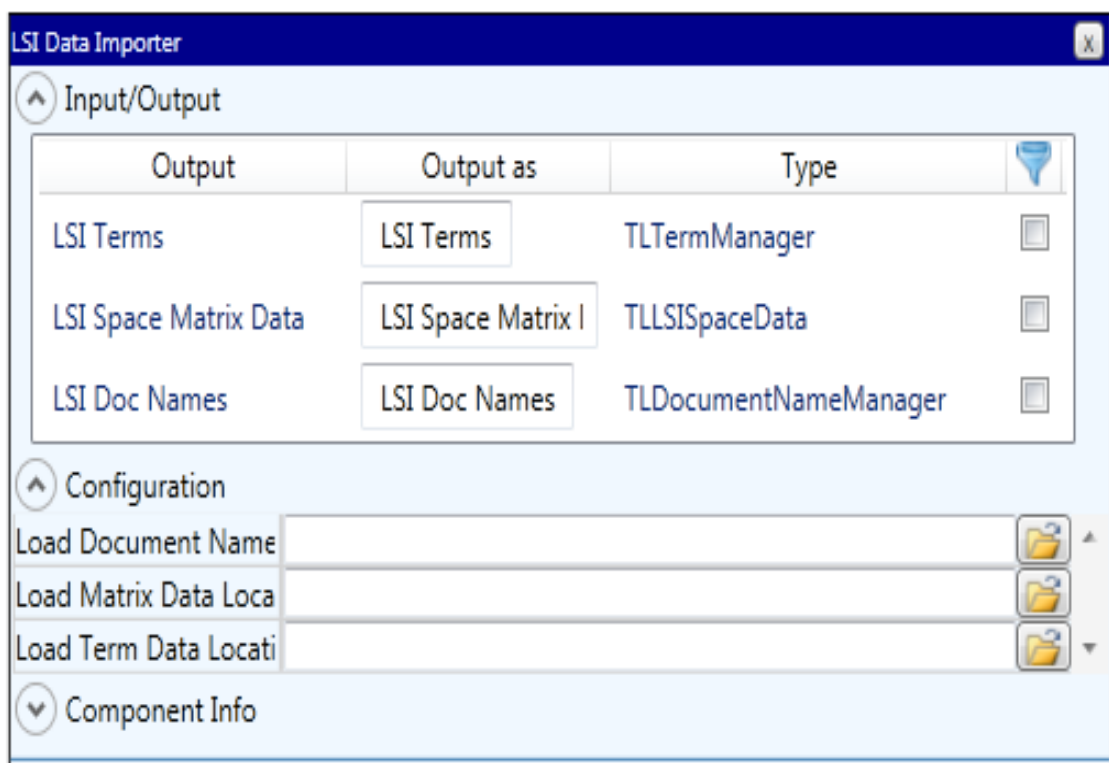


Figure 7-6. LSI Data Importer component.

7.3.4 LSI Data Exporter

As for this component, it exports resulting LSI data onto the file system for reuse with the LSI Querier. The LSI data exporter takes the output from an LSI Space Builder as input and allows the LSI space, dictionary of document titles, and dictionary of

vocabulary to be saved onto a specifiable location on the file system as shown in Figure 7-7.

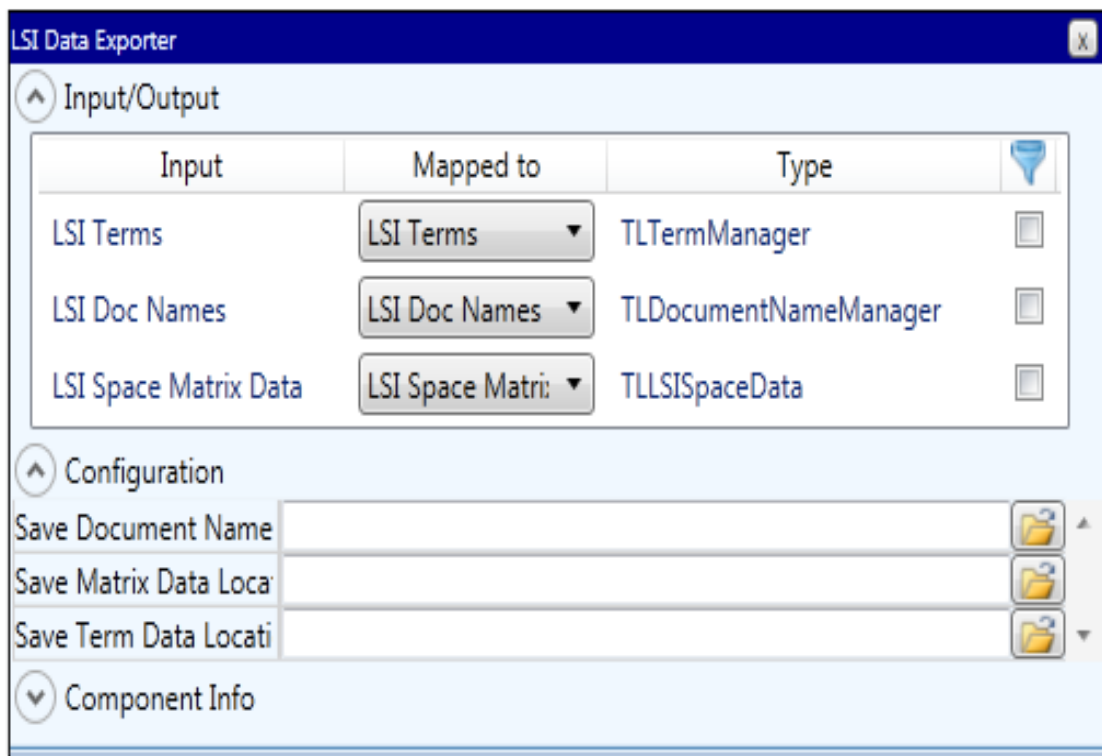


Figure 7-7. LSI Data Exporter component.

For the experiment exhibited here, the examples shown in Figure 7-4 and Figure 7-8 were used to save, reload, and query the resulting LSI data multiple times. Two corpora were created. The first was for the documentation, while the other one was for the function of the system. Later on, the corpus consisting of the documentation was used to query the LSI space built from the function corpus, and vice versa. The ending products of the experiment were the traceability links between the external documentation and the functions of the system.

