UNDERSTANDING AND IDENTIFYING LARGE-SCALE ADAPTIVE CHANGES FROM VERSION HISTORIES

A dissertation submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Doctor of Philosophy

by

Omar M. Meqdadi

August, 2013

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

Dissertation written by

Omar M. Meqdadi

B.S., Jordan University of Science and Technology, Jordan, 2002

M.S., Jordan University of Science and Technology, Jordan, 2007

Ph.D. Kent Sate University, USA, 2013

Approved by

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

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

Dr. L.Gwenn Volkert

Dr. Michael L. Collard

Dr. Joseph Ortiz

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 ..... **XI**

ACKNOWLEDGMENTS ..... **XV**

CHAPTER 1 INTRODUCTION **1**

1.1 Recognizing Adaptive Changes 3

1.2 Research Focus 4

1.3 Conducted Research 5

1.4 Contributions 6

1.5 Organization 7

CHAPTER 2 A BACKGROUND AND RELATED WORK ON ADAPTIVE MAINTENANCE **9**

2.1 Software Maintenance Categorization 9

2.2 Adaptive Maintenance Overview 11

2.3 Adaptive Maintenance Process 14

2.4 Adaptive Maintenance Challenges 15

2.5 Automated Adaptive Process Background 19

2.6 Summary 22

CHAPTER 3 A MANUAL IDENTIFICATION OF ADAPTIVE COMMITS **23**

3.1 Software Repository commits 24

3.2 Subject Systems 26

3.2.1 KOffice System 27

3.2.2 Extragear/graphics System 28
3.2.3 OSG System 28
3.3 Manual Investigation of Adaptive Commits 29
3.3.1 Investigation Approach 29
3.3.2 Summary of Findings 31
3.4 Manual Investigation Challenges 34
3.5 Summary 35
CHAPTER 4 CHARACTERIZING THE ADAPTIVE COMMITS 36
4.1 Adaptive Commit Size Categorization 37
4.2 Adaptive Commit Vocabulary 47
4.3 Adaptive Commit Developers 49
4.4 Developer and Vocabulary Correlation 54
4.5 Threats to Validity 59
4.6 Related Work 63
4.7 Summary 66
CHAPTER 5 EXAMINING ADAPTIVE CHANGES ON SOFTWARE ARTIFACTS 68
5.1 Software Artifact Types 68
5.2 Features Extraction from Changed Artifacts 70
5.3 Commit Categorization Using Changed Artifacts 72
5.4 Changed Files Categorization 76
5.5 Threats to Validity 85

5.6 Related Work 87

5.7 Summary 89

CHAPTER 6 STEREOTYPE OF ADAPTIVE MODIFIED METHODS: A CASE STUDY 91

6.1 Method Stereotypes 91

6.2 Method Stereotype Distributions 92

6.3 Stereotype-Based Commit Categorization 98

6.4 Threats to Validity 102

6.5 Related Work 103

6.6 Summary 105

CHAPTER 7 AUTOMATIC IDENTIFICATION OF ADAPTIVE COMMITS USING LSI 107

7.1 LSI -Based Approach 108

7.1.1 Overview of Latent Semantic Indexing 109

7.1.2 The Approach 109

7.2 Evaluation 114

7.2.1 LSI Topic Extraction Results 115

7.2.2 Adaptive commits Allocating Results 117

7.3 Threats to Validity 125

7.4 Related Work 126

7.5 Summary 128

CHAPTER 8 AUTOMATICALLY UNCOVERING FOR TRACEABILITY LINKS FROM ADAPTIVE COMMITS 130

8.1 Uncovering Traceability Link 131

8.1.1 Commit Grouping Heuristics 132

8.1.2 Frequent Pattern-Mining Approach 134

8.1.3 Uncovered Traceability Link Data Set 136

8.2 Validation 141

8.2.1 TraceLab - Based Validation Experiment 142

8.2.2 Validation Metrics 143

8.2.3 Validation Results 144

8.3 Threats to Validity 145

8.4 Related Work 147

8.5 Summary 150

CHAPTER 9 CONCLUSIONS AND FUTURE WORK 152

9.1 Conclusion 152

9.2 Future Work 155

APPENDIX A ADAPTIVE COMMITS DISTRIBUTIONS PER YEAR 157

A.1 Distribution for KOffice 157

A.2 Distributions for Extragear/graphics 160

A.3 KOffice Subsystems Changing Span 162

A.3.1 Adaptive Changing Span 162

A.3.2 Non-Adaptive Changing Span 163

APPENDIX B ADAPTIVE CHANGES AT SOURCE CODE LEVEL 164

B.1 Approach 164

B.2 SVN Diff Command 164

B.3 Examples of Adaptive Commits for KOffice 165

B.4 Classification of Adaptive Changes 169

B.4.1 Method Signature Change 169

B.4.2 Method Replacement 170

B.4.3 Variable Type Change 171

B.4.4 Introducing New Class 172

B.4.5 Introducing New Global Methods 172

B.4.6 Enum Value Renaming 173

B.4.7 Header File Renaming 173

B.4.8 Classification Case Study: KOffice 174

REFERENCES 176

LIST OF FIGURES

Figure 2-1. Adaptive maintenance process. 16

Figure 2-2. An example of a migration from Qt3 to Qt4, which was applied to the KDE project. 16

Figure 3-1. A subversion log entry from KOffice system. The entry includes the revision, the files in the change set, and a developer description of the change. 25

Figure 4-1. A box plot showing the five segments, which are created by the five-point summarization commits into five categories. 39

Figure 4-2. Histogram for the number of commits distributed over number of modules for Extragear/graphics system. 46

Figure 4-3. Term distribution for each active developer of KOffice system. 57

Figure 4-4. Term usage distribution of the top most active adaptive maintenance committers for KOffice. 58

Figure 5-1. Histogram of the number of different non-source code file extensions with regards to the number of commits in the KOffice system 76

Figure 5-2. Histograms of the number of different non-source code file extensions with regards to the number of commits in the Extragear/graphics system 77

Figure 5-3. Histogram of the number of different non-source code file extensions with regards to the number of commits in the OSG system. 77

Figure 5-4. Histogram of the number of *source code* files with regards to the type of affected commit in the three examined systems 80

Figure 5-5. Histogram of the number of *non-source* code files with regards to the type of affected commit in the three examined systems 81
Figure 5-6. Histogram of the non-source code artifacts with regards to the number of *adaptive* commits in the three examined system 83
Figure 5-7. Histogram of the non-source code artifacts with regards to the number of *non-adaptive* commits in the three examined systems 83
Figure 5-8. Frequencies of the two build management documents. Percentages are compared to total modified build management files for the *adaptive* commits in the three examined system..... 85
Figure 5-9. Frequencies of the two build management documents. Percentages are compared to total modified build management files for the *non-adaptive* commits in the three examined system..... 86
Figure 6-1. Approach to identify stereotype of modified methods for adaptive commits.94
Figure 6-2. Frequency distribution of stereotype categories for commits of KOffice system. 97
Figure 6-3. Histogram for the number of commits distributed over stereotype-based size for the KOffice system. 99
Figure 6-4. Histogram for the number of commits distributed over number of stereotype categories for the KOffice system. 101
Figure 7-1. Using LSI to identify the adaptive commit 110
Figure 7-2. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for KOffice system..... 122

Figure 7-3. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for Exrtragear/graphic system. 122

Figure 7-4. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for OSG system. 123

Figure 8-1. A snippet of input transactions from KOffice adaptive commits grouped by heuristic CommitDay. 135

Figure 8-2. A snapshot for the results of running one query sample (Results with first 2000 documents & LSI Dimensionality = 300. 146

LIST OF TABLES

Table 3-1. Adaptive and none adaptive commits for the three examined systems over the given time period. 32

Table 3-2. Distribution of adaptive commits per year. Percentages are compared to total adaptive commits. 33

Table 3-3. Distribution of adaptive commits per year. Percentages are compared to total commits (adaptive and non-adaptive). 33

Table 4-1. Commits categorized by file-size using five-point summarize approach for KOffice system. 40

Table 4-2. Commits categorized by file-size using five-point summarize approach for Extragear/graphics system. 41

Table 4-3. Commits categorized by file-size using five-point summarize approach for OSG system. 42

Table 4-4. Commits categorized by method-size using five-point summarize approach for KOffice system. 44

Table 4-5. Commits categorized by method-size using five-point summarize approach for Extragear/graphics system. 44

Table 4-6 . Commits categorized by method-size using five-point summarize approach for OSG system. 45

Table 4-7. Defined categories by module-size using five-point summarization. 45

Table 4-8. P-values for the three size measures using Mann-Whitney U tests for the three examined systems 47
Table 4-9. Top 12 average single term frequency in the adaptive commits and their frequency in non-adaptive commits. 50
Table 4-10. Top 12 average single term frequency in the non-adaptive commits and their frequency in adaptive commits. 51
Table 4-11. Top 12 average term set frequency in the adaptive commits and their frequency in non-adaptive commit s. 52
Table 4-12. Frequency distribution of KOffice developers contribution who made at least two adaptive commits. 55
Table 4-13. Frequency distribution of Extragear/graphics developers contribution who made at least two adaptive commits. 56
Table 4-14. Frequency distribution of OSG developers contribution who made at least two adaptive commits. 56
Table 4-15. Frequency distribution of developer-term sets over KOffice system commits. 60
Table 4-16. Frequency distribution of developer-term sets over Extragear/graphics system commits. 61
Table 4-17. Frequency distribution of developer-term sets over OSG system commits. . 62
Table 5-1 . Commit categorization based on type of modified files for KOffice system. 74
Table 5-2. Commit categorization based on type of modified files for Extargear/graphics system. 74

Table 5-3. Commit categorization based on type of modified files for OSG system. 75

Table 5-4. File extensions that were modified by adaptive commits for the three studied systems, (Y) means modified otherwise no modification were occurred. 78

Table 6-1. A taxonomy of method stereotypes. 93

Table 6-2. Frequency distribution of stereotypes of modified methods occurring in undertaken commits of KOffice system. 95

Table 6-3. Stereotype category frequency in the adaptive commits and their frequency in none adaptive commit for KOffice system. 96

Table 6-4. Defined categories by stereotype-based size using five-point summarization. 99

Table 7-1. Details of the LSI generated Corpora for the three studied systems. 115

Table 7-2. Topic extraction results for KOffice system. Each topic has 5 terms, where each term has a corresponding relevancy rank. 118

Table 7-3. Topic extraction results for Extragear/graphics system. Each topic has 5 terms, where each term has a corresponding relevancy rank. 119

Table 7-4. Topic extraction results for OSG system. Each topic has 5 terms, where each term has a corresponding relevancy rank. 120

Table 7-5. The Recall values of the adaptive commits using the TAM union set for each studied system. 124

Table 8-1. Groups formed from the adaptive commits by the different heuristics. 134

Table 8-2. Traceability patterns uncovered from the commits of the KOffice system... 136

Table 8-3. Traceability patterns uncovered from the commits of the Extragear/graphics system. 137

Table 8-4. Traceability patterns uncovered from the commits of the OSG system. 137

Table 8-5. Examples of traceability patterns uncovered from the *adaptive* commits of the examined systems. 138

Table 8-6. Distribution of patterns with regards to the number of documents per pattern for the *adaptive* commits of the examined systems at minimum support of 5. 139

Table 8-7. Distribution of patterns with regards to the number of documents per pattern for the *non-adaptive* commits of the examined systems at minimum support of 5. 139

Table 8-8. Ratio of uncovered traceability patterns from adaptive commits, which are not valid for the non-adaptive commits of the examined systems. 141

Table 8-9. Elements of the KOffice source code, documentation and LSI settings used in the experiments. 143

Table 8-10. Recovered links, recall, and precision using cosine threshold for KOffice system 145

ACKNOWLEDGMENTS

I express my deepest gratefulness to my wonderful parents, *Mohammed* and *Khitam*, who did more than their best to raise, educate and support me. I am deeply gratitude to my beloved wife, *Hala*, who surely made a difference. My loving brothers and sisters have always been on my side.

I was fortunate to have come across many great individuals during my journey of graduate studies. My research advisor, *Prof. Jonathan I. Maletic*, was always there in times of great needs and deeds. I would like also to thank *Dr. Michael Collard* for his valuable contributions. I am deeply grateful for all my colleagues at *Software Development Laboratory (<SDML>)* and *Kent State University*.

Finally, I thank my dissertation committee. I really appreciate their efforts and participations.

Omar M. Meqdadi

August, 2013, Kent, Ohio

CHAPTER 1

Introduction

Today's software systems have great longevity. We continually evolve and maintain them to assure their relevance and usefulness to the user communities. One direct side effect of longevity is that dependent platforms, compilers, libraries, frameworks, and APIs also change. Maintenance to address such changing dependent systems is termed *adaptive*. Adaptive maintenance tasks involve changing a software system in response to changes in its environment. Examples include migrating a system to work on a new version of an operating system or to support a new API or software library.

Adaptive maintenance is somewhat unique in that its cause is most often outside of the control of the organization/developer. That is, hardware changes, new versions of the compiler, and changes to APIs are enhancements that come from a third party. Additionally, adaptive maintenance tasks are typically enterprise/system wide and impact large bodies of source code.

To highlight the impact of adaptive maintenance efforts let us examine a recent example. During the past couple of years almost all organizations that utilize Microsoft's .Net development platform undertook a major adaptive maintenance task. That is, they migrated from Microsoft .Net 1.1/2.0 to .Net 3.5/4.0. We have some general details of this effort for a major US insurance agency. Their migration involved over 60,000 projects (approximately 11 million source-code files) enterprise wide. The two-year

effort incurred an effort of approximately 75,000 person/hours and a cost of over $6.5 million USD. While this is a substantial sum, this effort was only part of the organization's ongoing adaptive maintenance projects (over $25M during the past couple years). While this is just one example, it is reflective of project for thousands of organizations, world wide, and the impact of these types of adaptive maintenance tasks on an organization's software development budget.

Unfortunately, there is little (automated) tool support for adaptive maintenance tasks and the vast bulk of the work is done manually, thus adding to its overall cost. One of the long-term research goals, of this dissertation, is to help provide more automated support for the adaptive maintenance process. To undertake this, we feel one of the first steps to achieve this goal is to better understand what adaptive changes look like and how they are applied in actual software systems.

That said, it would be very useful to identify all the past adaptive maintenance changes in an effort to understand what adaptive changes look like, construct a set of transformation rules, which would address the entire adaptive change for other systems or parts of the same system, and build regression testing approaches that can then be used to verify accomplished adaptive changes. Otherwise, the adoption of such maintenance changes by organizations will be put in question. That is, the main goal of identifying past adaptive changes is lowers the cost and increases the quality of performing adaptive maintenance tasks to large-scale software systems.

1.1 Recognizing Adaptive Changes

Software system artifacts are created in an inherently incremental manner via continuous change. They undergo changes due to factors such as feature additions, defect corrections, bug resolving, platform migration, and design improvements. That is, large software systems continually undergo corrective maintenance and enhancements during the time of the adaptive maintenance task.

Generally, large software systems progress over years of development history, where millions of lines of code are maintained by a set of expert developers. Changing a software system is normally documented, for the entire period of a project, in a version control systems such as *subversion* or *CVS*. The documented data are organized as commits. Each commit include metadata about the accomplished change such as who made the change, why the change was made, and when the change was done. As a result, this history is a helpful source of data for understanding and identifying the maintenance changes during the software evolution process.

Version history of a software system preserves all these undertaken maintenance changes. However, this version history does not keep a tag that would identify the purpose of each change. In other words, there is rarely enough detail to clearly direct a developer to the needed changes that are associated with a specific type of maintenance.

Due to this, teasing out the adaptive changes from the non-adaptive changes can be quite challenging. Therefore, automatically categorizing version history commits (during a specific time frame) into either one of two categories, namely adaptive commits and non-adaptive commits is motivated. The adaptive commits involved changes to the

system associated with the adaptive maintenance task being examined. The non-adaptive commits were all other commits to the system that involved different maintenance tasks.

1.2 Research Focus

The dissertation focuses on pushing boundaries of the automatically examining version histories to identify a set of adaptive modifications in response to a change in an API or new features of a compiler for existing systems, and disgusting them from other non-adaptive changes. Based on this identification, approaches can be developed to decrease the cost and increase the quality of understanding adaptive changes to large software systems, which is directly aimed at increasing the productivity of software developers. For instance, the changes found can be categorized based on the specific problem they are addressing. These variations can then be generalized in some manner in order to construct the transformation rule necessary for efficient application to new systems.

The dissertation addresses the following main adaptive maintenance research questions:

- What are the main characteristics of a large portion of adaptive changes?
- What new software measures and metrics can be computed to distinguish adaptive changes from other changes?
- Can adaptive changes, for a specific adaptive maintenance tasks be found via automated methods? What are the best ways to perform this automation?

- Can information retrieval methods be used on version history and identify specific type of maintenance changes?
- Can information retrieval methods be used to identify traceability change patterns that relate to adaptive maintenance?

1.3 Conducted Research

The bulk of the research involves a case study of three open source systems that previously underwent major adaptive maintenance tasks. We examined multiple years of version history of these systems to determine exactly which commits were involved in the particular adaptive maintenance task. This inspection was done manually and involved reading system documentation, development notes, commit log messages, and source code. It was a labor-intensive study that involved many hours of work.

The result of the study was a categorization of commits into either adaptive commits or non-adaptive commits. Thus, the study aimed directly at uncovering changes, which are known to have undergone a specific adaptive maintenance task. These results form the basis for gathering examples of a specific type of adaptive maintenance. The examples, along with documentation, can then be used to develop rules that can be applied to other systems to address the same adaptive maintenance task (e.g., apply the same API migration to multiple systems).

We then analyzed these adaptive maintenance commits to identify any commonalities or trends. The collected data from our investigation has been used to recognize a set of distinguishing characteristics that hold for a broader range of adaptive changes. Several characteristics of the commits are examined, namely the size of the

commits, the vocabulary of the commits' log messages, the developers who made the commits, and the stereotype of adaptive modified methods.

With these characteristics identified, a unique view of the actual evolutionary path taken to realize the adaptive changes implemented over evolution time of the software systems is provided. That is, this allows a much deeper understanding how developers typically commit adaptive changes to software repositories.

A number of clear trends in the uncovered adaptive commits were found that we feel can be leveraged to help automatically identify adaptive changes within a version history. A means to use information retrieval techniques, namely Latent Semantic Indexing (LSI), to automatically identify adaptive changes is developed and validated. LSI shows the ability to retrieve relevant adaptive commits, when querying the commits available in a version control system. Additionally, we propose a means to uncover a set of traceability links, between source code files and other artifacts, resulting from adaptive maintenance tasks.