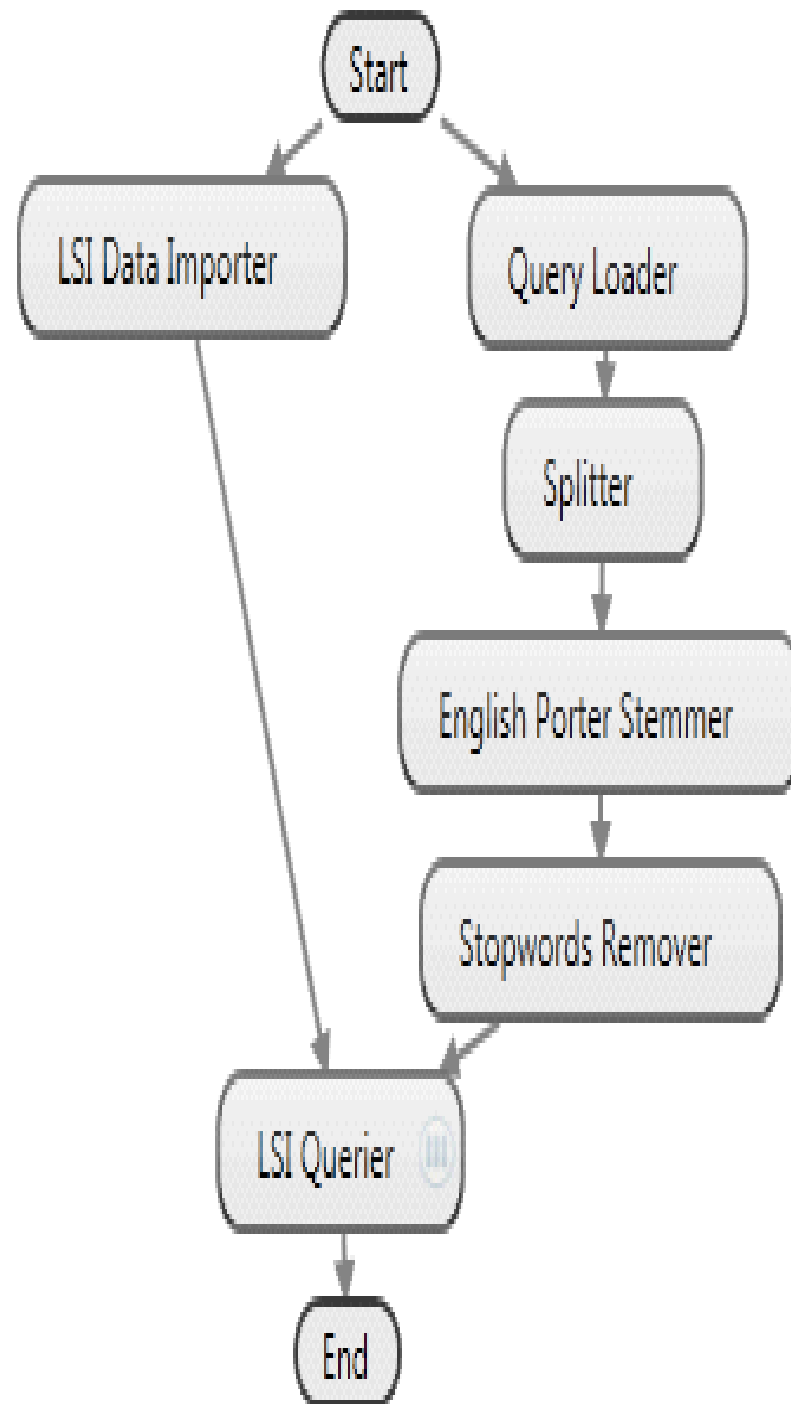


Figure 7-8. An example of an experiment set up of how to use LSI Querier, and LSI Data Importer to query the corpus, which was saved to the file system. The queries are preprocessed using the right side of the graph.

7.4 Retrieval Case Study: Traceability Recovery Process

Antoniol et al. [Antoniol et al. 1999, Antoniol et al. 2002] investigated the use of IR methods to support the traceability recovery process. In particular, they used both a probabilistic method [Antoniol et al. 1999] and a vector space model [Antoniol et al. 2002] to recover links between the source code and the documentation, and between the source code and the requirements. In the same domain, Maletic et al. in [Marcus and Maletic 2003], were able to use LSI to automatically identify such traceability links. Moreover, they argue that using LSI has a slightly better recall value than the approach previously proposed by Antoniol [Antoniol et al. 1999, Antoniol et al. 2002], where LSI helps in reaching a 100% recall value one step before their methods. On the other hand, the precision value is a lot better for LSI when compared with the probabilistic and the VSM methods used in [Antoniol et al. 1999, Antoniol et al. 2002]. A complete comparison between the use of LSI and the one in [Antoniol et al. 1999, Antoniol et al. 2000, Antoniol et al. 2002] with respect to identifying the traceability links is presented in [Marcus and Maletic 2003].

The traceability recovery process presented here is centered on the LSI component [Alhindawi et al. 2013]. However, user input is necessary, in addition to the degree of user involvement depending on the type of source code and the user's task. Recovering the links between source code and documentation supports various Software Engineering tasks [Antoniol et al. 2002, Marcus and Maletic 2003].

Different tasks (along with users) typically require different types of information. For example, there are instances where completeness is important. In other words, the

user needs to recover all the correct links even if that means recovering many incorrect ones at the same time. At other times, precision is preferred and the user restricts the search space so all the recovered links will be correct ones, even if this means not finding all of them. The proposed TraceLab based solution tries to accommodate both needs (individually that is). One way to accommodate the user needs is by offering multiple ways to recover the traceability links [Hammad et al. 2011].

The traceability recovery process is organized in a pipelined architecture; where the output from one phase constitutes the input for the next phase. However, TraceLab supports components to accomplish all of those phases. The user's involvement in the process occurs in the beginning for the selection of the source code and documentation files. This is followed by the user selecting the dimensionality of the LSI subspace. After the LSI subspace is generated, the user determines what the value of threshold will be used in determining the traceability links.

The input data consists of the source code and the external documentation. The golden set consists of the tractability links uncovered in the previous section. In order to construct a corpus that suits LSI, a simple preprocessing of the input texts is required. Both the source and the documentation need to be broken up into the proper granularity to define the documents, which will be represented as vectors [Deerwester et al. 1990, Marcus and Maletic 2003]. Therefore, the source code was split up into documents of different granularity levels (i.e., functions, interfaces, and classes). For external documentations, a paragraph is used as the granularity of a document. Table 7-1 contains

the size of the system, as well as the dimensionality used for the LSI subspace and the determined vocabulary.

The end goal of the conducted experiments here is to uncover traceability between the source code and other artifacts using TraceLab. Therefore, in the evaluation demonstrated herein, a set of experiments were conducted over the same dataset in CHAPTER 5, and the results were validated by comparing them with uncovered links extracted by applying a frequent-pattern mining technique on a set of adaptive commits of KDE/KOffice system.

As shown in Table 7-1, mining adaptive commits of the KDE/KOffice system uncovered 89 non-source code files, which have traceability links at minimum support of three. That is, these identified traceability links were utilized to validate “how well” the TraceLab discovers the existence of traceability links between the source code files and other artifacts.

Table 7-1. Elements of the KDE/KOffice source code documentation and list settings used in the experiments.

KDE/Koffice	Count	Documents
Source Code Files	1057	11492
Non-Source Code Files	89	102
Total # Documents		11594
Vocabulary	12839	-
LSI Dimensionality Used	300	-

Since the number of external documentations is much smaller than the number of source code files, the decision was made to trace the links from the external system documentations to the source code, rather than vice versa. Thus, a typical query will be used to find out which parts of the source code are described by a given external documentation.

Table 7-2 summarizes the results obtained on recovering the traceability links between external documentation and the source code for the KDE/KOffice. The first column (Cosine) represents the threshold value; while the second column (Total links retrieved) covers the total number of recovered links (correct + incorrect); and the last two columns are concerned with the precision and recall for each threshold. Comparing the results with those attained in [Marcus and Maletic 2003], LSI-TraceLab components were proven to enhance the precision and recall results. Figure 7-9 shows a snapshot for the results of running one query sample over 2000 documents.

Table 7-2. Recovered links, recall, and precision using cosine value threshold for KDE/KOffice.

Cosine Threshold	Total Links Retrieved	Precision	Recall
0.60	184	40.76%	84.26%
0.65	133	51.87%	77.52%
0.70	95	57.89%	61.79%

LSITypes.TLISQueryResultsEditor

Query 0

Query Index: 0

Rank: 300

Document Count: 2000

Save Results As...

Similarity Rankings

Similarity Rank	Cos Similarity	Document Name	Document Id
0	0.767615689781165	topBorderStyle 1	1172
1	0.767615689781165	bottomBorderStyle 1	1180
2	0.720701277954431	setTopBorderStyle 2	1105
3	0.720701277954431	setBottomBorderStyle 2	1113
4	0.72034171460426	leftBorderStyle 1	1167
5	0.683298729372566	setLeftBorderStyle 2	1101
6	0.644965209250034	style	1948
7	0.644965209250034	getStyle	805
8	0.639794534156413	setTopBorderStyle	221
9	0.639794534156413	setBottomBorderStyle	225
10	0.63253313979061	setLeftBorderStyle	213
11	0.614716079479444	leftBorderStyle	368
12	0.605715383717049	defaultStyleFormat	1067
13	0.604330452288624	lookup 5	184
14	0.602934833471055	bottomBorderStyle	389
15	0.602934833471055	topBorderStyle	382
16	0.592434994930821	findStyleName	108
17	0.590684451424657	rightBorderStyle 1	1176
18	0.590322116264843	createStyleFromCell	1527
19	0.58958553576187	getStyleManager	806
20	0.579916646622021	setLeftBorderStyle 1	362
21	0.571682558526123	setBottomBorderStyle 1	383
22	0.571682558526123	setTopBorderStyle 1	376
23	0.56053527069535	setRightBorderStyle 2	1109
24	0.55073176846824	setStyle	1073
25	0.53316030005512	slotUser1 1	716
26	0.526691823424533	saveOasisCellStyle	1085
27	0.524874226927642	fillComboBox	713

Figure 7-9. Snapshot for the results of running one query sample (Results with first 2000 documents & LSI Dimensionality =300).

7.5 Summary

This chapter presents how LSI components are implemented and how these components work as a part of TraceLab platform. The results show that the new implementation of LSI as TraceLab plug-in component is very helpful and supports researchers of Software Engineering.

Moreover, in this chapter, to ensure the effectiveness of the new components, an LSI-TraceLab-based experiment was conducted to uncover traceability links. We provided the details for an environment that allows researchers to recover traceability links between external documentation and source code.

A set of experiments was presented and the results validated by comparing them with uncovered links extracted by applying a frequent-pattern mining technique on a set of adaptive commits of KDE/Koffice system.

The results are promising enough to demonstrate LSI-TraceLab as an environment that can be used to conduct feature and concept location research and traceability link uncovering research, and aid the growth in the facilitating software comprehension research fields.