1.4 Contributions

The dissertation work provides the following research contributions in the field of software maintenance and evolution:

- The development of a large benchmark data set of adaptive maintenance changes. This is the first in-depth and systematic examination of large adaptive maintenance tasks. The data provides a point of reference for the study of these types of changes.

- Identified a set of distinguishing characteristics for adaptive change commits. These factors appear to hold for a large percentage of adaptive commits and across the systems studied.
- Uncovered set of commonalities and trends that can help automatically identify adaptive changes within a version history.
- Developed an automated approach, centered on the LSI, to identify adaptive maintenance changes for existing systems.

The first research contribution (CHAPTER 3) and the second contribution (CHAPTER 4) are submitted to the 17th IEEE International Conference on Software Maintenance (ICSM'13). The third contribution (CHAPTER 7) is addressed and the result is written up for submission to the 20th Working Conference on Reverse Engineering (WCRE'13). The fourth research contribution (CHAPTER 8) is partially presented and published in the 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '13).

1.5 Organization

The dissertation is logically organized into two components: background on adaptive maintenance process, and identification and analysis of adaptive maintenance changes. Each chapter of the second half addresses a specific research issue regarding adaptive maintenance. The remainder of the dissertation is organized as follows. An overview of adaptive maintenance and its process is provided as a background material in CHAPTER 2. CHAPTER 3 manually investigates the version history of several systems to determine exactly which commits can be categorized as adaptive commits. An analysis

of the uncovered adaptive commits along with the measures of size, vocabulary, and developer is presented in CHAPTER 4. Understanding the effect of adaptive changes on different software artifacts is given in CHAPTER 5. A set of main characteristics of the stereotype for adaptive modified methods are examined and analyzed in CHAPTER 6. CHAPTER 7 details how LSI can help automatically identify adaptive commits within a version history. An automated approach to uncover traceability links between source code files and other artifacts for a particular adaptive maintenance task that involves the migration of an API is shown in CHAPTER 8. Finally, we conclude in CHAPTER 9 along with the discussion on open issues and future direction.

CHAPTER 2

A Background and Related Work on Adaptive Maintenance

Software maintenance is a wide action that includes all work made on a software system after it becomes operational. The maintenance is initiated by modification requests. Then, the software artifacts are modified, testing is performed, and a resultant version of the system is released. Thus, the reliability and the structure of the software system should not be corrupted by the software maintenance operations. In fact the bulk of the costs and resources of a system are allocated during the maintenance process [Lientz et al. 1978].

This chapter will present a brief background of software maintenance, adaptive maintenance, and adaptive change process. Additionally, the challenges of adaptive maintenance process will show the vital of automatically identifying adaptive commits from version history systems.

2.1 Software Maintenance Categorization

Software maintenance activities are generally classified into four main categories [Swanson 1976]. Mainly, this classification is based on the purpose of undertaken activities. All changes to the system can be described by these four categories of maintenance operations. The four categories are corrective, adaptive, perfective, and preventive.

Corrective category involves the uncovering and correction activates of software faults. A fault result when accomplished changes are inaccurately or incompletely implemented, or the modification request is misunderstands. Faults can be classified as design errors, logic errors, and coding errors. Generally, faults are often introduced during the maintenance process. The corrective maintenance is usually initiated by users reporting bugs [Anbalagan and Vouk 2009]. An example of corrective maintenance includes correcting a failure that prevents the software from correctly processing data.

Perfective maintenance improves maintainability and performance of the system according to the user requirements. This maintenance is directly aimed at making software system a more perfect design implementation [Swanson 1976] . A change due to the additional functionality or processing efficiency is classified as perfective maintenance activity. Of the mentioned types of maintenance, perfective maintenance is considered as the most frequently occurring type of maintenance [Pressman 1986; Schach et al. 2003; Sommerville 2004] . An example of perfective maintenance includes adding new graphical user interface to an existing software system.

Preventive maintenance deals with updating documentation and results in more maintainable software. Preventive maintenance is very in appearance with corrective maintenance. Therefore, some software researchers place preventive maintenance under the corrective category. This type of changes prevents systems from future problem with maintenance or to anticipate future perfective maintenance problems. An example of preventive change includes restructuring and optimizing code to enhance the future maintainability of the software system [Pressman 1986].

Finally, maintaining the system to execute in a different environment and conditions is called adaptive maintenance. Adaptive maintenance contains changes in the specification of the system, such as changes in the machine or operating environment, and changes in the implementation of the system. Thus, adaptive maintenance only modifies the implementation and the requirement of the operating environments. Therefore, such maintenance does not effect the formal specification of the system. Adaptive maintenance tasks are done less frequently than other types of software maintenance such as perfective maintenance [Schach et al. 2003]. An example of adaptive change includes changes arise due to changes in Microsoft Visual Studio development platforms.

This research is focused on adaptive changes. Specifically, their characteristics and how they can be automatically identified.

2.2 Adaptive Maintenance Overview

Formally, the adaptive maintenance is defined as a set of activates intended to transform software to adapt to changes in the hardware or software environment. Such maintenance involves such things as adapting the software to address changes to dependent platform versions, APIs, operating systems, frameworks, and new compiler or libraries [Collard et al. 2010].

Adaptive maintenance mainly deals with new or changed conditions which act from outside upon the system. Thus, accommodating these unique changes is frequently extra-ordinary and can only be predictable by monitoring the environment (e.g., new compiler version). However, a successful implementation of software tends to be

subjected to a sequence of adaptive changes in order to access new features and services that may be included in the new compiler version or APIs. Moreover, the consequence adaptive modification of a software system performed after releasing helps keeping the software system usable in the new environment (e.g., moving from Windows NT to Windows 7).

Adaptive maintenance changes are typically very repetitive. That is, a single change to a compiler or API may require that a system be modified to deal with this update in hundreds or thousands of locations in a system. While sometimes these changes are exactly the same in each location, more often they require some variation for each case dependent on the syntax or usage.

Therefore, adaptive maintenance normally requires exhaustive knowledge of the entire system code is often needed because this maintenance regularly requires many code changes throughout the system [Maletic and Reynolds 1994]. Moreover, the complexity of the adaptive maintenance is affected mostly by the adaptive change identification process [Kozlov et al. 2007]. Consequently, this knowledge demonstrates that adaptive maintenance developers are very likely to have good familiarity of the files considered to be adapted.

Furthermore, adaptive maintenance tasks are often delayed for a number of reasons in practice. Delays maybe for short term cost saving. But more often migration to a new operating system or compiler would be too large of an impact to the stability of a system for little benefit. That is, there would be too much risk involved in the maintenance task just to take advantage of a few new features of an OS or compiler.

However, if the software system is to stay relevant (in the market place or organization) it will eventually need to be maintained to run with these new platforms. Organizations may even choose to skip releases of an API or OS and migrate to the most current.

Performing adaptive maintenance tasks to large-scale software systems is a challenged process. Adaptive maintenance usually necessitates wide code changes span across a large portion of the system. Additionally, adaptive changes more prone to error when done manually, especially for large software systems, where changes are propagated to related entities [Malik and Hassan 2008]. Moreover, the manual adaptive maintenance may miss changing some code statements; since detecting the statement's location to be adaptively modified is a difficult task. Therefore, tools and approaches that can automatically execute adaptive changes would be of great benefit.

Adaptive maintenance tasks are done less frequently than other types of software maintenance such as perfective maintenance. For instance, Schach et al. in [Schach et al. 2003] stated that less than 13.8% of maintenance effort falls under the category of adaptive maintenance. However, almost all facets of daily life are impacted by software systems that regularly require such adaptive maintenance. Vendors eventually stop supporting the old environment versions, and organizations are forced to address the adaptive maintenance problem. This is to prevent customers from having to perform invasive workarounds. Examples include financial and insurance systems, factory automation, and desktop and smart phone applications. For instance, several past real world projects have undergone adaptive maintenance changes due to a compiler/platform migration. This includes:

- Porting the KDE (K Desktop Environment) system from Qt3 to support Qt4.
- Modifying ABB industrial systems to deal with changes in the VxWorks and Microsoft Visual Studio (2003 to 2005) development platforms.
- Migrating OpenSceneGraph (3D graphics application) from operating with OpenGL 2.0 to OpenGL 3.0.

2.3 Adaptive Maintenance Process

Adaptive maintenance undergoes modification to address changes to dependent platforms, APIs, frameworks, and libraries. Thus, developers must be trained to use different APIs or compiler features in sequence to migrate the system to a new platform or new environment.

That is, developers must search for changes in the usages of old environment features and interfaces that were changed to the new features and interfaces found in the new environment. The documentations for the new releases of APIs, compilers, and operating systems typically have lists of the major changes and some basic suggestions for what needs to be corrected to support new features and interfaces [Nita and Notkin 2010].

To help understand the adaptive maintenance process, Figure 2-1 presents a detailed scenario of performing such maintenance. After identifying the new features and interfaces in new platform, developers must specify the location of the proposed change in system original code. For a given maintenance task, the area of the code that needs to be changed must be identified (e.g., specific method or statement).

Afterward, given the particular statement/location to be changed, developers must identify the appropriate code adaption transformer. The transformation describes how the code is modified, including any additions and/or deletions to the code [Collard et al. 2010]. The generation of the new target code often requires information from the original code, some of which may not be explicit, and static analysis of the code may be required to obtain this pertinent information and to construct the necessary modifications. That is, the new generated code can include literal text for added code, copies of parts of the original code. Figure 2-2 show an example of code adaption changes that were applied to KDE for the migration from Qt3 to Qt4.

The adaptive modifications can then be verified through testing. Regression testing can be done to identify any errors that happen due to the changes [Collard et al. 2010]. The essential of this testing is due to the fact that adaptive maintenance is an error prone task and the impact of such changes propagate through the system code. As shown in Figure 2-1 , the adaptive maintenance process is a repetitive one.

2.4 Adaptive Maintenance Challenges

To undertake the adaptive maintenance process, a number of intellectual challenges must be overcome. This maintenance is a very costly and error prone task, where code adaption process involves a large amount of statement modifications. For instance, in total, 1756 adaptive maintenance changes were made to accomplish due to a C++ compiler migration from Visual Studio 2003 to Visual Studio 2005 [Collard et al. 2010]. In this example, nearly 304 adaptive changes were missed by the developers during the manual modification process.

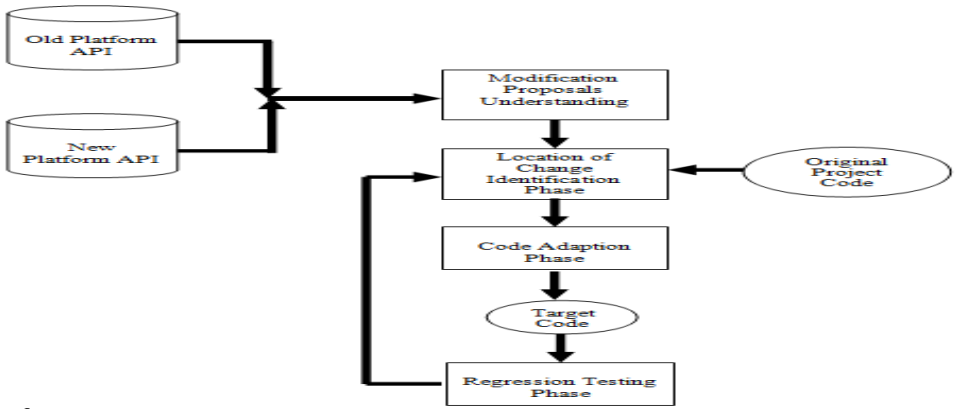


Figure 2-1. Adaptive maintenance process.

```
QString text;
m_listWidget = new QListWidget(page);
QStringList tabsList = m_pView->doc()->map()->VisibleSheets(false);

QStringList::Iterator it;
list = new QList(page);
for (it = tabsList.begin(); it != tabsList.end(); ++it)
{
     text = *it;
     list->addItem(text);
}
m_listWidget->addItems(list);
if (!list->count())
     enableButtonOk(false);
```

```
QString text;
m_listWidget = new QListWidget(page);
QStringList tabsList = m_pView->doc()->map()->hiddenSheets();

m_listWidget->addItems(tabsList);

if (! m_listWidget->count())
      enableButtonOk(false);
```

Figure 2-2. An example of a migration from Qt3 to Qt4, which was applied to the KDE project.

On the other hand, manually migration may lead to a rejection of new bugs and faults prone when developers did not all of changes correctly or if they unconsidered any possible change propagation to other entities in the system.

Moreover; there are a large number of variations on a change to address the different syntactic contexts in the programming language where the change is taking place [Collard et al. 2010]. Furthermore, even given a complete set of necessary items to change to support an adaptive maintenance task it will not cover all situations within the context of all software systems. That is, it is very difficult to predict all the usages of an API or feature for any given system. Conversely, if given a set of changes that are known to be involved in some adaptive change, developers must determine if all changes address the same problem or not. As such developers must identify all syntactic usages of each location to be changed in the context of the system.

That is, the mentioned adaptive maintenance challenges motivate the essential of automatically applied adaptive changes in order to decrease the cost and increase the quality of applying adaptive changes in the context of large scale systems [Waddington and Yao 2005].

To undertake the automatically adaptive modifications, we should automatically support generating transformation rules that address a specific adaptive maintenance problem, where these rules describe how the code is modified, including any necessary additions and deletions to the code in the situation of adaptive maintenance [Zhong et al. 2010] .

The main idea of the transformation rules construction centers on how to go the adaptive changes found in past versions to examples. These examples will be used, in conjunction with a syntactic differencing tool, to generate the target transformations, which can be applied to other system that yet, need to undergo the same adaptive change.

That is, automatically identifying adaptive changes, and disgusting them from other non-adaptive changes, in an existing system is the preliminary and essential step to generate adaptive transformation rules. Otherwise, the adoption of such automated transformation techniques will be put in question.

Automatically identifying adaptive changes, which are found in past versions, can be accomplished by examining the version history of systems for adaptive commits. Once the adaptive commits are identified, all associated source code changes can be determined. The changes found must then be organized to identify the syntactic variations of the given changes in the context of the programming language. This is aimed at categorizing the changes and determining which address the same problem. These variations must be generalized in some manner in order to construct the transformation rule necessary for efficient application to new systems.

Therefore, the work presented in this dissertation represents a supplementary attention to help the software research that concerns the invitation of adaptive transformation approaches. The proposed work will investigate how to automatically identify changes from the version history of systems that are known to have undergone a specific adaptive maintenance task. These changes can be used to reveal valuable

information about the adaptive evolutionary history of the systems to support transformation approaches.

2.5 Automated Adaptive Process Background

The issue of identification of locating where in the code to apply the transformation is tackled as vital issue for the success of transformation approaches. The identification of adaptive commits aims at supporting adaptive transformation tasks. To be precise, the code changes, which are associated with automatically identified adaptive commits, will be used as examples to automatically support generating transformation rules that address a specific adaptive maintenance problem.

It has been observed that automated source code transformations to enhance adaptive maintenance actions intended to be handled by a numerous of impressive researches a couple years ago. In [Waddington and Yao 2005] , the authors considered the issue of performing high-fidelity transformation in large C++ systems. They developed a new language called YATL for transformation .This language is based on a specialized form of Abstract Syntax Tree (AST), to allow for lexical detail that contained in higher-level abstractions. Their transformation was successfully validated over millions of code lines of C++ commercial source code in a version managed environment.

Automated transformation for adaptive changes due to language migration is explored by several researchers such as the work in [El-Ramly et al. 2006]. Here, a source-to-source transformation was proposed. The authors presented a new Java2C# tool that converts legacy Java code to C# (modern language). The proposed transformation is

written in TXL language based on tree-rewriting. The aim of using TXL is its ability for incremental updates of the language transformer where the transformation rules can be simply updated. A set of examples and challenges of such migration are presented deeply in this work.

Mapping between two API libraries is searched by Zhong et al. [Zhong et al. 2010] . In this work, the authors proposed a new approach called MAM (Mining API Mapping) to supply API mapping relations from one language to another using API client code. MAM performs API migration from existing projects with multiple versions in different languages. In this work, API Transformation Graphs (ATG) is proposed to describe data dependencies among inputs and outputs of API methods. By comparing two ATG, the approach can mine the migration between two different APIs. The evaluation results show that the proposed tool mines 25,805 unique mapping relations of APIs between Java and C# with about 80% accuracy, 54.4% compilation errors reduction, and 43.0% defects decreasing when compared with other existence tools.

Nita et al. [Nita and Notkin 2010] proposed a twinning approach to adapting programs to alternative APIs. The proposed approach allows the programmers to determine changes that migration a program between two different APIs. With this twinning approach, a mapping specifying how two APIs correspond to one another is created using abstract syntax tree. Then, the mapping is applied to the base code, which use the original API, to get a variant of it that uses new API. The identified changes can be imported by applying the resultant mapping to a later version in order to reduce the migration overhead. The main advantage of this twinning migration is using same

interface to call up either the old code or the replacement (base on new API) code. The approach was evaluated through the correctly migration between using the Crimson XML parsing API to use the Dom4J API instead. Additionally, the approach was efficiently adapting Twitter client to use the Face Book API instead.

A lightweight transformation approach to automate adaptive maintenance changes on large-scale software systems using srcML toolkit is proposed in [Collard et al. 2010] . In this approach, locating the source code lines that need changing is done automatically using XPath expressions in order to automate parts of the manual process. Then, the transformation rules describe how the code should be adaptively modified, including any additions and/or deletions to the code. In this study, the rules are written in XSLT since it allows expressing the location of the needed transformation in XPath. In this work , the authors deeply addressed a number of vital adaptive changes including the modifications of the new operator, template classes that require specialization, iterator variable scope modification, deprecated string functions, STL vector concerning access to the start of vector data, and fully-qualifying function pointers. The proposed approach was evaluated using APIs large adaptive changes that occurred a several years ago at ABB Inc. The causes of investigated adaptive maintenance are the changes in the VxWorks (C++ standard) and Microsoft Visual Studio development platforms. In this evaluation, automatically transformed codes by the proposed approach were compared with the manually changed code performed by the professional developers at ABB Company. The results show that the proposed approach correctly modified all cases (except for two

cases of the private data problem) that were manually done. Furthermore, the approach can capture and correctly modify about 13% of cases that the developers missed.

2.6 Summary

Software maintenance changes are classified into four categories, namely: corrective, adaptive, perfective, and preventive. All modifications to address changes to dependent platforms, APIs, frameworks, and libraries, are generally termed adaptive maintenance tasks. This adaptive maintenance is done less frequently than other types of maintenance such as perfective maintenance. However, because of the negative impact of adaptive maintenance postponement, organizations are forced to react to required adaptive changes as they are discovered.

To undertake an adaptive maintenance task, developers must discover all code statements to be changed. Then, they implement the necessary code adaption changes using the proper transformation, and test the new code by regression testing. This process is a repetitive task.

Performing adaptive maintenance process to large-scale software systems is a challenged practice. This motivates the important of automatically applied adaptive changes through development of the efficient transformation rules. Searching for before and after examples of the adaptive changes, which is done by examining version histories of adaptive commits and changes, will guide the development of required transformation rules. Thus, automatically identifying a set of adaptive commits for existing systems is a vital concerning regarding the quality of adaptive maintenance process.

CHAPTER 3

A Manual Identification of Adaptive Commits

The future long-term maintenance of the systems undergoing adaptive changes is a primary concern. The main goal of this concern is lower the cost and increase the quality of performing adaptive maintenance tasks to large-scale software systems, which is directly aimed at increasing the productivity of software developers. Currently, software community aims at pushing the boundaries of the adaptive maintenance automation in two directions. The first is to automatically identifying a set of adaptive changes for existing systems. The second is to automatically generate the transformations from these examples.

To undertake these two research components, previous adaptive changes must be identified. With these uncovered changes, all syntactic variations can be identified to form a basis for the automatically adaptive transformations and identification approaches. For instance, uncovered syntactic differencing is used to generalize the examples and generate the transformation. Therefore, the first essential task involves examining version histories to identify undertaken adaptive changes.

For that reason, we performed a case study of three open source systems that previously underwent major adaptive maintenance tasks. We inspected multiple years of version history of these systems to identify the commits that are associated with these

undertaken adaptive tasks, and differentiate them from non-adaptive commits, which involves underwent corrective, perfective, and enhancements maintenance activities.

This undertaken examination was done manually and involved reading system documentation, development notes, commit log messages, and source code. It was a labor-intensive study that involved a lot of work.

This chapter will address the problem of differentiation commits that contain adaptive changes from those commits which represent other maintenance categories. We now present the details of the systems along with the reasons behind the choice of each and a brief background on version control systems and the commit process. Lastly, we present the adaptive commits that were collected as an input for adaptive changes analysis.

3.1 Software Repository commits

Our study is rooted in investigating the software repositories. These repositories hold valuable information and provide a unique view of the actual evolutionary path taken to realize the maintenance changes implemented over evolution time of the studied systems. Actual past changes can be identified by analyze multiple versions of the software system. That is, this gives a better understanding of how developers typically commit adaptive changes to these repositories.

Version Control Systems, such as Subversion, are standard tools that preserve changes to source code artifacts during the development and maintenance of software systems. Version number assignment and metadata are associated at the change-set level, and recorded as a log entry.

Figure 3-1 shows a log entry from the Subversion repository of KOffice. A log entry corresponds to a single commit operation. This commit log information can be readily obtained through 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. In this example, 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*. Note that the order in which the files appear in the log entry is not necessarily the order in which they were changed.

```
<?xml version="1.0" encoding="utf-8"?>
<log>
<log entry 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>
</log entry>
</log>
```

Figure 3-1. A subversion log entry from KOffice system. The entry includes the revision, the files in the change set, and a developer description of the change.

Additionally, a text message describing the change entered by the developer is also recorded. The purpose of applied change may be indicated by the terms in this message.

3.2 Subject Systems

The goal of this case study is simple. Examine the version history of a software system and determine which commits to the system are concerned with a specific adaptive maintenance task. We selected three open source systems that were known to have undergone a major adaptive change in their history. Knowing the adaptive maintenance task allowed us to narrow down the time frame of version history to examine for each system.

As a case study, we investigated and studied three large open source systems, the office-applications suite *KDE/KOffice*, graphical applications associated with the KDE project[1] in the package *KDE/Extragear/graphics*, and the high-performance 3D graphics toolkit *OpenSceneGraph* (OSG)[2]. These systems cover a number of application domains and sizes, and are prime examples of successful open-source development that utilizes a number of different APIs. Many of these third party APIs are regularly updated thus requiring migration to new API versions. Here, we present a brief background of the systems along with the reasons behind the choice of each.

---

[1] See http://www.KDE.org

[2] See www.openscenegraph.org

3.2.1 KOffice System

KOffice is an office-applications suite associated with the KDE project. This system consists of 12 main applications. KOffice uses Qt for the windowing and interface for the line of office produces including a word processor, spreadsheet, database, etc. Qt is an open source platform independent framework for developing graphical user interfaces and is widely used by many open source projects. Qt uses standard C++, but makes a broad use of Meta Object Compiler, and special code generator, simultaneously with numerous macros to enhance the programming language. In 2004 a major new version of Qt, namely Qt4, was released[3].

In early 2006, the developers of KDE project started to initiate the porting of the all the KDE systems to support Qt4 (i.e., move from Qt 3.x to Qt 4.0). The process of moving KOffice completely to Qt4 lasted until the end of 2010, as indicated by the developers commits[4]. This included dealing with another substantial release of Qt, such as Qt 4.7. That is, we are looking for adaptive maintenance related to the migration of KOffice package from Qt3 to Qt4.

During this time period, there were nearly 38,000 commits (in subversion) to the KOffice project. Out of these commits, a subset was performed so that the system persists to operate mutually with using the new Qt4 features and services. This subset represents the undertaken adaptive maintenance during examining time period.

---

[3] See http://doc.qt.nokia.com/4.0/porting4.html

[4] See http://websvn.kde.org/trunk

3.2.2 Extragear/graphics System

Extragear/graphics is a collection of graphical applications that are associated with the KDE project. This graphical package consists of 8 main components. Similar to KOffice, Extragear/graphics uses Qt for the windowing and interface for the line of graphical productions.

However the Qt 4 was released in 2004, Extragear/graphics did not start to move to the new API's until the end of 2006. At the end of 2010, the system runs the new features and interfaces found in the Qt4. For this system, we investigated the subversion commits during the time period of 1/1/2006 to 12/31/2010 to recognize the commits associated with the mentioned porting process. In this time period, there were around 26,000 commits.

3.2.3 OSG System

The OSG project is a high performance 3D graphics toolkit. This toolkit is used as an application programming interface in the field of developing several 3D applications such as visual simulation, virtual reality, and computer games.

This toolkit is written in C++ using OpenGL specifications. OpenGL is a software interface to graphics hardware. OpenGL commands offer an abstract API for drawing 2D and 3D graphics and is broadly used by several scientific visualization and modeling systems.

In August 2008, a major new version of OpenGL, namely OpenGL 3, was released[5]. The adaptive maintenance related to the migration to OpenGL 3 was completed at the end of 2010, as mentioned in the commit annotations the OSG developers[6]. Hence, we investigated the commits of OSG in the time period of August 1, 2008 through December 31, 2010.

3.3 Manual Investigation of Adaptive Commits

Adaptive changes in response to a change in an API or new features of a compiler are often done in a systematic manner. The changes are often well planned and applied to large portions of the system within one release (or small number of releases). While identifying which specific releases are focused on adaptive changes, teasing out the adaptive changes from the non-adaptive changes can be quite challenging. However, since identifying previous adaptive changes represents the root of any future automation of adaptive maintenance, we performed a manual investigation to identify all commits associated with this type of maintenance.

3.3.1 Investigation Approach

The text message of any commit describes the change undertaken by the developer and what the intended purpose of the change was [Hindle et al. 2009].

---

[5] See http://www.opengl.org/registry/#oldspecs

[6] See www.openscenegraph.org/svn/osg/OpenSceneGraph

Therefore, it is evident that with the commit message text we can derive that this change contains an adaptive API task or not.

Accordingly, the subversion repository is a helpful source of data for uncovering the adaptive maintenance tasks during the software evolution process. That is, recognizing adaptive changes using only commits metadata is attractive because it does not involve retrieving and then analyzing the maintained source code of the commit [Hindle et al. 2009].

For the case study, we extracted the change log file associated with each studied system using the *svn log* command. Then, we manually read and investigated each commit message in the change-log file to determine if changes occurred in that commit that were adaptive changes to the specific APIs modifications we are investigating. Here, the adaptive changes involved porting from old platform (e.g., Qt 3.x) to support the new platform (e.g., Qt 4.x). The adaptive modifications can be identified by searching the commit log messages for indications that old platform features/interfaces were changed to support new features/interfaces found in new platform.

For instance, since the class *QButton* in Qt3 was replaced by the class *QAbstractButton* in Qt4[7], searching for these classes in the commit log messages is a criterion indicating adaptive commits associated with this task. Lists of changing features and classes from Qt3 to Qt4 are listed in the system's online documentation. We

---

[7] See http://doc.qt.nokia.com/4.0/porting4.html

used the same basic process to locate the relevant adaptive changes in OSG for the migration to OpenGL 3[8].

We demonstrate that standard well known XML technologies can be used to address the identification in a cost effective manner. Therefore, the XML format of the log file, as shown in Figure 3-1 , is used as the input to our manual investigation. This XML format can be generated using following *svn* command:

svn log URL -r {starting_Date} :{end_date} –xml –v>output.xml

Furthermore, in the retrieved commits, the candidate adaptive change commits are then inspected and verified that they were indeed an adaptive change. While we may have missed some adaptive changes, we feel fairly confident that all were located for this time period.

That is, this manual approach was performed over the extracted commits of the mentioned open source systems.

3.3.2 Summary of Findings

Our investigation shows that the vast majority of the commits during the studied time period had nothing to do with the API migrations. The majority of commits addressed other type of maintenance such as corrective, adding new features, or enhancements tasks going on in parallel. A summary of the obtained results is given Table 3-1.

---

[8] See http://www.opengl.org/registry/#oldspecs

Table 3-1. Adaptive and none adaptive commits for the three examined systems over the given time period.

	KOffice	Extragear/ graphics	OSG
Adaptive Maintenance Task	Migration from Qt3 to Qt4	Migration from Qt3 to Qt4	Migration to OpenGL 3
Adaptive Task Starting-Date	03/29/2006	11/07/2006	09/18/2008
Adaptive Task Ending-Date	12/31/2010	12/31/2010	12/31/2010
Total Number of Commits	38,980	26,336	4,310
Number of Non-Adaptive Commits	38,849 (99.3%)	26,117 (99.2%)	4,231 (98.2%)
Number of Adaptive Commits	131 (0.3%)	219 (0.8%)	79 (1.8%)

It was a bit surprising how few actual commits were involved in these major API migrations. However, as we will show in the next chapter, the commits were on average quite large, compared to a typical commit [Alali et al. 2008], and each impacted a large number of files. Typically, a single adaptive commit involved addressing one part of the migration for the entire software system. That is, the migration process was done incrementally but system wide. The committer (developer) normally grouped the same types of adaptive transformations into a single large commit. Additionally, adaptive maintenance tasks normally do not involve large complex changes to functionality so often require few specific changes but applied in many locations.

On the other hand, Table 3-2 and Table 3-3 shows that the ratio of commits involved with the adaptive maintenance decreased over time. This further supports that the systems were incrementally migrated to new API versions, as the API versions were released, e.g., Qt 4.1, Qt 4.2 etc.

Table 3-2. Distribution of adaptive commits per year. Percentages are compared to total adaptive commits.

	Ratio of Adaptive Commits Per Year		
Year	**KOffice**	**Extragear/graphics**	**OSG**
2006	40.5%	3.4%	0.0%
2007	24.2%	37.6%	0.0%
2008	14.1%	32.3%	50.3%
2009	11.8%	21.8%	37.1%
2010	9.4%	4.9%	12.6%

Table 3-3. Distribution of adaptive commits per year. Percentages are compared to total commits (adaptive and non-adaptive).

	Ratio of Adaptive Commits Per Year		
Year	**KOffice**	**Extragear/graphics**	**OSG**
2006	0.67%	0.17%	0.00%
2007	0.38%	1.73%	0.00%
2008	0.31%	1.47%	1.42%
2009	0.27%	0.72%	1.29%
2010	0.15%	0.32%	0.86%

After identifying the commits involved in the adaptive changes, we examined the main characteristics associated with a broad range of uncovered commit. In the reminder of this work, we will present how to use these commits to hypothesize that a large portion of adaptive changes, for a specific adaptive maintenance tasks (e.g., port of Qt) can be found via automated or semi-automated methods. Additionally, the outcomes of this manual investigation will aim in comparing adaptive maintenance activities with other maintenance tasks, as we will see in next chapters.

3.4 Manual Investigation Challenges

During the manual identification of adaptive commits, a number of intellectual challenges were faced. First, manually identifying adaptive commits was a very costly task. For instance, during the time period of January 1, 2006 to December 31, 2010 there were over 38,000 commits (in subversion) to the KOffice project. Hence, *the manual investigation took several months* of arduous work to accurately differentiate the adaptive commits from non-adaptive.

Secondly, there is no standard way as how developers commit changes. For example, instead of use one commit to solve one problem, developers implement same task through contiguous commits. As a result, some commits contain adaptive and other types of maintenance actions, while the accomplished adaptive task is not mentioned in the corresponding commit message.

Finally, the accomplished manual investigation was performed on prime examples of successful open-source development that involve a lot of adaptive maintenance.

However, we do not assert that this investigation be able to function with a high accuracy to a broad range of systems, such as the closed source systems or commercial systems.

3.5 Summary

We performed a manual investigation to identify changes from the version history of systems that are known to have undergone a specific adaptive maintenance task for three large open source systems. This investigation was accomplished by searching through the commit log messages and manual verifying that they were indeed an adaptive change. The vast majority of the commits during the examination time period did not have anything to do with the adaptive maintenance.

This mined information, (i.e., uncovered commits) will be used to study and explore how to construct automatic identifier of adaptive commits. Moreover, these commits will be used to perform a comparison study between adaptive and non-adaptive changes.

CHAPTER 4

Characterizing the Adaptive Commits

To better understand adaptive changes and how they are applied in actual software systems, it would be very useful to characterize the collected adaptive commits. Hence, after collecting the commits involved in the adaptive changes from the version histories, we examine the main characteristics associated between large portions of these commits.

The objective of this analysis is to develop a means to automatically identify commits involved in adaptive changes. That is, analyzed collected adaptive maintenance commits to identify any commonalities or trends can be leveraged to help automatically identify adaptive changes within a version history.

We take a closer look at the three sets of adaptive commits and attempt to uncover any similar characteristics and trends. First the commits are categorized based on how they impact the system. That is, what was added, deleted, or modified in each commit. Next, we examine the vocabulary of the commits' log messages to discern any trends or commonalities. Then, we investigate which and how many developers were actually involved in these adaptive commits. Lastly, we investigate the correlation between the developers and word distributions in their commits.

These three characteristics (i.e., size, vocabulary, and authorship) were selected for a variety of reasons. Given the raw data (version information) these measures are

readily available from the log entries. There is little else that one can compute from the commits beyond this information.

It has also been demonstrated [Alali et al. 2008] that the commit size and log message are correlated. Mockus and Votta [Mockus and Votta 2000] also discovered that the text description field of a change and the change size are essential to understanding why that change was performed. Hindle et al. [Hindle et al. 2009] showed that the author identity may be significant for predicting the type of a change. They also propose these same measures, size, author, and message terms to identify the type of each commit.

In this chapter, we present our analysis of commits along the measures of size, vocabulary, and developer.

4.1 Adaptive Commit Size Categorization

To examine the size of the adaptive commits in the context of how many items were added, delete, or modified within each commit, the following three size-based measures were used:

- *Method*: Number of methods/functions added, deleted, or modified.
- *File*: Number of files added, deleted, or modified.
- *Module*: Number of directories that contain a change in the commit.

These measures are used to determine the overall impact a given commit has to a system. That is, a given commit (change) can be much localized to one file or function. Alternatively, it can be system wide impacting a large number of files. The goal is to characterize what a typical adaptive commit looks like. Or, more importantly, the goal is

to see if there is even a typical adaptive commit. The file and module size measures are inexpensive to calculate, as only the log entries need to be analyzed.

For each commit, the three size measures are computed. We use the values of these measures as the data points for classification. To aid such classification, a descriptive-statistics method was used to classify the commits into different categories based on the number of files, methods, and modules being changed.

We use a 5-point summary approach, the same approach that was used for commit categorization in [Alali et al. 2008] This approach divides the dataset up into regions using the following Quartiles:

- Q0 ≡ Quartile 0 (minimum observation).
- Q1 ≡ Quartile 1 (upper edge for the smallest 25%).
- Q2 ≡ Quartile 2 (median).
- Q3 ≡ Quartile 3 (upper edge for 75%).
- Q4 ≡ Quartile 4 (maximum observation).

The Inter Quartile Range (IQR) is the data points covered in the range of quartiles Q3 and Q1 (i.e., Q3 – Q1). We use these quartiles to define five equal regions as extra-small, small, medium, large, and extra-large as shown in Figure 4-1.

Then, we classify the size of each commit based on these regions. This categorization can then be displayed using a boxplot method.

We developed a tool to apply the above commit categorization over the collected commits, both adaptive and non-adaptive, from the version history of the examined

systems in the study. The method-size, file-size, and module-size per commit can each be distributed over the defined five regions in Figure 4-1.

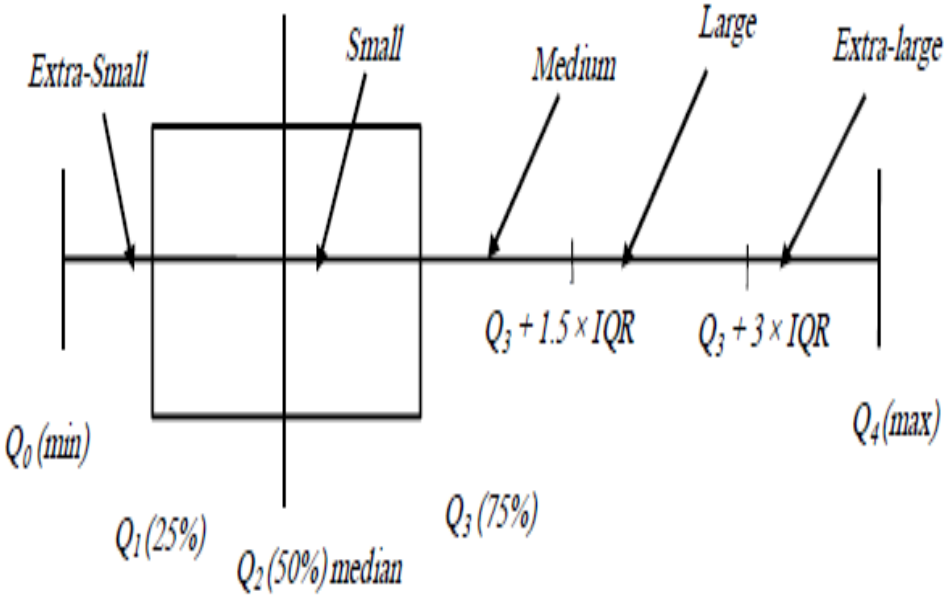


Figure 4-1. A box plot showing the five segments, which are created by the five-point summarization commits into five categories.