CHAPTER 8

Conclusions and Future Work

The dissertation addresses several research issues that relate to program comprehension. Specifically, it investigates the use of advanced Information Retrieval (IR) and Natural Language Processing (NLP) approaches to problems in software engineering for the problems of feature and concept location, maintenance categorization, and traceability links, for large-scale software systems undergoing maintenance and evolution.

The first issue deals with improving feature and concept location problem. The work presented advances the field by investigating approaches to augment and re-document the source code with different types of abstract behavior information. The hypothesis is that enriching the source code corpus with meaningful descriptive information, and integrating this orthogonal information (semantic and structural) that is extracted from source code, will improve the results of the IR methods for indexing and querying information. Adding this information is a form of supervision added on top of an unsupervised method (i.e., LSI). Generally, apriori knowledge is often used to direct and supervise machine-learning and information-retrieval approaches.

In particular, the work uses Latent Semantic Indexing (LSI), an advanced IR method that has been widely used for indexing and analyzing source code. The source code is augmented with method/function stereotype information. These stereotypes are

automatically reverse engineered from the source code and then added back into the code as comment annotations. Stereotypes describe the role and behavior of methods and functions in the code.

Furthermore, the dissertation presented a study about the effect of comments and function calls over feature location process. Two experiments for feature location concerning including or excluding the comments were conducted; the first one is with including the comment, where the second one is with ignoring the comments when indexing the source code. Additionally, another two experiments regarding including or excluding function calls are presented.

We feel that information that is orthogonal to the textual information will be the most relevant. Therefore, in our future work, we will experiment with such things as adding call graph data, frequent change pattern information, and program slice information, etc. Obtaining and identifying these types are particularly important to feature identification; their combination results in a very effective, accurate, and successful feature location technique [Liu et al. 2007].

The main directions for future work on this topic are to better answer the following questions:

- Why do stereotypes improve the accuracy of LSI results in feature location?
- What other types of information that when added to the source code corpus will more improve LSI results in feature location?
- What other types of information added to source code improves the results of IR methods?

- Does including multiple depths of function calls to the indexing process improve the process of feature and concept location or not, and to what level?

Moreover, we plan to develop a tool that can automatically rewrite any source code comments to be clearer and more understandable and helpful. In addition, we plan to make the proposed tool able to convert or translate any form of comments (e.g., UML) into natural language style.

The thesis addresses another important issue that relates to identifying the relevant methods from source code for a particular concept or feature (change request). More specifically, the dissertation presents a novel tool (QueBA) for generating based code queries. The QueBA expresses the names of documents, in most cases these are functions names, and synonyms, provided by WordNet. The tool employs the information extracted from both the problem domain and solution domain in order to provide better information about words/terms in the corpus in order to let the developers query the corpus more efficiently.

Regarding future work on this issue, the following two questions remain of interest in for future investigation:

1. What other types of information linked to QueBA helps in improving program comprehension?
2. Can adding visualization to QueBA help in better understanding source code concepts, features, relationships and dependency, and to what degree?

The dissertation also presents a discussion and demonstrates the usage of TraceLab in running experiments. More explicitly, it investigates the usage of LSI-

component in uncovering links between documents and source code. The discussions along with the presented experiments show how TraceLab can be used, and how the reusable component can be utilized to build experiments. In future work regarding this issue, we plan to employ TraceLab components to support other Software Engineering research such as predicting future maintenance activities.

Part of the plan, is to create a Singular Value Decomposition algorithm, which directly interacts with the matrix types inside of TraceLab. Also planned is the creation of a single component with the ability to compute the traceability links for further analysis within TraceLab, when given two corpora and their LSI Spaces.

Finally, the thesis presents an approach to categorize repository commits based on maintenance type; adaptive, corrective, perfective, and preventive. The approach is evaluated by identifying the adaptive commits changes over three open source systems, the next step to do in future, is to use the proposed approach in identifying all other types of maintenance. These experiments are to be repeated in future with the aim of locating other types of maintenance commits (i.e., corrective, perfective and inspection). With respect to the topic queries, the plan is to define several query templates through combining various methods to support selecting better terms from each topic to formulate enhanced queries.

This should improve the recall of the approach. Finally, efforts will also be directed at determining some good heuristics that the approach can use to determine the appropriate threshold value for investigating the retrieved ranked list to determine the criterion.

APPENDIX A

An Experiment Results of Qt System Commits Categorization with and without Stemming

Table 8-1. The resulted topics without stemming, number of topic chosen =5.

Topic	Topics Terms				
1	Svn_silent	Fix	Update	Compile	Warning
2	Fix	Compile	Svn_silent	Error	Warning
3	Compile	Fix	Warning	Crash	Bug
4	Warning	Fix	Compile	Remove	Update
5	Port	Add	Test	Api	Remove

Table 8-2. The resulted topics without stemming, number of topic chosen =10.

Topic	Topics Terms				
1	Compile	Fix	Error	Crash	Warnings
2	Fix	Compile	Add	Warnings	Crash
3	Update	Add	Fix	Compile	Warnings
4	Add	Update	Fix	Remove	Test
5	Warnings	Fix	Remove	Deprecated	Cleanup
6	Remove	Cleanup	Add	Upup	Code
7	Cleanup	Upup	Port	Support	Remove
8	Remove	Port	Support	Cleanup	Upup
9	Remove	Add	Port	Fix	Replace
10	Api	Replace	New	Adapt	Port

Table 8-3. The resulted topics with stemming, number of topic chosen =5.

Topic	Topics Terms				
1	Svn_silent	Updat	Fix	Warn	Compil
2	Compil	Fix	Error	Warn	Svn_silent
3	Warn	Compil	Fix	Remov	Add
4	Warn	Fix	Compil	Updat	Add
5	Fix	Updat	Replac	Remov	Port

Table 8-4. The resulted topics with stemming, number of topic chosen =10.

Topic	Topics Terms				
1	svn_silent	updat	Fix	Warn	compil
2	Compile	Fix	Error	Warn	Svn_silent
3	Warn	Compil	Fix	Remove	Add
4	Warn	Fix	Compile	Update	Add
5	Fix	Update	Add	Remove	Error
6	Update	Remove	Add	Cleanup	unus
7	Remove	Add	Cleanup	Unus	Test
8	Cleanup	Remove	Unus	Add	debug
9	Port	Fix	Adapt	Remove	crash
10	Replac	Remove	Api	Port	Add

Table 8-5. The resulted topics for the period 2005-2007 with stemming, number of topic chosen =5.

Topic	Topics Terms				
1	Svn_silent	Compil	Fix	Update	Warn
2	Compil	Fix	Svn_silent	Error	Crash
3	Fix	Compil	Error	Api	Crash
4	Warn	Remov	Deprec	Error	Fix
5	Port	Api	Remove	Replac	Updat

Table 8-6. The resulted topics for the period 2008-2010 with stemming, number of topic chosen =5.

Topic	Topics Terms				
1	Svn_silent	Fix	Update	Warn	Compil
2	Fix	Compil	Error	Svn_silent	Warn
3	Warn	Compil	Fix	Add	Api
4	Warn	Compil	Fix	Add	Updat
5	Remove	Add	Fix	Compil	Api

APPENDIX B

HippoDraw queried features (11 features) and the stereotypes for all relevant methods.

This appendix shows for the 11 features selected in the source code of the HippoDraw system, the relevant methods separately, and the stereotype type for each method.

Table 8-7. Stereotypes type of all relevant methods for the feature “change font size”.

#	Function name	Type
1	resetFontSize()	collaborational-command collaborator
2	setFontSize()	command collaborator
3	setZFontSize()	command collaborator
4	setXFontSize()	command collaborator
5	setYFontSize()	command collaborator
6	defaultFont()	command collaborator
7	initFont()	command collaborator
8	setDefaultFont()	command collaborator
9	Settingfonts()	command collaborator
10	setattributes()	collaborational-command collaborator

Table 8-8. Stereotypes type of all relevant methods for the feature “change font style italic”.

#	Function name	Type
1	setAttributes()	collaborational-command collaborator
2	setItalic()	command
3	setweight()	command
4	setFamily()	command
5	setDefaultFont()	command
6	setLabelFont()	command collaborator
7	createFontElements()	command collaborator
8	greateFontObject()	command collaborator
9	editLabelFontClicked()	command
10	editTitleFontClicked()	command
11	setXLabelFont()	command collaborator
12	setYLabelFont()	command collaborator
13	setZLabelFont()	command collaborator
14	setFonts()	command collaborator

Table 8-9. Stereotypes type of all relevant methods for the feature “update zoom mode”.