Here, we examined the following questions:

- What is the distribution of adaptive-change commits?
- Do adaptive-change commits cause more files to change than other commit types?
- Can we classify the adaptive-change tasks as a system-wide maintenance operation?

Table 4-1, Table 4-2, and Table 4-3 provide a summary (the data, quartiles) of the commit categorization using the file-size measure. The *Range* columns of these tables represent the boundaries of each defined region for the corresponding boxplot.

Table 4-1. Commits categorized by file-size using five-point summarize approach for KOffice system.

	KOffice		
Quartile	**Number of Files**		
Q0 (*Min*)	1		
Q1	1		
Q2 (*Median*)	2		
Q3	4		
Q4 (*Max*)	3806		
IQR	Q3 - Q1= 3		
Boxplot	**Range (#files)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-1	24.11 %	57.33 %
Small	2-4	32.59 %	23.30 %
Medium	5-8	21.69 %	10.57 %
Large	9-12	9.40 %	4.92 %
Extra-Large	>=13	12.21 %	3.88 %

Table 4-2. Commits categorized by file-size using five-point summarize approach for Extragear/graphics system.

	Extragear/graphics		
Quartile	**Number of Files**		
Q0 (*Min*)	1		
Q1	1		
Q2 (*Median*)	2		
Q3	3		
Q4 (*Max*)	3023		
IQR	Q3 – Q1= 2		
Boxplot	**Range (#files)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-1	29.61 %	58.95 %
Small	2-3	36.10 %	27.26 %
Medium	4-6	15.68 %	7.16 %
Large	7-9	10.02 %	4.61 %
Extra-Large	>=10	8.59 %	2.02 %

Table 4-3. Commits categorized by file-size using five-point summarize approach for OSG system.

	OSG		
Quartile	**Number of Files**		
Q0 (*Min*)	1		
Q1	1		
Q2 (*Median*)	2		
Q3	3		
Q4 (*Max*)	489		
IQR	Q3 – Q1= 2		
Boxplot	**Range (#files)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-1	24.14 %	54.89 %
Small	2-3	36.23 %	24.53 %
Medium	4-6	18.92 %	11.10 %
Large	7-9	9.32 %	5.44 %
Extra-Large	>=10	11.39 %	4.04 %

The results show that the majority of the non-adaptive commits fit into the small and extra small categories of the examined systems. However, larger commits (large and extra-large categories) occur with a higher percentage for adaptive commits. For

instance, about 21% of adaptive commits, compared with 9% of non-adaptive commits, are classified as large commits for KOffice.

Given this trend, the adaptive maintenance tasks on average touch more files than non-adaptive tasks. That is, developers performed an adaptive task by adding/deleting/modifying code statements across the entire system, and then committed these changes using one large commit. Therefore, the file-size measurement provides an explanation as to why the number of adaptive commits is much smaller then the number of other non-adaptive maintenance commits.

In an attempt to get a more detailed picture, we further processed the files to the granularity of methods/functions. We use the GNU Unix *diff* utility to identify modified, added, and deleted methods/functions occurring in each commit of the examined systems. Our observation, for the studied systems, is that nearly 27% of the adaptive commits are in the large or extra-large categories, while only approximately 7% of the non-adaptive commits are large or extra-large. For instance, this observation for studied systems is shown in Table 4-4, Table 4-5, and Table 4-6.

The boundaries of the defined regions using the module-size measure are shown in Table 4-7 for all systems. The outcomes of commit categorization histograms, such as the one shown in Figure 4-2, show that the module-based results reflect the fact that the adaptive changes were often system wide. That is, changing the GUI framework impacted the entire system in a similar manner.

Table 4-4. Commits categorized by method-size using five-point summarize approach for KOffice system.

KOffice			
Boxplot	**Range (#methods)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-2	9.20 %	22.06 %
Small	3-8	22.69 %	46.48 %
Medium	9-18	40.21 %	23.39 %
Large	19-28	16.74 %	4.44 %
Extra-Large	>=29	11.16 %	3.63 %

Table 4-5. Commits categorized by method-size using five-point summarize approach for Extragear/graphics system.

Extragear/graphics			
Boxplot	**Range (#methods)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-2	10.60 %	24.11 %
Small	3-9	27.32 %	49.54 %
Medium	10-17	35.50 %	19.76 %
Large	18-25	15.66 %	4.21 %
Extra-Large	>=26	10.92 %	2.38 %

Table 4-6 . Commits categorized by method-size using five-point summarize approach for OSG system.

OSG			
Boxplot	**Range (#methods)**	**Ratio in Adaptive commits**	**Ratio in Non-Adaptive commits**
Extra-Small	1-2	14.68 %	33.27 %
Small	3-6	26.51 %	38.17 %
Medium	7-12	37.29 %	19.58 %
Large	13-18	12.78 %	7.03 %
Extra-Large	>=19	8.74 %	1.95 %

Table 4-7. Defined categories by module-size using five-point summarization.

Range (Number of modules)			
Boxplot	**KOffice**	**Extragear/graphics**	**OSG**
Extra-Small and Small	1-2	1-1	1-1
Medium	2-6	2-4	2-3
Large	7-11	5-7	4-5
Extra-Large	>=12	>=8	>=6

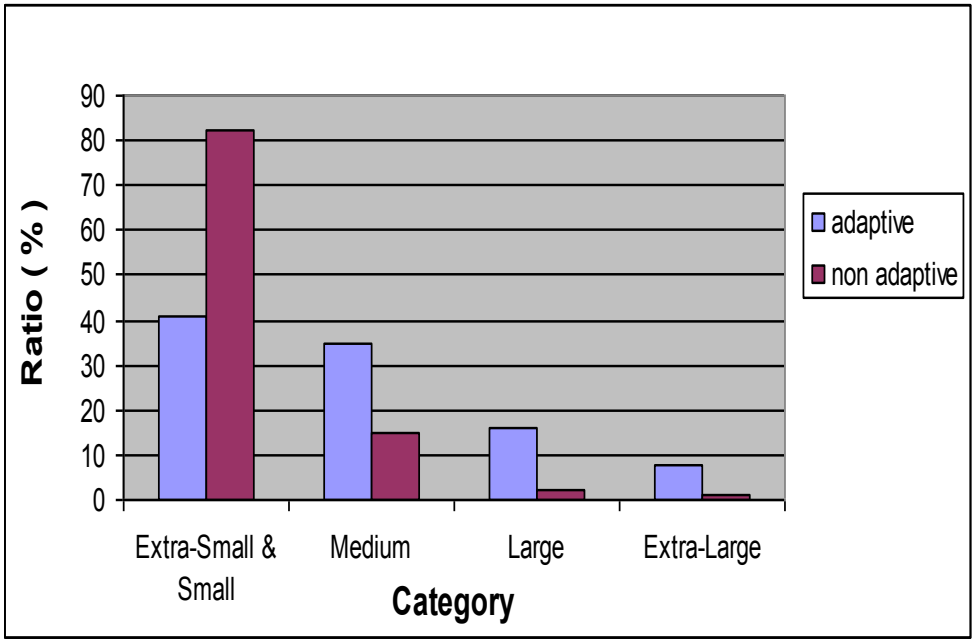


Figure 4-2. Histogram for the number of commits distributed over number of modules for Extragear/graphics system.

In order to better comprehend the differences between adaptive and non-adaptive with regards to the three size-based measures (e.g., file, method, and module), we calculated the level of significance of these measures using the *Mann-Whitney U* non-parametric tests [Hinkle et al. 1995]. In this statistical testing, a p-value of 0.05 represents the significance level [Hinkle et al. 1995]. That is, if the p-value is greater than 0.05, then we assume there is no observable difference between adaptive and non-adaptive commits.

The results of this test are provided in Table 4-8 . Based on the results presented in this table, we can conclude that there is a significant difference between the adaptive commits as compared to the non-adaptive commits from the perspective of file, method and module-size based measurements (all the p-values are less than 0.05). These results

are an additional motivation to use the mentioned size measures to build an automated classifier between adaptive and non-adaptive commits using such measures.

Table 4-8. P-values for the three size measures using Mann-Whitney U tests for the three examined systems

System	P-values		
	File	Module	Method
KOffice	0.0347	0.0463	0.0477
Extragear/graphics	0.0393	0.0434	0.0421
OSG	0.0414	0.0479	0.0466

4.2 Adaptive Commit Vocabulary

We now examine the vocabularies of the commits' log messages and examine the frequency of term usage. The goal is to understand the importance of specific terms for possible automatic identification of adaptive commits. A vocabulary is built of the most frequent words used in adaptive commits' log messages over the studied systems. For this examination, we developed a tool based on the approach proposed in [Alali et al. 2008].

For each system, we collected the adaptive commits' log messages and eliminated stop words using the Lovins stemmer algorithm [Lovins 1968]. This algorithm removes suffixes from words using a predefined list of about 250 different suffixes. The longest

suffix attached to the word is removed, where the stem after the removed suffix is at least three-characters long. We take the stemmed words and count the frequencies of each word by considering each commit as a customer basket [Srikant and Agrawal 1996].

The result is a ranked list of frequent terms for each system. Then we cross-join the three resultant lists from the examined systems and take the top 12 most frequent terms. For each term of the top 12 terms, we counted its occurrences in all non-adaptive commits for each of the three systems and find the corresponding occurrence average.

Table 4-9 has a list of the top 12 terms used across adaptive commits of the examined systems, in addition to their corresponding ranks in non-adaptive commits of these systems. In particular, we see that even over different systems, with different migration tasks (Qt and OpenGL), there is commonality of terms and frequency of use.

Similarly, Table 4-10 has a list of the top 12 terms used across non-adaptive commits and their corresponding ranks in adaptive commits of these systems. We see that the intersection of the two lists is *fix*, *add*, and *bug*. Additionally, the top three terms of adaptive commits (*port, support,* and *replace*) are not in the common terms for the non-adaptive commits.

There is a significant difference between term frequencies in adaptive commits and non-adaptive commits. In particular the keywords *port*, *support*, and *replace* appear to be excellent candidates to characterize adaptive change commits. Frequent use of the words *port* and *support* are key indicators that some type of adaptation or migration is taking place, porting to or supporting the new version. The higher ranking of terms *replace* and *remove* reflect that many changes involved in adaptive maintenance tasks

involve removing and replacing old methods/class types (e.g., Qt3) with new versions (e.g., Qt4).

Next we examine frequent term combinations and consider sets consisting of two terms. Additionally, we created sets consisting of the terms (*port*, *add*, *remove*, and *replace*) and (*support*, *add*, *remove*, and *replace*). For the term combinations, we applied the same approach as applied for single term sets. The results are given in Table 4-11.

The results are not surprising; the term-sets that contain the keywords *port*, *support*, *replace*, or *remove* occur in 1% or less of non-adaptive commits, as compared to 6% or more in adaptive commits. Furthermore, the adaptive top frequency set (*port*, *replace*) doesn't occur in any of the non-adaptive changes of the systems.

The lesson learned from this section is that by using specific identifiable vocabularies in the commits' log messages, adaptive change commits can be automatically identified using Information Retrieval (IR) techniques. More investigation is necessary to find the top most frequent terms-sets for each file-size category, in order to use the commits' log messages keywords to understand what types of changes most commonly happen for each size category.

4.3 Adaptive Commit Developers

We now examine the developers who actually committed the changes. This information can be taken directly from the commits in subversion, as the developer ids saved with each commit. We are interested to see how many and what types of developers were involved in these changes.

Table 4-9. Top 12 average single term frequency in the adaptive commits and their frequency in non-adaptive commits.

Term	Average Rank	
	Adaptive change commits	**Non-adaptive change commits**
port	45.10%	3.05%
support	36.30%	2.45%
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%
cleanup	3.20%	1.60%
update	1.80%	8.60%
patch	0.85%	1.20%

Table 4-10. Top 12 average single term frequency in the non-adaptive commits and their frequency in adaptive commits.

Term	Average Rank	
	Adaptive change commits	**Non-adaptive change commits**
new	0.47%	22.87%
fix	18.70%	22.25%
warnings	0.30%	20.07%
add	14.60%	19.45%
crash	0.00%	14.80%
show	0.00%	13.10%
avoid	0.00%	12.45%
use	0.00%	12.01%
change	0.20%	10.25%
bug	8.90%	10.10%
get	0.00%	9.93%
load	0.00%	8.71%

Table 4-11. Top 12 average term set frequency in the adaptive commits and their frequency in non-adaptive commit s.

Term Set	Average Rank	
	Adaptive change commits	**Non-adaptive change commits**
port, replace	11.50%	0.00%
support, replace	7.10%	0.10%
port, remove	6.45%	0.08%
add, replace	4.95%	1.05%
port, add	4.35%	0.20%
remove, replace	3.90%	0.35%
port, test	3.05%	0.02%
support , add	2.80%	0.90%
add, fix	2.70%	2.65%
port, bug	2.00%	0.01%
support, compile	1.50%	0.01%
port, add, remove, replace	1.45%	0.00%

That is, are these developers who have made substantial contributions to the project or are they very specialized and only involved in this part of the development process.

Accordingly, we developed a tool to analyze the adaptive change commit contributions of the KOffice, Extragear/graphics, and OSG developers, using author information in the subversion log entries. We then determined the total number of adaptive commits performed by each developer.

The total number of KOffice developers who made commits during the studied time period is 238. Only 26 of these developers (10.29%) are involved in adaptive change (i.e., author of an adaptive commit). The outcome is similar for Extragear/graphics, where only 17 out of 175 developers (9.72%) were involved in adaptive changes and for OSG where only 8 out of 102 developers (7.84%) were involved in adaptive maintenance activities.

However, it is relatively well-known that most of the commits made to an open source system are by a very small fraction of total developers [Kagdi et al. 2008] across all maintenance activities. The size measure results shows that adaptive changes were often system wide. Moreover, adaptive changes are typically complex and propagated over the source code [Malik and Hassan 2008].

Therefore, performing adaptive changes requires a developer who has broad knowledge of the system. To see if this phenomenon holds for the systems we examined, the expertise of the developers involved in these adaptive commits is investigated.

Table 4-12, Table 4-13, and Table 4-14 present, for each of the three systems, the top most active developers who were involved in the adaptive maintenance task. These are developers who made two or more adaptive commits during the time frame. There are only 7 developers for KOffice in this category, 4 for Extragear/graphics and 3 for OSG. In general there are a very small number of developers making substantial contributions for these adaptive maintenance tasks.

We also see that many of these top adaptive commit authors are also contributing large amounts of effort to other aspects of the system maintenance. For example, the top 7 adaptive developers contributed approximately 36% of non-adaptive commits for KOffice, while the other 231 developers contributed the remaining 65%. That is, these 7 contribute a great deal to the project and can be considered expert developers.

As mentioned previously, these results are comparable to the results of developer contribution found in other studies [Kagdi et al. 2008] (i.e., a small number of expert developers contribute the majority of the code). We now examine if these developers use a more specific vocabulary within their commit log messages pertaining to the adaptive maintenance task.

4.4 Developer and Vocabulary Correlation

We will now investigate the correlation between the developers and the word distributions in the commit annotations. It has been found that some developers repeat same words in their commits' log messages or they have a message style rather than a project-wide lexicon or idiom [Hindle et al. 2009].

We focus on the most active developer (i.e., those names given in Table 4-12, Table 4-13, and Table 4-14) and the top most ranked terms (i.e., the first 9 terms in Table 4-9). For each of these developers, we extracted the term frequency of each of the top terms in the messages of the adaptive commits that they made.

Table 4-12. Frequency distribution of KOffice developers contribution who made at least two adaptive commits.

Author	Frequency			
	Adaptive Commits		Non-adaptive Commits	
	Number	Ratio	Number	Ratio
ingwa	25	19.08%	816	2.10 %
mpfeiffer	23	17.55 %	271	0.70 %
zander	16	12.21 %	4180	10.76 %
staniek	15	11.45 %	2156	5.55 %
dfaure	10	7.63 %	695	1.79 %
rempt	10	7.63 %	3554	9.14 %
nikolaus	9	6.87 %	2035	5.24 %
Total	**108**	**82.42%**	**13707**	**35.28 %**

Table 4-13. Frequency distribution of Extragear/graphics developers contribution who made at least two adaptive commits.

Author	Frequency			
	Adaptive Commits		Non-adaptive Commits	
	Number	Ratio	Number	Ratio
cgilles	160	73.06 %	5769	22.09 %
mwiesweg	19	8.68 %	2146	8.21 %
aclemens	17	7.76 %	3040	11.64 %
nlecureuil	4	1.83 %	101	0.39 %
Total	**200**	**91.33 %**	**11056**	**42.33%**

Table 4-14. Frequency distribution of OSG developers contribution who made at least two adaptive commits.

Author	Frequency			
	Adaptive Commits		Non-adaptive Commits	
	Number	Ratio	Number	Ratio
robert	28	35.44 %	984	23.26 %
cedricpinson	23	29.11 %	284	6.71 %
paulmartz	20	25.31 %	374	8.84 %
Total	**71**	**89.86 %**	**1642**	**38.81 %**

Figure 4-3 summarizes the results obtained for the KOffice system. These results are helpful when determining the types of adaptive actions performed by a developer [Kagdi et al. 2008]. For example, from Figure 4-3, the most frequent activity of developer mpfeifer is removing the old Qt3 functions/classes and then compiling the updated versions of the modified files. That is, this developer has a great deal of experience with the subtask of deleting and updating old Qt3 features. This data could enhance a developer recommendation system based on maintenance action (e.g., test, replace) similar to the one proposed in [Kagdi et al. 2008] that is based on the file/class being changed.

However, our main object here is to try and automatically identify adaptive commits. As such we examine what terms each developer uses most frequently for adaptive commits. That is, who is the best developer to be with a given term in one set? The goal is to identify any trends in term usage.

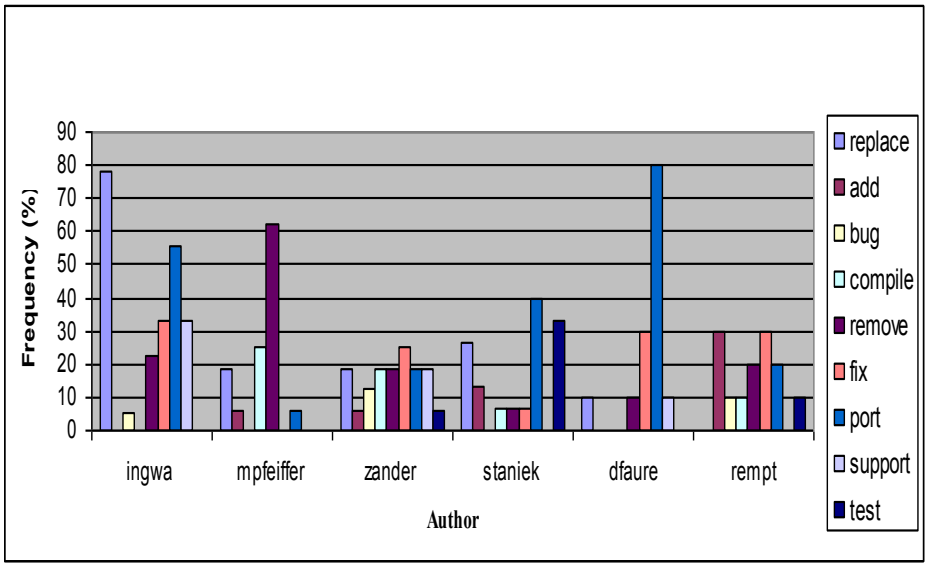


Figure 4-3. Term distribution for each active developer of KOffice system.

To accomplish this, for each term, we place the adaptive commits that contains the term and have the same developer in one group, which is assigned a name similar to the name of its commits developer. Then, the frequency of each group is calculated as the numbers of adaptive commits in that group over the total number of adaptive commits contain the term. For any term, the group with higher frequency represents the developer who used this term rather than other developers, and so a set involves this developer and the term is created.

Figure 4-4 presents the distribution of most common terms (from Table 4-9) for each of the top developers (from Table 4-12). We see that developers tend to use the same terms regularly. For example, the developer *ingwa* uses the terms *replace* and *support* quite often and far more than others.

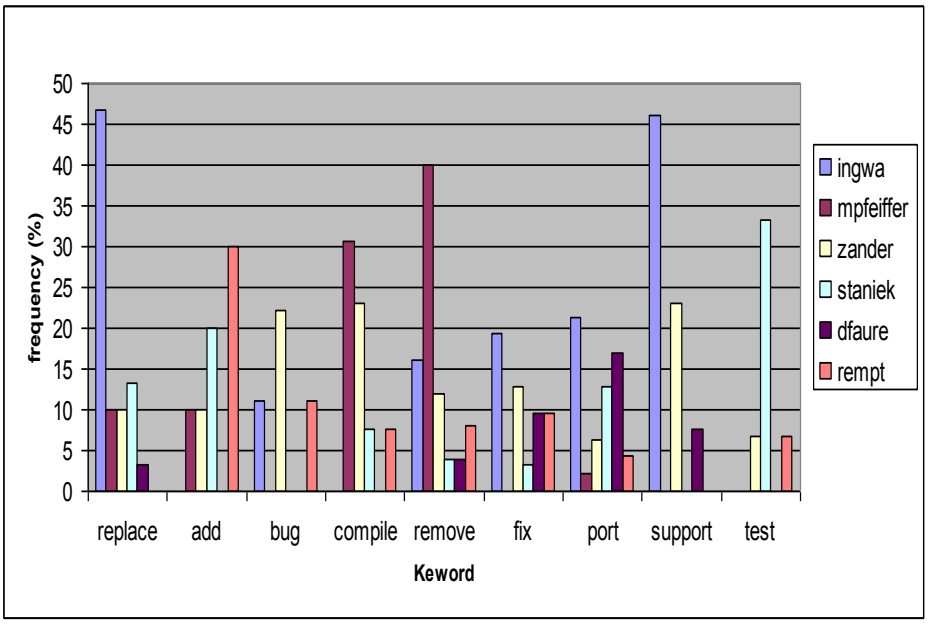


Figure 4-4. Term usage distribution of the top most active adaptive maintenance committers for KOffice.

We now examine the frequency for each of the created developer-term pairs in both adaptive and non-adaptive commits for the studied systems in the considered period of time.

As shown in Table 4-15, Table 4-16, and Table 4-17, there is a significant difference between frequencies of these sets in adaptive commits and non-adaptive commits. Additionally, some single terms occur frequently in non-adaptive changes, but when combined with developers, this frequency is decreased considerably. For instance, the frequency of the term *fix* is 22.25% for non-adaptive commits, but the frequency of (*fix*, *ingwa*) is 0.14% for this type of commit.

Here, we see that the combination of developer and term tends to produce very good results in discriminating adaptive and non-adaptive commits. Therefore, using developer and term combination is possibly a better approach for automated adaptive commits classification compared with the use of them independently.

4.5 Threats to Validity

Several threats to validity may influence the results of our work and the capability to generalize the obtained results. In other words, the initial research and findings seem promising, but a number of important questions still need to be addressed.

The presented case study is based on investigating the version history. The packages that we examined are prime examples of successful open-source development that involve a number of APIs modifications. Moreover, the test systems for the preliminary experiments were chosen because of their availability and the fact that other researchers ran experiments using these systems. However, we do not claim that the

obtained results would generalize to a broad range of systems, such as the closed-source systems.

Table 4-15. Frequency distribution of developer-term sets over KOffice system commits.

Developer- Term Set	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
replace, ingwa	10.69 %	0.11 %
remove, mpfeiffer	7.63 %	0.16 %
port, ingwa	7.63 %	0.14 %
fix, ingwa	4.58 %	0.64 %
support, ingwa	4.58 %	0.13 %
test, staniek	3.82 %	0.1 %
compile,mpfeiffer	3.05 %	0.07 %
add, rempt	2.29 %	1.31 %
bug, zander	1.52%	0.98 %

Table 4-16. Frequency distribution of developer-term sets over Extragear/graphics system commits.

Developer- Term Set	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
port, cgilles	46.58 %	0.09 %
add, cgilles	9.13 %	2.89 %
replace, cgilles	6.39 %	0.02 %
bug, cgilles	5.02 %	4.20 %
remove, mwiesweg	4.57 %	0.90 %
fix, cgilles	3.20 %	7.96 %
compile, cgilles	2.28 %	2.39 %
test, cgilles	1.83 %	0.45 %
support, aclemens	1.37%	0.05 %

Table 4-17. Frequency distribution of developer-term sets over OSG system commits.

Developer- Term Set	Frequency	
	Adaptive change commits	Non-adaptive change commits
support , robert	13.77 %	0.11 %
replace, robert	8.43 %	3.12 %
add, cedricpinson	5.82 %	1.22 %
remove, robert	4.98 %	0.80%
port, paulmartz	4.47 %	0.00 %
fix, cedricpinson	4.18 %	4.96 %
test, cedricpinson	3.08 %	5.06 %
compile, robert	1.65 %	8.62 %
cleanup, paulmartz	1.23%	5.54 %

The terms that a committer writes in the log messages can be subjective, which means that the usage of a particular term in a project could be unintentional. We decreased the effect of this threat by studying two different packages of KDE and the separate project OSG. We used the committer id that represents the developer's identity.

However, from the repository data we do not know precisely which developer adaptively changed the files, only the developer that committed the changes.

Essentially, developers do not follow a standard pattern of when to commit. Some commit every small successful change, where others wait until the completion of the whole task. Developers may use one commit or several contiguous commits. Thus, some adaptive maintenance tasks are accomplished via several commits and some commits contain both adaptive and other types of maintenance actions.

Our study examines a particular type of adaptive change (i.e., API migration) and we cannot generalize our results for other types of adaptive changes (e.g., changes to underlying operating systems). This is a motivating concern of future investigation.

4.6 Related Work

A number of studies based on software maintenance categorization have been suggested. In terms of the frequency of maintenance commits, Lientz [Lientz et al. 1978] published a survey of software maintenance, where they found that 18.2% of the accomplished maintenance was categorized as adaptive. Similarly, Schach [Schach et al. 2003] performed an empirical study of maintenance activities and found the distributions of adaptive maintenance category were as 13.8% for a commercial real-time product (RTP), 0.9% for the Linux kernel, and 6.3% for GCC.

Several researches have proposed studying and extracting information from the commits of software version histories. Robles et al.[Robles et al. 2006] investigated version histories to study the evolution of a software distribution in terms of the number of packages, lines of code, use of programming languages, and sizes of packages/files.

The characterization of a typical version history commit is investigated with respect to the number of files, number of lines, and number of hunks committed together in[Alali et al. 2008]. In this study, the most common terms in the commits' log messages are extracted and linked with the size categories of the commits. The results showed commonality of terms and the frequency of their usage even over different projects. This shows the idea of using a set of predefined terms for use in commit messages.

In quantitative analysis, numerous researchers are particularly interested in keyword frequency analysis and their use in classification of commits' log messages for the purpose of identification of maintenance changes. In [Mockus and Votta 2000], Mockus and Votta proposed the usage of textual descriptions (message terms) to identify the reason for the maintenance activity. Additionally, they found several results on using version control data in detecting the relationships between the type and size of the software changes. One of their results is using the difficulty, size, and times intervals to identify the type of changes.

An information-retrieval method is used in [Canfora and Cerulo 2005] to index the changed files in software repositories by exploiting the textual description of past bug reports in the Bugzilla repository and the CVS commits' log messages.

Hindle [Hindle et al. 2009] proposed an automated classifier for large commits by training machine learners on features extracted from the commit metadata. Their results demonstrate that the terms in the commits' log messages and the identity of authors provide helpful information to allow large commits to be classified automatically.

In [Kagdi et al. 2008], an approach is presented to recommend a ranked list of developers to assist in performing software changes to a particular file. This approach is derived from the analysis of the previous commits involving the considered files, with a recommendation based on change expertise, experience, and contributions of developers.

Minto and Murphy [Minto and Murphy 2007] developed a tool called the Emergent Expertise Locator (EEL). This tool mines the history to assist in finding that files were changed together and who contributed to the change. Zimmerman et al. [Zimmermann et al. 2004] developed a tool for version archives that guides programmers to related changes for the purpose of predicting future changes and discovering file coupling. An approach to discover traceability links between software artifacts via the examination of software version histories is presented in [Kagdi et al. 2007], where sequential pattern mining was used to uncover the ordering of committed files over a sequence of change-sets, and not just in a single change set.

Several works dealing with the API migrations have been presented. A manual analysis to determine what the correct adaptations are to migrate between several versions of a set of Java-based software libraries is presented in [Cossette and Walker 2012]. Also, the authors developed a set of adaptation recommender techniques to identify the correct adaptations for the library migration. A wrapper-based API migration is proposed in [Bartolomei et al. 2010]. In this approach, the original API is replaced by a wrapper-based re-implementation that reuses the other API. This approach can extract the design patterns from the solutions that developers used in the GUI wrappers. In [Wei Wu et al. 2010] , the authors introduce a novel hybrid approach that is called AURA.

This approach combines call dependency and text similarity analyses to identify change rules for one-replaced-by-many and many-replaced-by-one methods, which are needed during the APIs migration process.

To the best of our knowledge, this is the first work that not only uncovers but also further analyzes adaptive change commits, with comparison to non-adaptive maintenance tasks. There is no previous work in the literature that identifies the main characteristics of adaptive commits in large-scale systems, including the identifiable vocabulary and the commit size.

4.7 Summary

The results of a case study to identify and analyze the adaptive changes occurring due to a migration to a new API were presented. The study is based on data obtained by manually examining the version history to distinguish adaptive from non-adaptive commits of the KOffice, Extragear/graphics, and OSG open source projects. Both KOffice and Extragear/graphics underwent an adaptive maintenance task that involved migration from Qt3 to Qt4. OSG underwent a migration of OpenGL to a new version.

The study uncovered four main results. First, the commits involved in an adaptive maintenance task are typically system wide and large. That is, on average the adaptive maintenance tasks touch more files than non-adaptive tasks. For instance, with respect to a file-size measure the obtained result shows that 21% of adaptive commits, compared with 9% of non-adaptive commits, are classified as large commits for KOffice.

Second, the commit messages of adaptive changes typically use a small set of terms that are indicative to adaptive changes. That is, some terms, such as *port*, occur

frequently in adaptive commits (i.e., 45%), but infrequently in non-adaptive commits (i.e., 3%) for the systems we studied. Similarly, some combinations of two terms, such as (*port*, *replace*), occur with a higher frequency for adaptive commits compared with non-adaptive commits. The frequency of (*port, replace*) is 11.5% for adaptive commits, while it never appeared in non-adaptive commits in the studied systems. This demonstrates that selected sets of terms could be employed in the automated detection of adaptive commits.

Third, the developers who carry out the adaptive maintenance tasks are few in number and appear to have a broad knowledge of the entire system. The developer contribution results show that only 4 developers performed 92% of adaptive changes for Extragear/graphics system. Moreover, those same developers also were involved in 42% of the non-adaptive commits.

Finally, we observed that the combination of commit message term and developer was predictive of adaptive change. That is, the developer-term (*replace*, *ingwa*) occurs frequently in adaptive changes (10.69%), but infrequently in non-adaptive changes (0.11%). Thus, the combination between the developers and the terms tends to give very good results in discriminating adaptive and non-adaptive commits.

CHAPTER 5

Examining Adaptive Changes on Software Artifacts

Software system is a composite of several artifacts such as source code, test cases, design diagrams, and other documentations. Each one of these artifacts is responsible for a set of parts of the software system [Reiss 2006] . Software evolution is the process where the changes were undertaken at the different artifacts of the system at different rates. For instance, test cases must be updated to include the addition of new features.

As discuss in CHAPTER 3, each version history commit provides the complete coverage of the documents, from different artifacts, that were changed by this commit. Therefore, we take an examination of the adaptive commits into a view that gives an understanding of how adaptive maintenance tasks involve changes across various artifacts of the system.

In this chapter, we investigate the software artifacts that were changed by adaptive commits to address three related questions: what kind of artifacts that were modified by adaptive maintenance? Is there a relation between modified artifacts and maintenance type? How modifications at software artifacts differ between adaptive and non-adaptive maintenance?

5.1 Software Artifact Types

Open source software is a multidimensional system consists of a set of artifacts. These artifacts help explain the function, architecture, and design of software system.

Additionally, some artifacts concerned with the process of software evolution, such as the progress documents. Software artifacts ranging from those at a high level of abstraction, such as requirement, to architecture, design, and source code-level artifacts, to documentation and test suites [Buckley et al. 2005]. The non- source artifacts are valuable as the source code itself. Thus, software development and evolution productivity depend on the ability to develop, relate, and consistently maintain all these different artifacts of the software [Reiss 2002].

Here, in addition to the source code files, non-source artifacts, as reported in [Scacchi 2002] , can be classified as follow:

- User documents, such as *HTML*, *XML/docbook*, *LaTeX* and *Doxygen* files.
- Build management documents. This includes *automake*, *cmakelists*, and *makefile* files.
- How To guides, such as *FAQs*.
- Release and distribution documents. This consists of *ChangeLogs*, *whatsNew*, *README*, and *INSTALL* guides files.
- Progress monitoring documents, such as *TODO* and *STATUS*.
- Extensible mechanisms. An example of this type is: *Python*, *Ruby*, and *Pearl* bindings for an API.

The preferred way of building and evolving systems would be have a formal definition for every artifact, and then guarantee the consistency of their undertaken changes during software maintenance [Reiss 2006] .

Here, we interest in studying the effect of adaptive maintenance on modifying each of the mentioned artifact through examining the manually captured adaptive commits of the three studied systems, namely: KOffice, Extragear/graphics, and the OSG systems.

5.2 Features Extraction from Changed Artifacts

It is evident from CHAPTER 3 that it is fairly straightforward to determine the files and their types that are involved in a commit using the dimension: *path*. Additionally, each commit stores the corresponding files and their change action (modified, deleted, and added). On the other hand, multiple developers may change the same file in different commits. This file-level knowledge is cheap to compute, as only the log-entries need to be examined and does not need any processing of the files for their content changes [Alali et al. 2008]. Explicitly, Knowing and understanding what artifacts of files being changed in a given commit is very valuable to developers, testers, and managers.

In this context, feature often means some property and characteristic of the change, such as the frequency of words and the kind of modified files that are present in repository commits. By mining change histories, we can derive features examining what software artifacts are changing in a given commit. The number of file types could be used as the macro-level size measure of a commit. For instance, this file-based measure would indicate that the commit with more different artifacts and a larger number of files is ‘larger’. Such a measure may be useful in decisively categorizing different commits into different classes [Purushothaman and Perry 2005]. Moreover, the size of files being

changed can reflect the type of undergoing changes. For example, Mockus and Votta [Mockus and Votta 2000] show that changes to smallest files are likely adaptive changes, while modifications in larger mature files classified as corrective tasks.

The directory/module and file name counted as features, since they predetermine both module information and some behavioral semantics of the system. For example, changing the file 'eventslot.ui' (from the KOffice project) in the directory '/trunk/koffice/libs/kross/api' can be considered as a modification dealt with change to system APIs, which represents an adaptive maintenance task. Additionally, commits are composed of files that belong to several modules. Therefore, the number of modules is a valuable feature that assists recognizing the commit type. For instance, adaptive maintenance spans across a large number of modules.

Many researchers achieve bug predictions at the file granularity [Ostrand et al. 2004], where particular files, of different artifacts, in the system are probably contain faults, and so changes to these files seem as corrective maintenance. Further, a frequent-pattern mining technique can be applied to identify the sets of files, of different types, that frequently co-occur in several commits. We refer to such a set of files as a traceability pattern. This extracted feature was taken for change prediction at the file level. As a result, this prediction helps in identifying the maintenance type associated with predicted commits.

The frequency and extent of changes to files committed by developers can be yet another feature of file metadata. A developer who commits changes to files from different artifacts has (or acquires) of knowledge of these files, or a developer commit spanning

across a large number of files may indicate that the developer has a wider knowledge of the system [Kagdi et al. 2008].

Classifying commits using features extracted from changed files, of different artifacts, is attractive because it may not require retrieving and then analyzing the source code of the touched files. However, few efforts have been made to infer and then utilize commit file metadata to uncover specific category of repository commits.

Here, we examine the changed files by investigating the *path* metadata of each commit. The goal is to understand the importance of using the knowledge about modified artifacts for possible automatic identification of adaptive commits.

5.3 Commit Categorization Using Changed Artifacts

Maintenance problems happen when a modification on one artifact is not consistently mapped and applied to another related artifacts of a software system [Ivkovic and Kontogiannis 2004] . Thus, developers must correctly make decisions regarding which artifacts might be affected by each proposed change. Otherwise, different artifacts will be evolved independently and inconsistently [Reiss 2002]. That is, some maintenance tasks involved changes occurred at different levels of software artifacts. To trace such changes, we can examine those commits that involved a set of files from different types of software artifacts.

Complexity of a maintenance change can be characterized in a way that reflects the number of software artifacts being modified [Greevy et al. 2006]. Here, we analyze the undertaken commits by examining the affect of such commits in modifying different

software artifacts files. This analysis will assist the comparison between adaptive and non-adaptive commits.