#	Function name	Type
1	getZoomMode()	command
2	isZoomPanChecked()	command
3	viewZoomReset()	command
4	viewZoomOut()	command
5	viewZoomIn()	command
6	enterEvent()	command
7	setZoomMode()	command
8	setZoomPan()	command
9	isZoomPan()	predicate

Table 8-10. Stereotypes type of all relevant methods for the feature” change printer settings”.

#	Function name	Type
1	initPrinter()	command collaborator
2	savePrinterSettings()	command
3	setPrinterSettings()	command
4	settingPrinter()	command
5	setPrinterBounds()	command
6	Print()	command
7	calcPrinterMetrics()	command collaborator
8	autosaveSettings()	command
9	saveSettings()	voidaccessor
10	initSettings()	command collaborator
11	editColorModel()	command collaborator
12	newColorModel()	command collaborator
13	setAppKey()	command

Table 8-11. Stereotypes type of all relevant methods for the feature” add item to canvas”.

#	Function name	Type
1	addFromPasteboard()	command collaborator
2	addSelectedItem()	command collaborator
3	placeGraphOnSelected()	command
4	addRecentFile()	command
5	addView()	command collaborator
6	Q3CanvasItem()	command
7	addPage()	command
8	initialize()	command collaborator
9	moduloAdd()	property collaborator
10	listSorted()	command collaborator

Table 8-12. Stereotypes type of all relevant methods for the feature” remove item from canvas”.

#	Function name	Type
1	Remove()	collaborator
2	removeDisplay()	command collaborator
3	removeSelected()	command collaborator
4	removeSelectedItem()	command collaborator
5	removeFromItemList()	command collaborator
6	clear()	command collaborator
7	deleteSelectedItem()	command collaborator
8	deleteSelected()	command collaborator
9	reTile()	command collaborator

Table 8-13. Stereotypes type of all relevant methods for the feature” change mouse property”.

#	Function name	Type
1	mouseEventData()	non-void-command collaborator
2	mouseSelectedDataRep()	property
3	contentsMouseEvent()	property
4	contentsMouseMoveEvent()	property
5	contentsMouseReleaseEvent()	property
6	mouseMoveMultiItem()	property
7	mouseData()	non-void-command collaborator
8	mousePressEvent()	command collaborator
9	controlMouseEvent()	command collaborator
10	fillMouseData()	command collaborator
11	leaveEvent()	command collaborator
12	enterEvent()	command collaborator

Table 8-14. Stereotypes type of all relevant methods for the feature” change cut color”.

#	Function name	Type
1	setCutColor()	command collaborator
2	setCutMode()	command collaborator
3	setCutEnabled()	command collaborator
4	setCuts()	command collaborator
5	updateTargets()	command collaborator
6	getCutColor()	property collaborator
7	setCut()	set
8	colorSelect_clicked()	command collaborator
9	colorSelect_2_clicked()	command collaborator

Table 8-15. Stereotypes type of all relevant methods for the feature” change representation color”.

#	Function name	Type
1	setRepColor()	command collaborator
2	setValueRep()	command collaborator
3	setRepresentation()	command collaborator
4	setRepStyle()	command
5	repColor()	property collaborator
6	getValueRep()	property collaborator
7	getRepColor()	property collaborator
8	representation()	property collaborator

Table 8-16. Stereotypes type of all relevant methods for the feature” make new display”.

#	Function name	Type
1	selectDisplay()	command collaborator
2	addDisplay()	command collaborator
3	addTextDisplay()	command collaborator
4	addFuncDisplay()	command collaborator
5	addPlotDisplay()	command collaborator
6	setX()	command collaborator
7	setY()	command collaborator
8	initialize()	command collaborator
9	createResidualsDisplay()	property collaborator factory
10	getDisplay()	non-void-command collaborator factory
11	addTextDisplayAt()	non-void-command collaborator factory
12	createDisplay()	non-void-command collaborator factory

Table 8-17. Stereotypes type of all relevant methods for the feature” update axis modeling”.

#	Function name	Type
1	fillAxisSizes()	command collaborator
2	setAutoRanging()	command collaborator
3	createAxisModels()	command collaborator
4	setAxisModel()	command collaborator
5	setAllAxisModels()	command collaborator
6	setAxisModelWithoutSetBin()	command collaborator
8	setAxisAttributes()	voidaccessor collaborator
9	checkAxisScaling()	command collaborator
10	getAxisModel()	get collaborator

APPENDIX C

Rules for Stereotype Identification

The authors in [Dragan et al. 2006], automatically recognized the main features to support reverse engineering method stereotypes from source code. Figure 8-1, shows the steps taken by the authors for identifying stereotypes and re-documenting the source code.

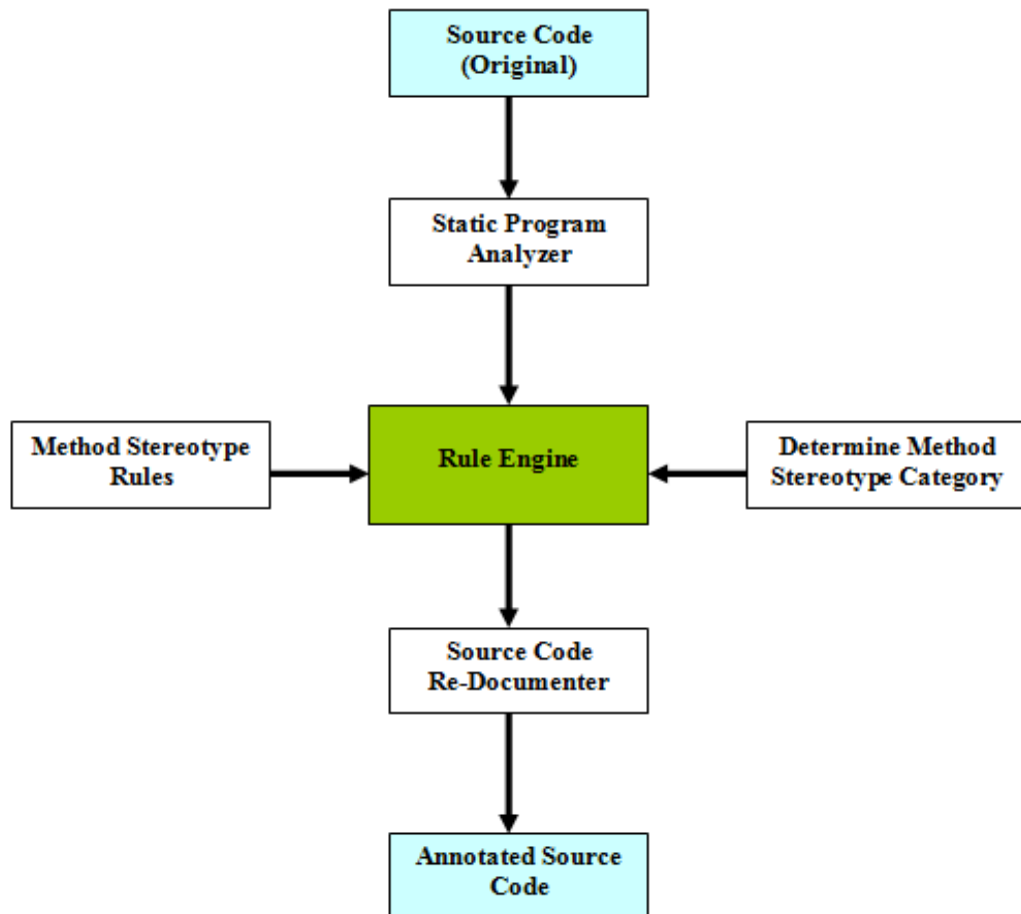


Figure 8-1. Steps for automatically identifying and re-documenting the source code with method stereotypes [Dragan et al. 2006].

Table 8-18. Stereotypes Identification Rules.