To aid such comparison, a set of factors is used to classify the commits, in the context of the version history of the studied system. Firstly, we categorize the commits using the kind of usage of touched files, where the following three categories were defined:

- Commits affect only *source* code files.
- Commits affect only *non-source* files.
- Commits affect *both* source and non-source files.

This classification is used to determine the overall impact a given adaptive commit has to a system. Moreover, the objective is to see if there is even a comparison between adaptive and non-adaptive commits in the context of modified files.

Table 5-1, Table 5-2, and Table 5-3 provide a summary of the obtained results from commit categorization using above mentioned classes for the three examined systems. The results show that the majority of the both the adaptive and non-adaptive commits affected only source code files. However, all adaptive commits modify source files, since they represent an API migration, and never touch other artifacts separately. In contrast, some non-adaptive commits (e.g., 13.8% for KOffice system) changed other artifacts without involving changes to the source code files. Given this trend, the appearance of non-source code files only can be an indication of the commit type. That is, such appearance can help in automatically identifying a commit as a non-adaptive commit.

Table 5-1 . Commit categorization based on type of modified files for KOffice system.

Category	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
affect only source code files	61.1%	70.5%
affect only non-source files	0.0%	13.8%
affect both source and non-source files	38.9%	15.7%

Table 5-2. Commit categorization based on type of modified files for Extargear/graphics system.

Category	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
affect only source code files	73.5%	61.8%
affect only non-source files	0.0%	27.1%
affect both source and non-source files	26.5%	11.1%

Now let us look at the extension of those non-source code files that were modified and affected by the adaptive commits of the three examined systems. Table 5-4 presents a picture of the extensions of the non-source code files that were modified by the adaptive commits. We see that even over different projects there is commonality of the usage and

roles of such modified files occurred in adaptive commits. This gives some credence to the idea of using role of files for teasing out adaptive and non-adaptive commits.

In an attempt to get a zoomed-in picture, we further process the commits that fit into the third category (commits affect both source and non-source files). Here, we count how many non-source code file extensions, see Table 5-4, that occurred for each of those commit, and then find the corresponding distribution for both adaptive and non-adaptive commits.

Table 5-3. Commit categorization based on type of modified files for OSG system.

Category	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
affect only source code files	58.3%	47.7%
affect only non-source files	0.0%	25.6%
affect both source and non-source files	41.7%	26.7%

Let us consider the histograms given in Figure 5-1, Figure 5-2, and Figure 5-3 for the three-studied system. Our first observation is the right skew to the distribution. That is most of the commits, from both types, involved changes to one or two non-source code file extensions only along with the source code files.

However, adaptive commits are most likely modifying more file extensions comparing with non-adaptive commits. This result is comparable with the obtained

results in CHAPTER 4, where the adaptive maintenance tasks on average touch more files than non-adaptive tasks.

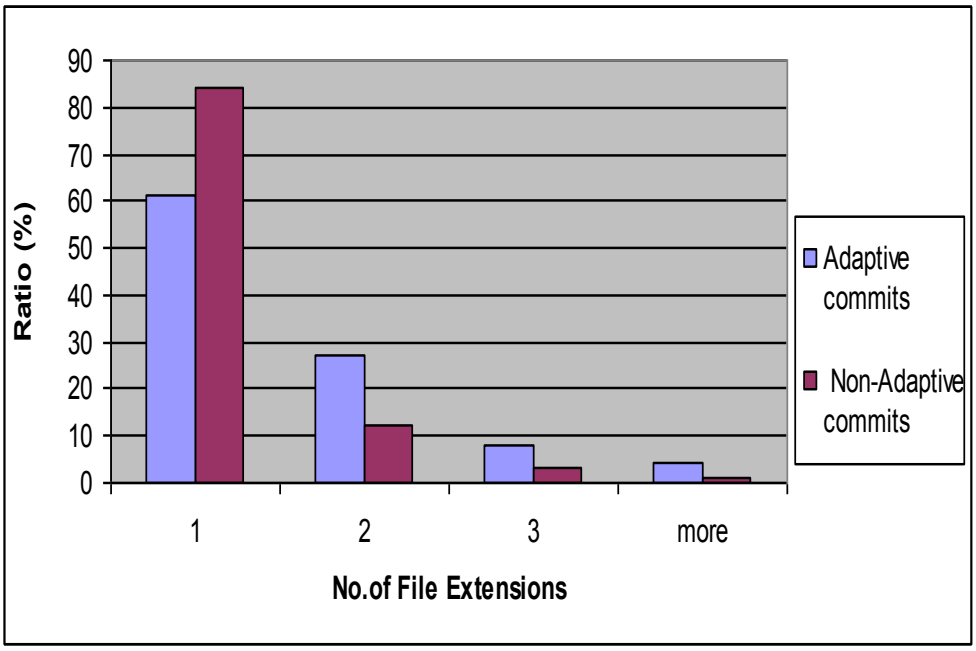


Figure 5-1. Histogram of the number of different non-source code file extensions with regards to the number of commits in the KOffice system

5.4 Changed Files Categorization

Next, we further examine the idea of the comparison between adaptive and non-adaptive commits but now with respect to a fair question to ask concerning modified files, are there files (source and non-source) changed *exclusively* by the adaptive commits?

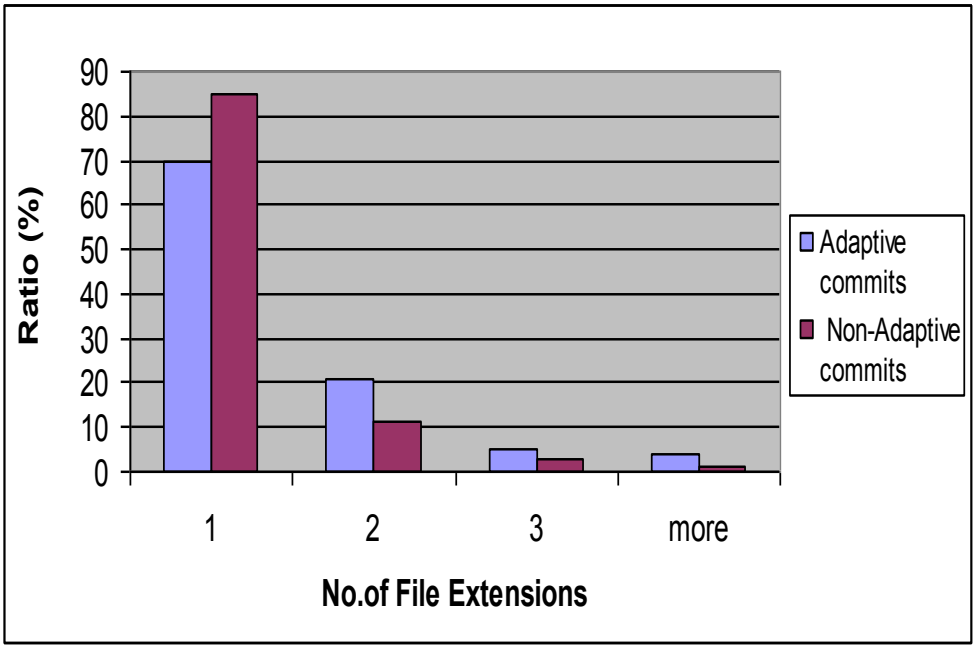


Figure 5-2. Histograms of the number of different non-source code file extensions with regards to the number of commits in the Extragear/graphics system

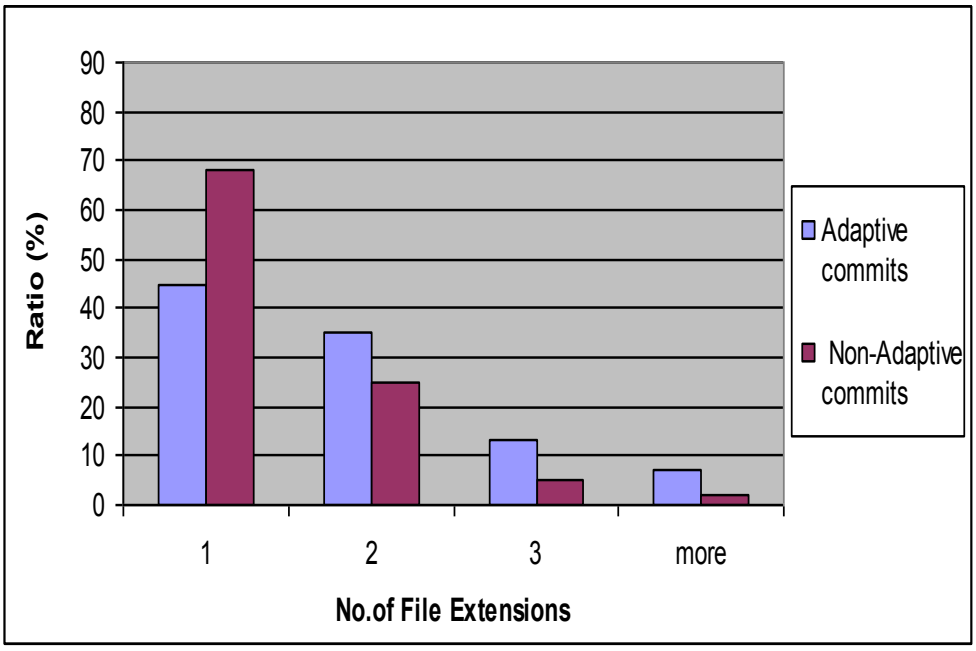


Figure 5-3. Histogram of the number of different non-source code file extensions with regards to the number of commits in the OSG system.

Table 5-4. File extensions that were modified by adaptive commits for the three studied systems, (Y) means modified otherwise no modification were occurred.

Extension	Description	Affected By adaptive Commits		
		KOffice	Extragear/graphics	OSG
.qs	Stores a library of styles that can be incorporated into a user interface.	Y	Y	-
.txt	Standard text document that contains unformatted text	Y	Y	Y
.ui	Stores the user interface configuration for a program; saved in an XML format	Y	Y	Y
.rc	Includes references to resources, such as user interface components	Y	-	Y
.py	Program file or script written in Python	Y	Y	Y
.sh	Script programmed for bash contains instructions written in the Bash language	Y	Y	Y
.desktop	Data file that provides information about an item in a program's menu	Y	Y	-
.xpm	Stores pixmaps as static character arrays in the C programming language	Y	Y	-
.odt	XML-based file format for spreadsheets, charts, present ations and word processing documents	Y	Y	-
.in	Generic file used to track activity on a computer	Y	-	Y
.rb	Software program written in Ruby, an object-oriented scripting language	Y	-	-
.am	Contains a Makefile template	Y	Y	Y
.bot	Contains settings for an installation	Y	Y	Y
.icm	Contains a color settings for a specific software program or device	-	Y	-

To answer this question, we classify all modified files based on the type of affected commit. We categorize changed files of the three examined systems into the following categorizations:

- Only changed by *adaptive* commits.
- Only changed by *non-adaptive* commits.
- Changed by *both* types (adaptive and non-adaptive commits).

Our study shows that more of source files were touched only by the adaptive commits comparing with those files changed by both adaptive and non-adaptive commits, as shown in Figure 5-4. Since no solid comparisons can be drawn in the coarse view of changed source code files, it is apparent that examining the modification of other artifacts is vital. Hence, we examined the non-source documentations that were influenced by the adaptive tasks. Whether such maintenance affects the system documentation in minor or major way, a portion of the documentation was modified by both types of commits, as shown in Figure 5-5. Consequently, we view this as a very important and positive result.

The lesson from these obtained results is that changing some files is associated with a specific type of maintenance (e.g., adaptive maintenance). For instance, from an inspection of those files that were only modified by the adaptive maintenance, we find that they represent an interface for calling the operating platform (e.g., Qt4) APIs, and so the migration to new APIs request changing such files. That is, the modified system documentations and source files can be leveraged to assist automatically identify adaptive changes within a version history.

It is evident from the previous results that adaptive maintenance involves changes to non-source code files. We now examine the type of these modified files in more details. Here, we address the questions: What kinds of software artifacts that were modified by the adaptive maintenance? Are there artifacts that *never* touched by adaptive commits?

Delving a bit deeper, we calculated the distribution of modified documentations over the commits. The distribution of our data is divided into six regions based on the non-source artifact types given before in section 5.1.

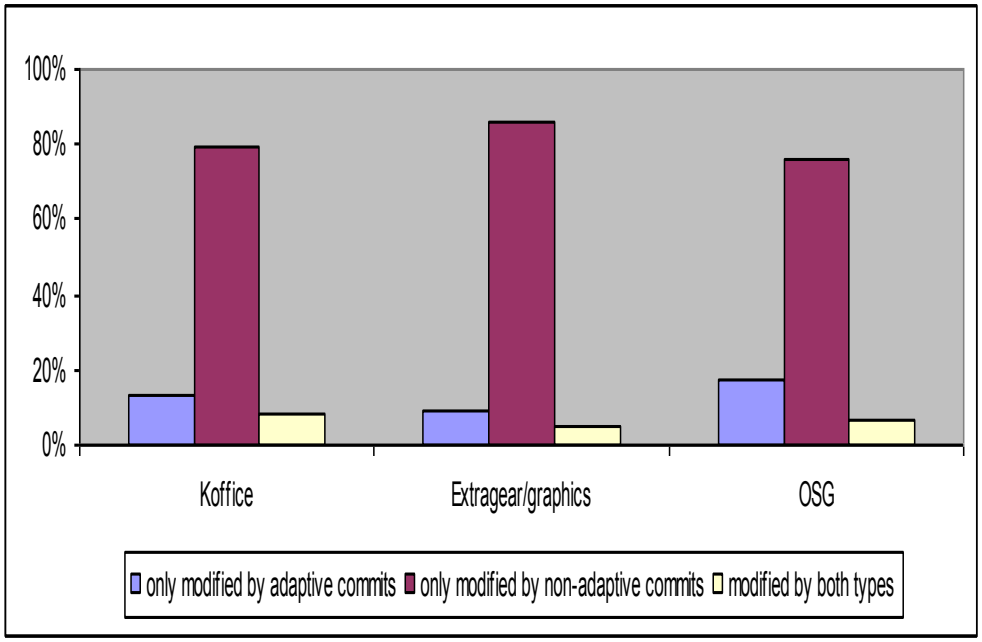


Figure 5-4. Histogram of the number of *source code* files with regards to the type of affected commit in the three examined systems

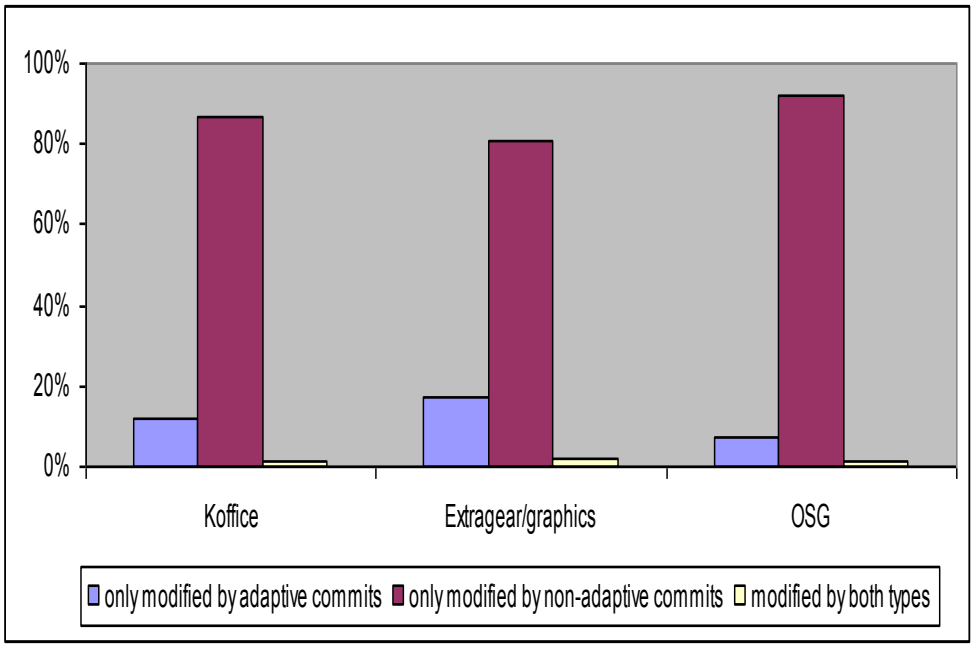


Figure 5-5. Histogram of the number of *non-source* code files with regards to the type of affected commit in the three examined systems

As shown in Figure 5-6, adaptive maintenance affected only two kinds of artifacts, namely, build system managements and extensible mechanisms (scripts for an API), while the non-adaptive maintenance affected all non-source code artifacts, as shown in Figure 5-7. For instance, the files with extension '*.qs*', which stores a library of styles that can be incorporated into a user interface, were exclusively modified by adaptive commits. This is not surprising given that adaptive maintenance deals with changes in processing environments and APIs.

Also, from the distribution results, we see a number of interesting trends. In particular, we see that even over different projects there is commonality of modified documentation kinds and the frequency of their usage. This gives some credence to the

idea of the correlation between the maintenance type and the kinds of artifacts being modified.

One of the most important steps of developing a software system is the compilation process where source code files are converted into executable code files. Such process is usually managed by the build management files [Scacchi 2002], which use a compiler independent method. It was decided that *CMake* would be the build management tool that are used to base all of the development with the KDE project[9]. Similarly, this build tool is also used for managing the building process of the OSG project[10]. *CMake* reads script files (*CMakeList.txt*) and produces input files for the native build system (*Makefile* documents) of the platform where it runs on[11].

As shown in Figure 5-6, adaptive maintenance affected build management files rather than other types of non-source code artifacts. For that reason, we look at the adaptive modification of build management files in a bit more detail and address the following questions: which of the build management files was changed frequently by the adaptive commits? How can adaptive maintenance be compared with non-adaptive maintenance regarding the modification of the build management files?

---

[9] See http://www.KDE.org

[10] See www.openscenegraph.org

[11] See http://techbase.kde.org/Development/Tutorials/CMake

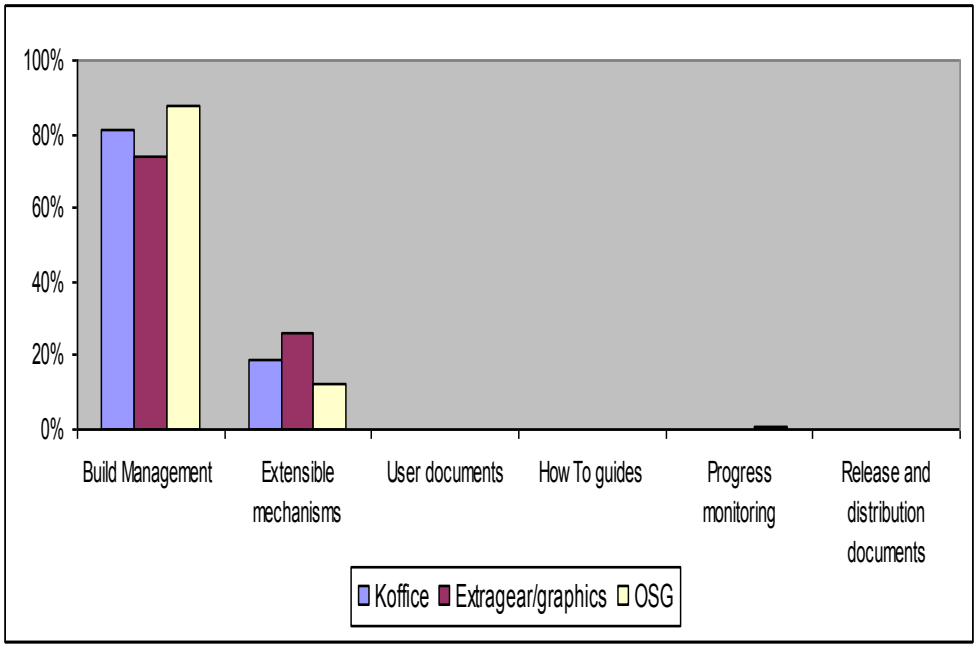


Figure 5-6. Histogram of the non-source code artifacts with regards to the number of *adaptive* commits in the three examined system

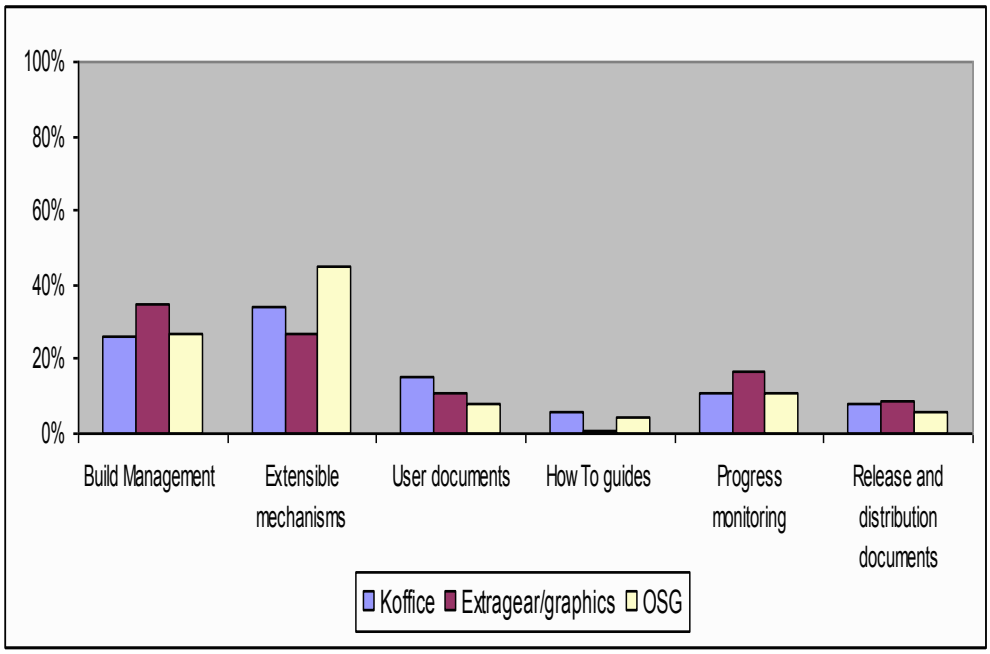


Figure 5-7. Histogram of the non-source code artifacts with regards to the number of *non-adaptive* commits in the three examined systems

To address our goal of understanding how adaptive changes affected the build management files, we examine the frequency of change for both build documents (*CMakeLists* and *Makefile*) that occurred by adaptive and non-adaptive commits for each of the studied systems. In other words, we divide all changes that occurred in build management files into two groups: changes to *CMakeLists* documents, and changes to *Makefile* documents, and find the frequency of each group relative to the total number of build management changes. The results are shown in Figure 5-8 and Figure 5-9.

The obtained results are hardly surprising. Adaptive commits are from different systems but have the same effect of changes regarding build management files. That is, in adaptive maintenance process, more than 98% of the build management changes were occurred in the *CMakeLists* documents. This is mainly due to the fact that such documents represent the input text files that contain the project parameters and describe the flow control of the build process in simple *CMake* language, where these parameters are affected by the migration from old platform to the new one (e.g., migration from Qt3 to Qt4). For instance, *CMakeList.txt* documents should be update to correctly locate Qt4 libraries, which differ from those related to Qt3[12]. In contrast, a lot of changes occurred at *Makefile* documents (e.g., 48% for the Extragear/graphic system) by the non-adaptive maintenance tasks.

As a result, the occurring of *makefile* changes in a given commit is an indication that this commit is most likely a non-adaptive commit. That is, using the undertaken

---

[12] See http://techbase.kde.org/Development/Tutorials/CMake

changes in the build management files can be helpful to recognize adaptive and non-adaptive commits for any source system.

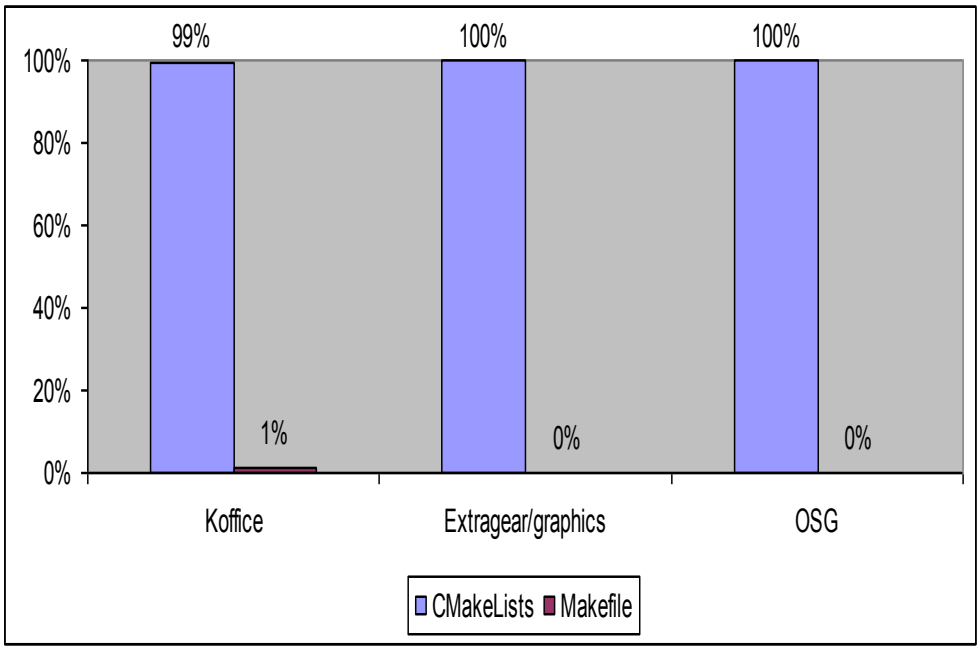


Figure 5-8. Frequencies of the two build management documents. Percentages are compared to total modified build management files for the *adaptive* commits in the three examined system.

Therefore, from the results obtained in this section, knowing and understanding the changed files in a given commit could effectively be a very valuable to automatically teasing out the type of an undertaken commit (Adaptive or non-adaptive).

5.5 Threats to Validity

The assessment of our investigation, the validity of our work results and the ability to generalize obtained results are subject to a number of threats to validity.

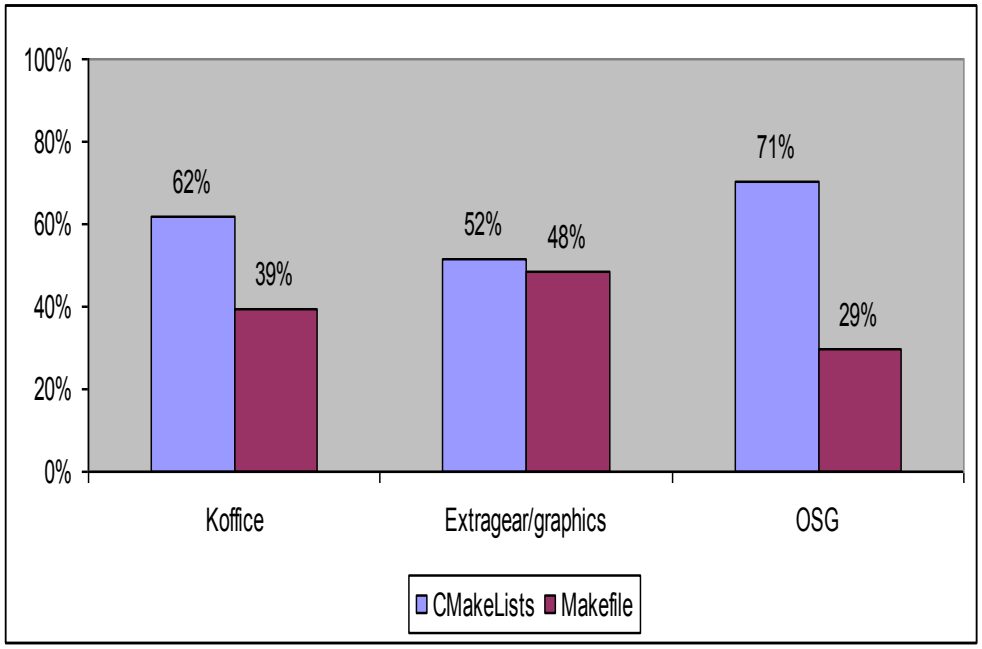


Figure 5-9. Frequencies of the two build management documents. Percentages are compared to total modified build management files for the *non-adaptive* commits in the three examined system

The open systems that we examined are prime examples of successful open-source development that involve a lot of APIs modifications. We believe that our approach is applicable to any software development with a practice of committing related files together in a software repository and change logs with good quality. However, we do not claim that the obtained results likely be able to operate with a high accuracy to a wide range of systems, such as the closed source system or commercially developed software.

Essentially, the assessment of our investigation is relied on manual annotation that we performed early on the commits. Therefore, it is possible that the output of the examination study will involve false positive and false negatives. This is an interesting

issue of future investigation. On the other hand, developers have none-standard as when to commit. Thus, some adaptive maintenance tasks are accomplished via several commits, and some commits contain adaptive and other types of maintenance actions.

5.6 Related Work

The software repositories, containing the historical data (and metadata) of an evolving software system, are useful for extracting knowledge regarding changed files within the framework of the commit analyzing and understanding researches.

The characteristics of the modification requests for several open source software projects are examined by German [German 2004] . In this study, several observations regarding the number of changed files were obtained. Firstly, the modification request corresponding to the bug fixing maintenance affected few files, while the comment requests contain a large number of files. Secondly, the typical modification request is small and contains at least two modified files. Finally, the files that were modified by a modification request are belongs to the same module. Similar study was performed in [German 2006] The study focused on studying different types of modification requests regarding number of touched files. From this study, the author observed that the same developer modified most of the files that were changed several times. On the other hand, the study state that a few cases where multiple developers changed a set of common files. In [Gall et al. 2003] , CVS logs are used to find the relation between classes and files that might not be found by other methods, such as call graphs.

Characterize what a typical or normal commit looks like regarding the number of changed files, number of changed lines, and number of changed hunks is proposed in

[Alali et al. 2008]. Here, the obtained results show that most of undertaken commits were fairly small with respect to the examined size characteristics (e.g., changed files, lines, and hunks), while larger commits do happen with a non-trivial frequency. The main observation of this study is that size measures (e.g., number of changed files) can be helpful to identify the types of maintenance activities being performed.

Predicting the type of undergoing changes in software systems using the characteristics of modified files was proposed by several researches. Mockus and Votta [Mockus and Votta 2000] performed an empirical study on Apache and Mozilla open systems. They found that the number of modified files was strongly correlated with the type of accomplished maintenance. For instance, smallest file are likely modified by adaptive maintenance, while the bug corrective tasks usually affected the larger size files. Spohrer et al. [Spohrer et al. 1985]found that there is a set of files, which are probably contain faults. Therefore, any undertaken change occurred at these files is most likely represent a corrective and bug resolving maintenance.

Several research efforts share a similarity with using machine learning to classify software maintenance changes. One main principle that was used to build such classifier is the using of features extracted from the changed files and modules. Automatic classification of large change request in software systems is proposed in [Hindle et al. 2009]. This machine learning classifier is based on extracting features from submitted commit including features extracted from modified files and modules. These features were number of changed modules, number of changed files per module, and the number of files by their kind of usage. As well, the authors trained the classifier using the linkage

between author and modified files/modules. The obtained results show the capability of using such features in accurately and automatically classifying large maintenance tasks.

Sunghun et al. [Sunghun et al. 2008] introduced a new technique for using machine learning algorithms to determine whether a new change is a buggy or a clean change. This automatic classifier is trained using features extracted from the revision history of a software system. The extracted features were the terms that were generated from changed source code, modified directory/file names, and the change log message. Additionally, the introduced change classifier can be applied to predict the existence of bugs in software changes.

Our investigation is distinguished from these literatures by focusing in recognizing the characteristics of source and non-source code files that were changed because of adaptive maintenance, and using such characteristics, as extracted features, for a comparison between adaptive and non-adaptive commits.

5.7 Summary

A means to investigate the modified files from different software artifacts, which were changed by the undertaken adaptive maintenance tasks, is proposed. The goal presented here is to distinguish the adaptive and non-adaptive commits in terms of modified source and non-source code files. The study is based on features gathered from the changed files that are extracted from the *path* metadata of the commit logs.

In our examination study, using the collected commits obtained by manually examining the version history to distinguish adaptive from non-adaptive commits of the

KOffice, Extragear/graphics, and OSG open source projects as an oracle, we investigate the main characteristics of adaptive modified files.

The study uncovered four main results. First, adaptive commits never affect other artifact files without changing the source code files. However, non-adaptive commits (e.g., 13.8% for KOffice system) changed other artifacts without any modification to the source code files. Second, adaptive commits affect more file kinds comparing with non-adaptive commits. For instance, for OSG system, 8% of adaptive commits and 2% of non-adaptive commits affect 5 or more kinds of files.

Third, most of files are modified by the same type of commit (e.g., either adaptive or non-adaptive commits). There were a few cases where a set of common files (e.g., 10% of source code files and 2% of non-source code files for KOffice system), where changed by both type of maintenance. Finally, adaptive maintenance modifies only two kinds of artifacts (e.g., build system managements and extensible mechanisms), while the non-adaptive maintenance affected all software artifacts.

Therefore, our preliminary results show that name, changing frequency and kind distribution of modified files have strong discriminative power as an indicator for distinguishing between adaptive and non-adaptive maintenance activities, suggesting the possibility of using those features to support adaptive change prediction with considerable precision.

CHAPTER 6

Stereotype of Adaptive Modified Methods: A Case Study

Knowing and understanding what types of changes are happening by a given commit is very precious to the developers. For instance, characteristics of a given commit in the context of how it impacts the behavior or structure of the system assist developers in understanding what maintenance activities are occurring in a given commit. This knowledge, in turn, can explicitly be documented to help developing an efficient approach that provides the ability to identify the type of undertaken commits (e.g., adaptive, corrective, or perfective).

During system evolution, software maintenance tasks modify the system through the changes in structural, behavioral, creational, and collaborational characteristics that are realized in system classes and methods. Here, an additional knowledge is derived from the stereotypes [Collard et al. 2010] of modified method taking place in each commit. To accomplish this, we investigate the frequency of method stereotype distributions for the KOffice package, as a case study, by examining both adaptive and non-adaptive commits. The goal of this study is to learn how such distribution of method stereotypes can be an indicator of automatically identifying adaptive commits.

6.1 Method Stereotypes

Method stereotypes are generalizations that specify some intrinsic or atomic behavior of a method and indicate the responsibility and roles of that method within the

system [Dragan et al. 2006]. Similarly, stereotypes for classes are also used to describe their role and responsibility within a system's design [Dragan et al. 2010]. In terms of stereotype of methods within a class, we can also recognize what parts of the class are responsible for its creational, behavioral structural, and control tasks. Thus, Method stereotypes can be used to assist in program development.

The taxonomy of method stereotypes is organized by the main role of a method, while simultaneously emphasizing its creational, structural, behavioral, and collaborational characteristics regarding a class's design [Dragan et al. 2011]. On other words, the methods stereotypes are categorized by the data access type (i.e., read or write to the object's state) and by functionality, which is given in the creational, structural, behavioral and collaborational aspects. Table 6-1 [Dragan et al. 2010] provides such taxonomy.

6.2 Method Stereotype Distributions

The distribution of method stereotype for adaptive modified methods based on adaptive commits that were collected from the evolution history of the KOffice package is offered by this section. The main goal of this investigation is comparing between adaptive and non-adaptive commits in terms of stereotype distribution of modified methods occurring in both types of commits. Moreover, identifying this distribution will help developers in understanding the responsibilities and roles of methods being modified by adaptive maintenance activities.

Table 6-1. A taxonomy of method stereotypes.

Stereotype Category		Stereotype	Description
Structural Accessor		get	Returns a data member.
	Behavioral	predicate	Returns Boolean value.
		property	Returns information about data members.
		void-accessor	Returns information through a parameter.
Structural Mutator		set	Sets a data member.
	Behavioral	command	Performs a complex change to the object's state.
		non-void-command	
Creational		factory	Creates and/or destroys objects.
Collaborational		collaborator	Works with objects.
		controller	Changes an external object's state (not this).
Degenerate		incidental	Does not read/change an object's state.
		empty	Has no statements.

Our investigation approach starts by automatically identifying and labeling all methods in the examined systems with their stereotype to construct re-documented source code. For this step, we used the *StereoCode* tool constructed in [Dragan et al. 2006], which tag each method in the system by the appropriate stereotype. Then, we developed a C++ tool (using tools such as Unix *diff* utility) to extract all modified methods per commit from the version history of the examined system. For those extracted methods, we then mine their stereotype from the re-documented source code of the corresponding system. This information is then collected and a distribution of each method stereotypes in adaptive and non-adaptive commits is calculated. Figure 6-1 shows the method extraction and stereotype mining steps of our developed tool.