Stereotypes types	Conditions
Accessor::Get	<ul style="list-style-type: none"> • method is const • returns a data member • return type is primitive or container of a primitive
Accessor::Predicate	<ul style="list-style-type: none"> • method is const • returns a Boolean value that is not a data member
Accessor::Property	<ul style="list-style-type: none"> • method is const • does not return a data member • return type is primitive or container of primitives • return type is not Boolean
Mutator::Set	<ul style="list-style-type: none"> • method is not const • return type is void or Boolean • only one data member is changed
Mutator::Command	<ul style="list-style-type: none"> • method is not const • return type is void or Boolean • complex change to the object's state is perform
Collaborator	<ul style="list-style-type: none"> • returns void and at least one of the method's parameters or local variables is an object or • returns a parameter or local variable that is an Object
Creator::Factory	<ul style="list-style-type: none"> • returns an object created in the method's body
<ul style="list-style-type: none"> • accessors, mutators, and factory will result in a method only having a single stereotype 	
<ul style="list-style-type: none"> • A method may have a second stereotype of collaborator if it has a parameter or a local variable that is an object 	

REFERENCES

- [Alhindawi et al. 2013] Alhindawi, N., N. Dragan, et al. (2013). Improving Feature Location by Enhancing Source Code with Stereotypes. 29th IEEE International Conference on Software Maintenance (ICSM), Eindhoven, The Netherlands.
- [Alhindawi et al. 2013] Alhindawi, N., O. Meqdadi, et al. (2013). A TraceLab-Based Solution for Identifying Traceability Links using LSI. 7th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE). California, USA: 79-82.
- [Alhindawi et al. 2014] Alhindawi, N., O. Meqdadi, et al. (2014). Source Code Query Assistant Builder International Conference on Software Engineering (ICSE). To Be Submitted.
- [Alhindawi et al. 2013] Alhindawi, N., O. Meqdadi, et al. (2013). LSI-Based Solution for Categorizing Software Repository Commits for Maintenance Working Conference on Reverse Engineering (WCRE). To Be Submitted.
- [Alhindawi et al. 2013] Alhindawi, N., O. Meqdadi, et al. (2013). Source Code Indexing for Feature Location Working Conference on Reverse Engineering (WCRE). To Be Submitted.
- [Alonso et al. 2004] Alonso, O., P. T. Devanbu, et al. (2004). Database Techniques for the Analysis and Exploration of Software Repositories. 1st International Workshop on Mining Software Repositories (MSR), Edinburgh, Scotland, UK, IEE: Stevenage Herts, UK.

- [Anderson et al. 1999] Anderson, E., Z. Bai, et al. (1999). LAPACK Users' guide (third ed.), Society for Industrial and Applied Mathematics.
- [Anquetil and Lethbridge 1998] Anquetil, N. and T. Lethbridge (1998). Assessing the relevance of identifier names in a legacy software system. Centre for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, IBM Press.
- [Antoniol et al. 2002] Antoniol, G., G. Canfora, et al. (2002). "Recovering traceability links between code and documentation." IEEE Transactions on Software Engineering 28: 970-983.
- [Antoniol et al. 1999] Antoniol, G., G. Canfora, et al. (1999). Recovering code to documentation links in OO systems. 6th Working Conference on Reverse Engineering (WCRE), Georgia, USA.
- [Antoniol et al. 2000] Antoniol, G., G. Canfora, et al. (2000). Information Retrieval Models for Recovering Traceability Links between Code and Documentation. International Conference on Software Maintenance (ICSM), IEEE Computer Society.
- [Banker et al. 1991] Banker, R. D., S. M. Datar, et al. (1991). "A model to evaluate variables impacting the productivity of software maintenance projects." Manage. Sci. 37(1): 1-18.
- [Bellay and Gall 1998] Bellay, B. and H. Gall (1998). "An evaluation of reverse engineering tool capabilities." Journal of Software Maintenance: Research and Practice 10(5): 305-331.

- [Bennett and Rajlich 2000] Bennett, K. H. and V. T. Rajlich (2000). Software maintenance and evolution: a roadmap. The Future of Software Engineering, Limerick, Ireland, ACM.
- [Berg 1995] Berg, V. d. (1995). Software Measurement and Functional Programming. PhD, University of Twente, Enschede, the Netherlands.
- [Berry 1992] Berry, M. W. (1992). "Large Scale Sparse Singular Value Computations." International Journal of Supercomputer Applications 6: 13-49.
- [Biggerstaff and Richter 1987] Biggerstaff, T. and C. Richter (1987). "Reusability Framework, Assessment, and Directions." Software, IEEE 4(2): 41-49.
- [Biggerstaff et al. 1994] Biggerstaff, T. J., B. G. Mitbender, et al. (1994). "Program understanding and the concept assignment problem." Communications of the ACM 37(5): 72-82.
- [Binkley and Lawrie 2003] Binkley, D. and D. Lawrie (2003). "Information retrieval and the philosophy of language." Annual Review of Information Science and Technology 37(1): 3-50.
- [Binkley and Lawrie 2010] Binkley, D. and D. Lawrie (2010). Information Retrieval Applications in Software Development. Encyclopedia of Software Engineering. Taylor & Francis LLC.
- [Binkley and Lawrie 2010] Binkley, D. and D. Lawrie (2010). Information Retrieval Applications in Software Maintenance and Evolution. Encyclopedia of Software Engineering, P. Laplante, Ed. Taylor & Francis LLC.

- [Blei et al. 2003] Blei, D. M., A. Y. Ng, et al. (2003). "Latent dirichlet allocation." *J. Mach. Learn. Res.* 3: 993-1022.
- [Bohnet and Döllner 2006] Bohnet, J. and J. Döllner (2006). Visual exploration of function call graphs for feature location in complex software systems. *ACM symposium on Software visualization*, Brighton, United Kingdom, ACM.
- [Brooks 1983] Brooks, R. (1983). "Towards a theory of the comprehension of computer programs." *International Journal of Man-Machine Studies* 18(6): 543-554.
- [Canfora and Cerulo 2005] Canfora, G. and L. Cerulo (2005). Impact analysis by mining software and change request repositories. *11th IEEE International Symposium on Software Metrics*.
- [Canfora et al. 1993] Canfora, G., A. Cimitile, et al. (1993). A reverse engineering method for identifying reusable abstract data types. *Working Conference on Reverse Engineering*.
- [Carpineto and Romano 2012] Carpineto, C. and G. Romano (2012). "A Survey of Automatic Query Expansion in Information Retrieval." *ACM Comput. Surv.* 44(1): 1-50.
- [Castro-Herrera et al. 2009] Castro-Herrera, C., J. Cleland-Huang, et al. (2009). Enhancing Stakeholder Profiles to Improve Recommendations in Online Requirements Elicitation. *17th IEEE International Requirements Engineering Conference*.
- [Chatterjee et al. 2009] Chatterjee, S., S. Juvekar, et al. (2009). SNIFF: A Search Engine for Java Using Free-Form Queries. *12th International Conference on Fundamental*

Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS), York, UK, Springer-Verlag.

- [Chen et al. 2001] Chen, A., E. Chou, et al. (2001). CVSSearch: Searching through Source Code using CVS Comments. 17th IEEE International Conference on Software Maintenance (ICSM) Florence, Italy, IEEE Computer Society: Los Alamitos CA.
- [Chen and Rajlich 2000] Chen, K. and V. Rajlich (2000). Case Study of Feature Location Using Dependence Graph. Proceedings of the 8th International Workshop on Program Comprehension.
- [Choi and Scacchi 1990] Choi, S. C. and W. Scacchi (1990). "Extracting and Restructuring the Design of Large Systems." IEEE Softw. 7(1): 66-71.
- [Cleary et al. 2009] Cleary, B., C. Exton, et al. (2009). "An empirical analysis of information retrieval based concept location techniques in software comprehension." Empirical Software Engineering 14(1): 93-130.
- [Collard et al. 2011] Collard, M. L., M. J. Decker, et al. (2011). Lightweight Transformation and Fact Extraction with the srcML Toolkit. IEEE 11th International Working Conference on Source Code Analysis and Manipulation.
- [Collard et al. 2010] Collard, M. L., J. I. Maletic, et al. (2010). A lightweight transformational approach to support large scale adaptive changes. IEEE International Conference on Software Maintenance, IEEE Computer Society: 1-10.

- [Conklin et al. 2005] Conklin, M., J. Howison, et al. (2005). Collaboration using OSSmole: a repository of FLOSS data and analyses. 2nd International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri ACM Press: New York NY.
- [Cubranic and Murphy 2003] Cubranic, D. and G. C. Murphy (2003). Hipikat: recommending pertinent software development artifacts. 25th International Conference on Software Engineering.
- [Deerwester et al. 1990] Deerwester, S., S. T. Dumais, et al. (1990). "Indexing by Latent Semantic Analysis." *Journal of the American Society of Information Science* 41(6): 391-407.
- [DeLine et al. 2005] DeLine, R., A. Khella, et al. (2005). Towards understanding programs through wear-based filtering. ACM symposium on Software visualization. St. Louis, Missouri, ACM: 183-192.
- [Denys et al. 2005] Denys, P., M. Andrian, et al. (2005). IRiSS - A Source Code Exploration Tool. 21st IEEE International Conference on Software Maintenance (ICSM): 69-72.
- [Dit et al. 2012] Dit, B., E. Moritz, et al. (2012). A TraceLab-based solution for creating, conducting, and sharing feature location experiments. IEEE 20th International Conference on Program Comprehension (ICPC).
- [Dit et al. 2008] Dit, B., D. Poshyvanyk, et al. (2008). Measuring the Semantic Similarity of Comments in Bug Reports. 1st International Workshop on Semantic Technologies in System Maintenance (STSM).

- [Dit et al. 2011] Dit, B., M. Reville, et al. (2011). "Feature location in source code: a taxonomy and survey." *Journal of Software Maintenance and Evolution: Research and Practice* 25(1): 53 - 95.
- [Dragan et al. 2006] Dragan, N., M. L. Collard, et al. (2006). Reverse Engineering Method Stereotypes. 22nd IEEE International Conference on Software Maintenance.
- [Dragan et al. 2010] Dragan, N., M. L. Collard, et al. (2010). Automatic identification of class stereotypes. *IEEE International Conference on Software Maintenance (ICSM)*.
- [Eick et al. 1992] Eick, S. G., J. L. Steffen, et al. (1992). "Seesoft-A Tool for Visualizing Line Oriented Software Statistics." *IEEE Transaction on Software Engineering* 18(11): 957-968.
- [Eisenbarth et al. 2003] Eisenbarth, T., R. Koschke, et al. (2003). "Locating features in source code." *Transactions on Software Engineering* 29(3): 210-224.
- [Elliott Sim et al. 1999] Elliott Sim, S., C. L. A. Clarke, et al. (1999). Browsing and searching software architectures. *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*.
- [Elshoff and Marcotty 1982] Elshoff, J. L. and M. Marcotty (1982). "Improving computer program readability to aid modification." *Communication of the ACM* 25(8): 512-521.

- [Erdos and Sneed 1998] Erdos, K. and H. M. Sneed (1998). Partial comprehension of complex programs (enough to perform maintenance). 6th International Workshop on Program Comprehension (IWPC).
- [Etzkorn et al. 1999] Etzkorn, L. H., L. L. Bowen, et al. (1999). "An approach to program understanding by natural language understanding." *Nat. Lang. Eng.* 5(3): 219-236.
- [Faloutsos and Oard 1995] Faloutsos, C. and D. W. Oard (1995). A survey of information retrieval and filtering methods, University of Maryland at College Park: 23.
- [Fowler 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*, Addison-Wesley Longman Publishing Co., Inc.
- [Gasser et al. 2004] Gasser, L., G. Ripoché, et al. (2004). Research Infrastructure for Empirical Science of F/OSS. 1st International Workshop on Mining Software Repositories (MSR), Edinburgh, Scotland, UK, IEE: Stevenage Herts, UK.
- [German 2004] German, D. M. (2004). Mining CVS Repositories, the SoftChange Experience. 1st International Workshop on Mining Software Repositories (MSR), Edinburgh, Scotland, IEE: Stevenage Herts, UK.
- [Gleich et al. 2010] Gleich, D. F., P. G. Constantine, et al. (2010). Tracking the random surfer: empirically measured teleportation parameters in PageRank.
- [Grant et al. 2011] Grant, S., J. R. Cordy, et al. (2011). Reverse Engineering Co-maintenance Relationships Using Conceptual Analysis of Source Code. 18th Working Conference on Reverse Engineering (WCRE).

- [Gyongyi and Garcia-Molina 2005] Gyongyi, Z. and H. Garcia-Molina (2005). Link spam alliances. 31st international conference on Very large data bases, Trondheim, Norway, VLDB Endowment.
- [Haiduc 2011] Haiduc, S. (2011). Automatically detecting the quality of the query and its implications in IR-based concept location. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)
- [Haiduc et al. 2010] Haiduc, S., J. Aponte, et al. (2010). Supporting program comprehension with source code summarization. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, ACM.
- [Haiduc et al. 2010] Haiduc, S., J. Aponte, et al. (2010). On the Use of Automated Text Summarization Techniques for Summarizing Source Code. 17th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society.
- [Haiduc et al. 2013] Haiduc, S., G. Bavota, et al. (2013). Automatic Query Reformulations for Text Retrieval in Software Engineering. 35th IEEE/ACM International Conference on Software Engineering (ICSE). San Francisco, USA.
- [Haiduc et al. 2012] Haiduc, S., G. Bavota, et al. (2012). Evaluating the specificity of text retrieval queries to support software engineering tasks. 34th International Conference on Software Engineering (ICSE).
- [Hammad et al. 2011] Hammad, M., M. L. Collard, et al. (2011). "Automatically identifying changes that impact code-to-design traceability during evolution." Software Quality Control 19(1): 35-64.

- [Hartman 1991] Hartman, J. E. (1991). Automatic control understanding for natural programs, University of Texas at Austin.
- [Hattori and Lanza 2008] Hattori, L. P. and M. Lanza (2008). On the nature of commits. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)
- [Hill et al. 2011] Hill, E., L. Pollock, et al. (2011). Improving source code search with natural language phrasal representations of method signatures. 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society.
- [Hindle and German 2005] Hindle, A. and D. M. German (2005). SCQL: A Formal Model and a Query Language for Source Control Repositories. 2nd International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri ACM Press: New York NY.
- [Hindle et al. 2009] Hindle, A., D. M. German, et al. (2009). Automatic classification of large changes into maintenance categories. 17th IEEE International Conference on Program Comprehension (ICPC).
- [Hindle et al. 2009] Hindle, A., M. W. Godfrey, et al. (2009). What's hot and what's not: Windowed developer topic analysis. 25th International Conference on Software Maintenance (ICSM).
- [Holmes and Murphy 2005] Holmes, R. and G. C. Murphy (2005). Using structural context to recommend source code examples. 27th international conference on Software engineering (ICSE). St. Louis, MO, USA, ACM: 117-125.

- [Holzmann 2002] Holzmann, G. J. (2002). Static source code checking for user-defined properties. Integrated Design and Process Technology (IDPT), CA, USA.
- [Howden 1990] Howden, W. E. (1990). "Comments Analysis and Programming Errors." IEEE Transition on Software Engineering 16(1): 72-81.
- [Huibers et al. 1996] Huibers, T. W. C., M. Lalmas, et al. (1996). "Information retrieval and situation theory." SIGIR Forum 30(1): 11-25.
- [Hussein et al. 2009] Hussein, K., E. Tilevich, et al. (2009). Sonification design guidelines to enhance program comprehension. 17th IEEE International Conference on Program Comprehension (ICPC)
- [Jolliffe 1986] Jolliffe, I. T. (1986). Principal Component Analysis, Springer Verlag.
- [Kagdi et al. 2007] Kagdi, H., M. L. Collard, et al. (2007). An approach to mining call-usage patterns with syntactic context. twenty-second IEEE/ACM international conference on Automated software engineering (ASE). Atlanta, Georgia, USA, ACM: 457-460.
- [Kagdi et al. 2007] Kagdi, H., M. L. Collard, et al. (2007). "A survey and taxonomy of approaches for mining software repositories in the context of software evolution." Software Maintenance and Evolution 19(2): 77-131.
- [Kawaguchi et al. 2003] Kawaguchi, S., P. K. Garg, et al. (2003). Automatic Categorization Algorithm for Evolvable Software Archive. 6th International Workshop on Principles of Software Evolution, IEEE Computer Society: 195.

- [Kawaguchi et al. 2006] Kawaguchi, S., P. K. Garg, et al. (2006). "MUDABlue: an automatic categorization system for open source repositories." *Systems and Software* 79(7): 939-953.
- [Keenan et al. 2012] Keenan, E., A. Czauderna, et al. (2012). TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. 34th International Conference on Software Engineering (ICSE)
- [Kersten and Murphy 2005] Kersten, M. and G. C. Murphy (2005). Mylar: a degree-of-interest model for IDEs. 4th international conference on Aspect-oriented software development, Chicago, Illinois, ACM.
- [Khamis et al. 2010] Khamis, N., R. Witte, et al. (2010). Automatic quality assessment of source code comments: the JavadocMiner. 15th international conference on Applications of natural language to information systems, Cardiff, UK, Springer-Verlag.
- [Kim and Stohr 1998] Kim, Y. and E. A. Stohr (1998). "Software reuse: survey and research directions." *Management Information Systems* 14(4): 113-147.
- [Kuhn et al. 2007] Kuhn, A., S. Ducasse, et al. (2007). "Semantic clustering: Identifying topics in source code." *Information and Software Technology* 49(3): 230-243.
- [Landauer and Dumais 1997] Landauer, T. K. and S. T. Dumais (1997). "A solution to Plato's problem: The Latent Semantic Analysis theory of the acquisition, induction, and representation of knowledge." *Psychological Review* 104.

- [Laski and Korel 1983] Laski, J. W. and B. Korel (1983). "A Data Flow Oriented Program Testing Strategy." IEEE Transactions on Software Engineering SE-9(3): 347-354.
- [Lehman 1980] Lehman, M. M. (1980). "Programs, life cycles, and laws of software evolution." Proceedings of the IEEE 68(9): 1060-1076.
- [Lehman et al. 1997] Lehman, M. M., J. F. Ramil, et al. (1997). Metrics and laws of software evolution-the nineties view. 4th International Software Metrics Symposium
- [Lientz and Swanson 1980] Lientz, B. P. and E. B. Swanson (1980). Software Maintenance Management, Addison-Wesley Longman Publishing Co., Inc.
- [Lientz et al. 1978] Lientz, B. P., E. B. Swanson, et al. (1978). "Characteristics of application software maintenance." Communications of the ACM 21(6): 466-471.
- [Linstead et al. 2008] Linstead, E., C. Lopes, et al. (2008). An Application of Latent Dirichlet Allocation to Analyzing Software Evolution. Seventh International Conference on Machine Learning and Applications, IEEE Computer Society.
- [Littman et al. 1986] Littman, D. C., J. Pinto, et al. (1986). Mental models and software maintenance. first workshop on empirical studies of programmers on Empirical studies of programmers. Washington, D.C., USA, Ablex Publishing Corp.: 80-98.
- [Liu et al. 2007] Liu, D., A. Marcus, et al. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. twenty-second IEEE/ACM international conference on Automated software engineering (ASE), Atlanta, Georgia, USA, ACM.

- [Livadas and Alden 1993] Livadas, P. E. and S. D. Alden (1993). A toolset for program understanding. 2nd IEEE Workshop on Program Comprehension.
- [Livshits and Zimmermann 2005] Livshits, B. and T. Zimmermann (2005). "DynaMine: finding common error patterns by mining software revision histories." SIGSOFT Software Engineering Notes 30(5): 296-305.
- [Maarek et al. 1991] Maarek, Y. S., D. M. Berry, et al. (1991). "An Information Retrieval Approach for Automatically Constructing Software Libraries." IEEE Transactions on Software Engineering 17(8): 800-813.
- [Mahmoud and Niu 2011] Mahmoud, A. and N. Niu (2011). Source code indexing for automated tracing. 6th International Workshop on Traceability in Emerging Forms of Software Engineering, Waikiki, Honolulu, HI, USA, ACM.
- [Maletic and Kagdi 2008] Maletic, J. I. and H. Kagdi (2008). Expressiveness and effectiveness of program comprehension: Thoughts on future research directions. Frontiers of Software Maintenance (FoSM).
- [Maletic and Marcus 2000] Maletic, J. I. and A. Marcus (2000). Support for Software Maintenance Using Latent Semantic Analysis. 4th Annual IASTED International Conference on Software Engineering and Applications (SEA).
- [Maletic and Marcus 2000] Maletic, J. I. and A. Marcus (2000). Using latent semantic analysis to identify similarities in source code to support program understanding. 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)
- [Maletic and Marcus 2001] Maletic, J. I. and A. Marcus (2001). Supporting program comprehension using semantic and structural information. 23rd International

Conference on Software Engineering (ICSE). Toronto, Ontario, Canada, IEEE Computer Society: 103-112.

[Maletic and Reynolds 1994] Maletic, J. I. and R. G. Reynolds (1994). A tool to support knowledge based software maintenance: the Software Service Bay. 6th International Conference on Tools with Artificial Intelligence

[Maletic and Valluri 1999] Maletic, J. I. and N. Valluri (1999). Automatic Software Clustering via Latent Semantic Analysis. 14th IEEE international conference on Automated software engineering (ASE), IEEE Computer Society: 251.

[Mandelin et al. 2005] Mandelin, D., L. Xu, et al. (2005). "Jungloid mining: helping to navigate the API jungle." SIGPLAN 40(6): 48-61.

[Marcus and Maletic 2001] Marcus, A. and J. I. Maletic (2001). Identification of High-Level Concept Clones in Source Code. 16th IEEE international conference on Automated software engineering (ASE), IEEE Computer Society: 107.

[Marcus and Maletic 2003] Marcus, A. and J. I. Maletic (2003). Recovering documentation-to-source-code traceability links using latent semantic indexing. 25th International Conference on Software Engineering. Portland, Oregon, IEEE Computer Society: 125-135.

[Marcus and Poshyvanyk 2005] Marcus, A. and D. Poshyvanyk (2005). The Conceptual Cohesion of Classes. 21st IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society.

- [Marcus et al. 2008] Marcus, A., D. Poshyvanyk, et al. (2008). "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems." IEEE Transactions on Software Engineering 34(2): 287-300.
- [Marcus et al. 2004] Marcus, A., A. Sergeyev, et al. (2004). An Information Retrieval Approach to Concept Location in Source Code. 11th Working Conference on Reverse Engineering, IEEE Computer Society.
- [MARTIN and MCCLURE. 1983] MARTIN, J. and C. MCCLURE. (1983). Software Maintenance: The Problem and Its Solutions. Englewood Cliffs, NJ, Prentice Hall.
- [Mayrhauser and Vans 1997] Mayrhauser, A. v. and A. M. Vans (1997). Program understanding behavior during debugging of large scale software. seventh workshop on Empirical studies of programmers. Alexandria, Virginia, USA, ACM: 157-179.
- [Mayrhauser and A 1994] Mayrhauser, V. and V. A (1994). Program Understanding - A Survey. Department of Computer Science, Colorado State University: 32.
- [McMillan et al. 2011] McMillan, C., M. Grechanik, et al. (2011). Portfolio: finding relevant functions and their usage. 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, ACM.
- [McMillan et al. 2011] McMillan, C., M. Linares-Vasquez, et al. (2011). Categorizing software applications for maintenance. 27th IEEE International Conference on Software Maintenance (ICSM).

- [Mockus and Votta 2000] Mockus, A. and L. G. Votta (2000). Identifying reasons for software changes using historic databases. International Conference on Software Maintenance (ICSM)
- [Müller et al. 2004] Müller, H.-M., E. E. Kenny, et al. (2004). Textpresso: An Ontology-Based Information Retrieval and Extraction System for Biological Literature. PLoS Biol., 2, e309.
- [Niessink and Vliet 2000] Niessink, F. and H. v. Vliet (2000). "Software maintenance from a service perspective." Journal of Software Maintenance 12(2): 103-120.
- [Ohba and Gondow 2005] Ohba, M. and K. Gondow (2005). Toward Mining Concept Keywords from Identifiers in Large Software Projects. 2nd International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri ACM Press: New York NY.
- [Oram 2001] Oram, P. (2001). "WordNet: An electronic lexical database. Christiane Fellbaum (Ed.). Cambridge, MA: MIT Press, 1998. Pp. 423." Applied Psycholinguistics 22(01): 131-134.
- [Ossher et al. 2009] Ossher, J., S. Bajracharya, et al. (2009). SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. 6th IEEE International Working Conference on Mining Software Repositories (MSR)
- [Padioleau et al. 2009] Padioleau, Y., T. Lin, et al. (2009). Listening to programmers- Taxonomies and characteristics of comments in operating system code. 31st IEEE International Conference on Software Engineering (ICSE)

- [Pennington 1987] Pennington, N. (1987). "Stimulus structures and mental representations in expert comprehension of computer programs." *Cognitive Psychology* 19: 295-341.
- [Penta et al. 2007] Penta, M. D., R. E. K. Stirewalt, et al. (2007). Designing your Next Empirical Study on Program Comprehension. 15th IEEE International Conference on Program Comprehension (ICPC), IEEE Computer Society.
- [Perotte et al. 2011] Perotte, A., N. Bartlett, et al. (2011). Hierarchically Supervised Latent Dirichlet Allocation. *Neural Information Processing Systems*.
- [Pollock et al. 2007] Pollock, L., K. Vijay-Shanker, et al. (2007). Introducing natural language program analysis. 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, San Diego, California, USA, ACM.
- [Poshyvanyk 2009] Poshyvanyk, D. (2009). Using information retrieval to support software maintenance tasks. *IEEE International Conference on Software Maintenance (ICSM)*.
- [Poshyvanyk et al. 2013] Poshyvanyk, D., M. Gethers, et al. (2013). "Concept location using formal concept analysis and information retrieval." *TOSEM* 21(4): 1-34.
- [Poshyvanyk et al. 2007] Poshyvanyk, D., Y. G. Gueheneuc, et al. (2007). "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval." *Software Engineering, IEEE Transactions on* 33(6): 420-432.

- [Poshyvanyk and Marcus 2007] Poshyvanyk, D. and A. Marcus (2007). Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. 15th IEEE International Conference on Program Comprehension (ICPC).
- [Poshyvanyk et al. 2005] Poshyvanyk, D., A. Marcus, et al. (2005). IRiSS - A Source Code Exploration Tool. 21st IEEE International Conference on Software Maintenance (ICSM).
- [Poshyvanyk et al. 2009] Poshyvanyk, D., A. Marcus, et al. (2009). "Using information retrieval based coupling measures for impact analysis." *Empirical Software Engineering* 14(1): 5-32.
- [Poshyvanyk et al. 2006] Poshyvanyk, D., A. Marcus, et al. (2006). Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. 14th IEEE International Conference on Program Comprehension (ICPC), IEEE Computer Society.
- [Reiss 2009] Reiss, S. (2009). Semantics-based code search. 31 st IEEE International Conference on Software Engineering. Canada: 243–253.
- [Revelle et al. 2010] Revelle, M., B. Dit, et al. (2010). Using Data Fusion and Web Mining to Support Feature Location in Software. 18th IEEE International Conference on Program Comprehension (ICPC).
- [Revelle and Poshyvanyk 2009] Revelle, M. and D. Poshyvanyk (2009). An exploratory study on assessing feature location techniques. 17th IEEE International Conference on Program Comprehension (ICPC).

- [Rist 1986] Rist, R. (1986). Plans in programming: definition, demonstration, and development. first workshop on empirical studies of programmers on Empirical studies of programmers. Washington, D.C., USA, Ablex Publishing Corp.: 28-47.
- [Robillard and Murphy 2003] Robillard, M. P. and G. C. Murphy (2003). FEAT a tool for locating, describing, and analyzing concerns in source code. 25th IEEE International Conference on Software Engineering (ICSE)
- [Robles et al. 2004] Robles, G., J. M. González-Barahona, et al. (2004). GlueTheos: Automating the Retrieval and Analysis of Data from Publicly Available Software Repositories. 1st International Workshop on Mining Software Repositories (MSR), Edinburgh, Scotland, UK, IEE: Stevenage Herts, UK.
- [Ryder 1979] Ryder, B. G. (1979). "Constructing the Call Graph of a Program." IEEE Transactions on Software Engineering SE-5(3): 216-226.
- [Salton and McGill 1983] Salton, G. and M. J. McGill (1983). Introduction to Modern Information Retrieval. , McGraw-Hill.
- [Salton et al. 1975] Salton, G., A. Wong, et al. (1975). "A vector space model for automatic indexing." Communications of the ACM 18(11): 613-620.
- [Savage et al. 2010] Savage, T., B. Dit, et al. (2010). TopicXP: Exploring topics in source code using Latent Dirichlet Allocation. IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society.
- [Schach et al. 2003] Schach, S. R., B. Jin, et al. (2003). "Determining the Distribution of Maintenance Categories: Survey versus Measurement." empirical software engineering 8(4): 351-365.

- [Schreck et al. 2007] Schreck, D., V. Dallmeier, et al. (2007). How documentation evolves over time. Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting. Dubrovnik, Croatia, ACM: 4-10.
- [Shepherd et al. 2007] Shepherd, D., Z. P. Fry, et al. (2007). Using natural language program analysis to locate and understand action-oriented concerns. 6th IEEE international conference on Aspect-oriented software development, Vancouver, British Columbia, Canada, ACM.
- [Shneiderman and Mayer 1979] Shneiderman, B. and R. Mayer (1979). "Syntactic/semantic interactions in programmer behavior: A model and experimental results." International Journal of Computer & Information Sciences 8(3): 219-238.
- [Sillito et al. 2008] Sillito, J., G. C. Murphy., et al. (2008). "Asking and Answering Questions during a Programming Change Task." IEEE Transactions on Software Engineering 34(4): 434-451.
- [Sim et al. 1998] Sim, S. E., C. L. Clarke, et al. (1998). Archetypal source code searches: a survey of software developers and maintainers. 6th IEEE International Workshop on Program Comprehension (IWPC).
- [Singer et al. 2005] Singer, J., R. Elves, et al. (2005). NavTracks: supporting navigation in software maintenance. 21st IEEE International Conference on Software Maintenance (ICSM)

- [Sliwerski et al. 2005] Sliwerski, J., T. Zimmermann, et al. (2005). When do changes induce fixes? 2nd International Workshop on Mining Software Repositories (MSR), St. Louis, Missouri ACM Press: New York NY.
- [Smart et al. 2008] Smart, P. R., A. Russell, et al. (2008). A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. 16th international conference on Knowledge Engineering: Practice and Patterns, Acitrezza, Italy, Springer-Verlag.
- [Spuida 2002] Spuida, B. (2002). The fine Art of Commenting. Tech Notes, general Series. S. W. Wrangler.
- [Sridhara et al. 2008] Sridhara, G., E. Hill, et al. (2008). Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. 16th IEEE International Conference on Program Comprehension (ICPC)
- [Stanchev 2012] Stanchev, L. (2012). Building semantic corpus from wordNet. IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW).
- [Starke et al. 2009] Starke, J., C. Luce, et al. (2009). Searching and skimming: An exploratory study. IEEE International Conference on Software Maintenance (ICSM)
- [Storey et al. 1997] Storey, M.-A. D., K. Wong, et al. (1997). How Do Program Understanding Tools Affect How Programmers Understand Programs. 4th IEEE Working Conference on Reverse Engineering (WCRE), IEEE Computer Society: 12.

- [Storey 2005] Storey, M. (2005). Theories, methods and tools in program comprehension: past, present and future. 13th International Workshop on Program Comprehension (IWPC)
- [Storey and Muller 1995] Storey, M. A. D. and H. A. Muller (1995). Manipulating and documenting software structures using SHriMP views. International Conference on Software Maintenance (ICSM)
- [Stylos and Myers 2006] Stylos, J. and B. A. Myers (2006). Mica: A Web-Search Tool for Finding API Components and Examples. IEEE Symposium on Visual Languages and Human-Centric Computing
- [Tan et al. 2007] Tan, L., D. Yuan, et al. (2007). Hotcomments: how to make program comments more useful? 11th USENIX workshop on Hot topics in operating systems, San Diego, CA, USENIX Association.
- [Teevan 2001] Teevan, J. (2001). Improving Information Retrieval with Textual Analysis: Bayesian Models and Beyond.
- [Tian et al. 2009] Tian, K., M. Reville, et al. (2009). Using Latent Dirichlet Allocation for automatic categorization of software. 6th IEEE International Working Conference on Mining Software Repositories (MSR), IEEE Computer Society.
- [Toffolon and Dakhli 2008] Toffolon, C. and S. Dakhli (2008). An Iterative Meta-Lifecycle for Software Development, Evolution and Maintenance. 3rd International Conference on Software Engineering Advances (ICSEA)

- [Turver and Munro 1994] Turver, R. J. and M. Munro (1994). "An early impact analysis technique for software maintenance." *Journal of Software Maintenance: Research and Practice* 6(1): 35-52.
- [Vliet 2000] Vliet, V. (2000). *Software Engineering: Principles and Practices*. John Wiley & Sons, West Sussex, England.
- [von Mayrhauser and Vans 1993] von Mayrhauser, A. and A. M. Vans (1993). From code understanding needs to reverse engineering tool capabilities. 6th International Workshop on Computer-Aided Software Engineering (CASE).
- [Wilde et al. 1992] Wilde, N., J. A. Gomez, et al. (1992). Locating user functionality in old code. *Software Maintenance*, .
- [Wilde and Scully 1995] Wilde, N. and M. C. Scully (1995). "Software reconnaissance: mapping program features to code." *Journal of Software Maintenance* 7(1): 49-62.
- [Wong et al. 2000] Wong, E., S. Gokhale, et al. (2000). "Quantifying the closeness between program components and features." *Journal of Systems and Software - Special issue on software maintenance* 54(2): 87-98.
- [Woodfield et al. 1981] Woodfield, S. N., H. E. Dunsmore, et al. (1981). The effect of modularization and comments on program comprehension. 5th IEEE International Conference on Software Engineering (ICSE), San Diego, California, USA, IEEE Press.
- [Yin 2009] Yin, R. K. (2009). *Case Study Research: Design and Methods* (4th Edition). Thousand Oaks, CA, Sage.

[Zhao et al. 2004] Zhao, W., L. Zhang, et al. (2004). SNIAFL: Towards a Static Non-Interactive Approach to Feature Location. 26th IEEE International Conference on Software Engineering (ICSE).

[Zimmermann et al. 2004] Zimmermann, T., P. Weißgerber, et al. (2004). Mining Version Histories to Guide Software Changes. 26th IEEE International Conference on Software Engineering (ICSE), Edinburgh, Scotland, United Kingdom.