Table 6-2 presents the frequency distribution of stereotypes of modified methods occurring in adaptive and non-adaptive commits for KOffice system. As can be seen in Table 6-2 , a broad range of adaptive modified methods, nearly 60% are of type *Command Collaborator*, while the frequency of this type is only 18% in non-adaptive commits. Generally, most of the command stereotypes (e.g., *Collaborational-Command* , *Collaborator* , and *Command*), which perform a complex change to the states of objects, do happen with higher ratio for adaptive commits rather than the case of none adaptive set. For instance, about 10% of adaptive modified methods (comparing with 6% of none adaptive methods) can be classified as *Command* stereotype.

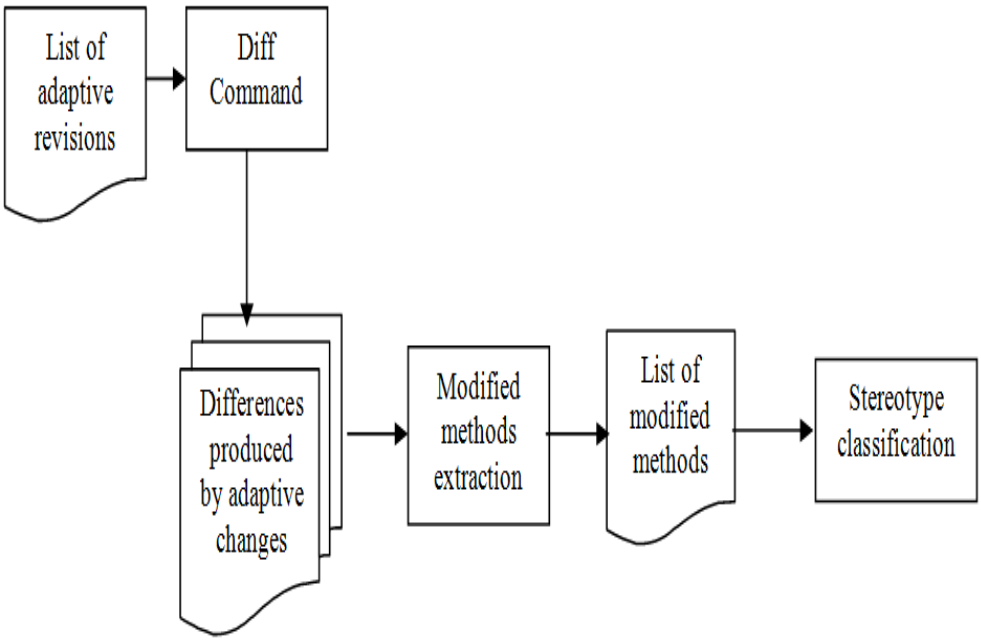


Figure 6-1. Approach to identify stereotype of modified methods for adaptive commits.

Table 6-2. Frequency distribution of stereotypes of modified methods occurring in undertaken commits of KOffice system.

Stereotype	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
Command collaborator	59.48%	17.80 %
Non-void-command collaborator	9.56%	11.44%
Command	9.65%	6.32%
Property collaborator	4.67%	8.25%
Collaborational-command collaborator	3.32%	1.32%
Get	1.70%	4.38%
Predicate	1.51%	7.96%
Collaborator	1.43%	1.07%
Predicate collaborator	1.36%	1.38%
Property	1.28%	4.50%
Voidaccessor collaborator	1.28%	2.49%
Get collaborator	1.21%	2.12%
Set collaborator	1.06%	2.08%
Set	1.06%	4.91%
Command collaborator stateless	0.83%	1.66%
Collaborator empty	0.60%	1.03%
Factory	0.00%	9.12%
Unclassified	0.00%	1.49%
Voidaccessor	0.00%	5.21%
Empty	0.00%	4.47%
Nonconstget	0.00%	1.00%

On the other hand, the remaining stereotypes occur with higher frequency for non-adaptive commits rather than the adaptive commits. Additionally, all adaptive modified methods are classified and non-empty methods, while some modified methods by non-adaptive commits are empty or unclassified methods.

In an attempt to get a zoomed-in picture, we further process the stereotype of modified methods to the granularity of stereotype category, see Table 6-1. The stereotype category distribution aggregates the data and highlights the degree of responsibilities among modified method by occurred commits. Table 6-3 and its histogram in Figure 6-2 show the frequency distribution of stereotype categories for the examined system. Note that, some stereotypes can be placed in several categories. For example, *Command Collaborator* is placed in *Collaborational* (collaborator) and in *Structural_Mutator* (Command).

Table 6-3. Stereotype category frequency in the adaptive commits and their frequency in none adaptive commit for KOffice system.

Stereotype	Frequency	
	Adaptive change commits	**Non-adaptive change commits**
Collaborational	84.4%	50.6%
Structural_Mutator	84.5%	45.49%
Structural_Accessor	11.82%	37.29%
Creational	0.00%	9.12%
Degenerate	0.00%	4.47%
Collaborational	84.4%	50.6%

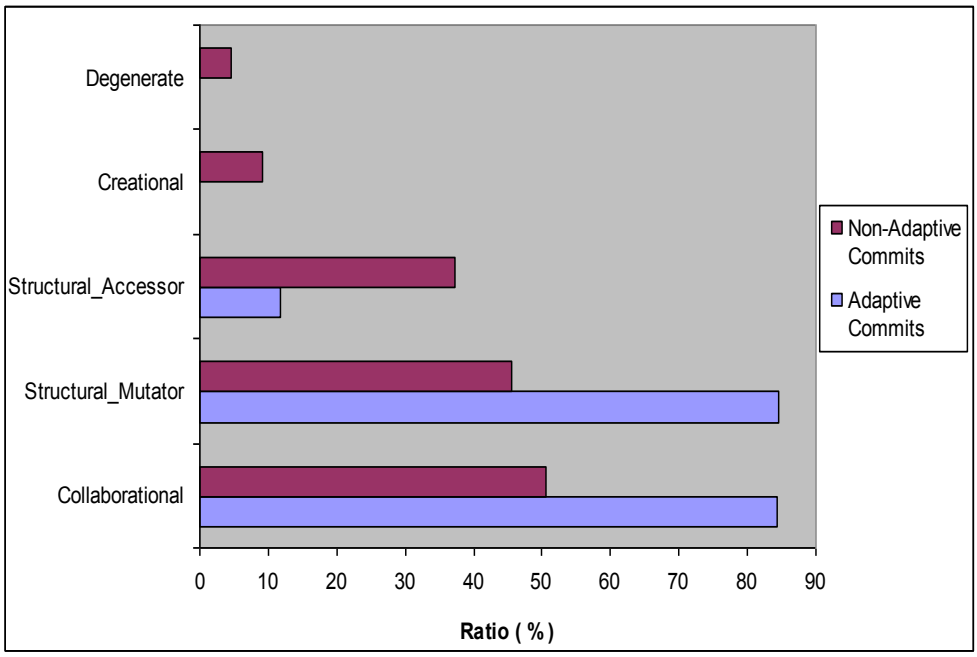


Figure 6-2. Frequency distribution of stereotype categories for commits of KOffice system.

As shown in Figure 6-2 , nearly 85% of adaptive modified methods (comparing with 50% of non-adaptive modified methods) can be categories as *Collaborational* or *Structural_ Mutator* methods. This means, most adaptive changes influence methods that characterize the communication between objects and how objects are controlled in the system (e.g., API roles). Therefore, adaptive changes represent complex modifications of relationships between classes and objects within the system. Additionally, the frequency of *Structural-Accessor* occurs higher in non-adaptive commits comparing with adaptive commits. For instance; about 40 % of non-adaptive modified methods (comparing with 10% of adaptive modified methods) can be classified as *Structural_Accessor*. On the other hand, the *factory* methods, which are related to creation of objects, were never

touched by adaptive maintenance, while about 10% of non-adaptive modified methods are *factory*. As a result, if a commit involves a creational or degenerate method, then this commit is most likely non-adaptive commit.

Here, the presented stereotype distributions make up the comparison between adaptive and non-adaptive commits and provide us with a basis for the automatic identification of adaptive commits.

6.3 Stereotype-Based Commit Categorization

Now, we present a categorization of commits based on stereotype of modified methods. The process of commit categorization is influenced by the results that were obtained in the previous section. The goal of this categorization is to address following questions:

- Do modified methods in an adaptive commit have same stereotype?
- How many different kinds of Stereotype categories are in each adaptive commit?

Here, we define a *stereotype-based* size as a new measure. This measure can be defined as the number of stereotype kinds occurring in a given commit. For instance, if a commit modify 10 *Command* and 5 *Get* methods, then this commit only modify methods with 2 kinds of stereotype. Thus, the stereotype-based size for this commit is 2.

To answer above two questions, we need to categorize commits using stereotype-based size measure, and then use a descriptive statistics method to classify them into different categories. For each commit, the stereotype-based size is computed. We use the values of this measure as the data points for classification using the 5-Point summaries method. The boundaries of the defined regions using the stereotype-based size are shown

in Table 6-4 and the outcomes of commit categorization histogram for KOffice are given in Figure 6-3.

Table 6-4. Defined categories by stereotype-based size using five-point summarization.

Boxplot	Range
Extra-Small	1 - 1
Small	2 - 3
Medium	4 - 6
Large	7 - 9
Extra-Large	>=10

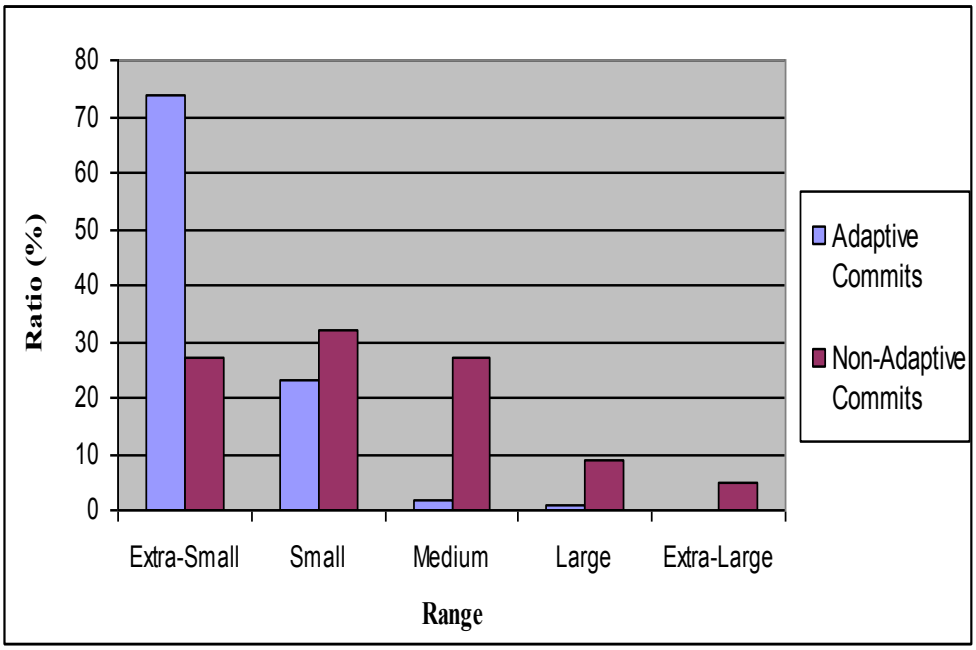


Figure 6-3. Histogram for the number of commits distributed over stereotype-based size for the KOffice system.

From Figure 6-3, we see a number of interesting trends. In particular, we see that even adaptive commits touch more methods rather than the non-adaptive commits as shown in CHAPTER 4, the adaptive modified methods are most likely of the same stereotype. This gives some credence to the idea of using stereotype of modified methods for adaptive commit identification approach.

To be precious, we calculated the level of significance of these measures using the *Mann-Whitney U* non-parametric tests [Hinkle et al. 1995]. The obtained *p-value* is 0.0312, which is less that the significance level of 0.05. Thus, we can observe that there is a significant difference between the adaptive commits as compared to the non-adaptive commits from the perspective of stereotype-based size measure.

On the other hand, from module-based and file-based measures, the adaptive commits being system wide and on average large, as discussed in CHAPTER 4. However, the results in Figure 6-3 show that an adaptive maintenance task changes several files in same style through modifying methods of same role and responsibilities rather than non-adaptive activities.

Here, we want to determine the range, measured by the number of stereotype categories, in which most of the adaptive commits fall (i.e., how many stereotype categories of a typical adaptive commit). Let us consider the histogram given in Figure 6-4 for the number of stereotype categories. Our observation is the right skew to the distribution. That is, large portion of the adaptive commits (e.g., 85%) are with only 1 or 2 categories. The explanation that the majority of adaptive commits are occurring with 2 categories, is because of the stereotype *Command Collaborator* (classified as both

Collaborational and *Structural_Mutator*), which is occurred with higher frequency in adaptive commits.

Moreover, as shown in Figure 6-4, the number of stereotype categories can be used as criteria to recognize between adaptive and non-adaptive commits. For instance, if a commit with 4 or 5 stereotype categories, then this commit is most likely represents a non-adaptive commits.

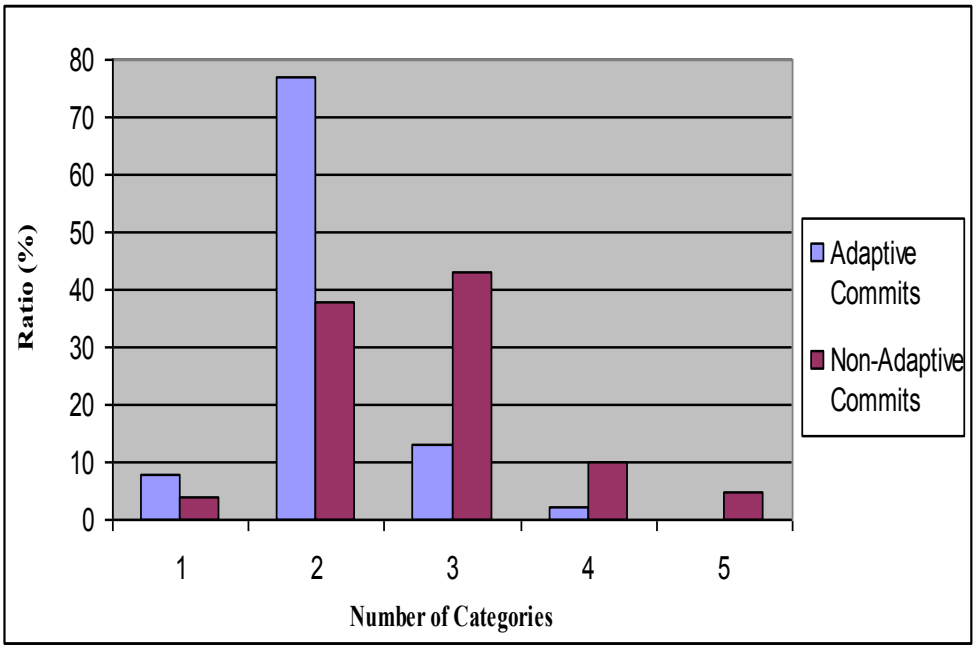


Figure 6-4. Histogram for the number of commits distributed over number of stereotype categories for the KOffice system.

Thus far, the obtained results show that knowing and understanding what stereotypes of modified methods are occurring in a given commit is very valuable to recognize adaptive commits from non-adaptive set. The results demonstrate that the

distribution of method stereotypes is comparable between adaptive and non-adaptive maintenance activities.

6.4 Threats to Validity

The assessment of using stereotype distributions to compare between adaptive and non-adaptive commits is subject to a number of threats to validity. The rules for stereotype identification are subjective and may differ depending on differences in subject's understanding.

In this case study, we investigate and analyze modified methods occurring in the commits of KOffice system. This system represents a successful open source system that involves a lot of APIs migrations, with a good portion of development history. Therefore, we cannot demonstrate that the obtained results are applicable for other systems, including other open source systems or commercial systems.

The adaptive maintenance that we investigate in this study was associated with migration to new complier and platform (e.g., move from Qt 3 to Qt 4). Then, the obtained stereotype distributions may not valid for other causes of adaptive maintenance (e.g., changes to underlying operating systems).

Developers may not commit all changes associated with a specific problem through one commit. That is, one adaptive problem may be accomplished within several commits, and so some commits contain adaptive and other types of maintenance actions. This committing manner may represent a threat to validate our obtained results.

6.5 Related Work

The related work broadly falls into two categories: the using stereotype for program comprehension and for commit characteristics analysis. The role and effect of using stereotype in understanding and constructing knowledge in the model understanding and comprehension of class and collaboration diagrams from the perspective of UML model is presented by Kuzniarz [Kuzniarz et al. 2004; Staron et al. 2006]. The proposed empirical study, which was performed for the telecommunication domain, shows the assistance of using stereotype domain in serving and improving system comprehension, such as saving the time required to answer the comprehension questionnaires for each UML model. An empirical study regarding comprehension tasks on Web applications is presented by Ricca [Ricca et al. 2006].In this study, the conducted experiments aimed at answering whether using Web specific stereotypes that were added to enrich the standard UML diagrams with gives any considerable improvement to the understandability of the Web applications. The results indicate that such addition of stereotypes has a statistically noteworthy improvement effect on the successive comprehension understanding activities.

Additionally, several studies investigated the usefulness of using class stereotype to enhance working with UML diagrams, such as [Andriyevska et al. 2005; Yusuf et al. 2007; Sharif and Maletic 2009]. These studies show that a significant improvement in performance accuracy is achieved when using class stereotype to understand UML diagrams.

On the other hand, further researches from the perspective of commit characteristics were introduced. Alali [Alali et al. 2008] inspected the properties of commits with respect to several size measurements. Also, their study shows that there is commonality of vocabularies and the frequency of their usage even over different projects. An inspection of the CVS modification requests and commits was performed by German [German 2004]. The inspection is focused in modification requests in term of number of files and their kind, number of modifications per month, the frequency of change per file and module, and most active contributors. In [German 2006], the author study the interaction between modification authors of and the coupling between modified files by examining modification requests that are stored in the CVS system. Similarly, changes coupling between files were extracted from change commits through several proposed approaches [Gall et al. 1998; Gall et al. 2003; Zimmermann et al. 2005].

However, a little effort has been spent in examining characteristics and distribution regarding stereotype of touched methods that are occurring in undertaken commits. Natalia [Dragan et al. 2011] proposed an approach to investigate the overall characteristics of a given commit in the context of how it impacts the behavior or structure of classes through examining the stereotype of added/deleted methods. The authors defined commit signature as new measure for commits analyzing, which represents the frequency distribution of stereotypes for the added/deleted methods by a given commit. The study shows that signature can provide developers with valuable information about the influence of a commit regarding about what types of design changes were resulted from this commit. Subsequently, the authors used the impact of

change by each commit, which extracted from the signature, to classify commits into several categorize (e.g., structure modifier, state access modifier, behavior modifier, and object creation modifier).

Our work is closed to the one introduced in [Dragan et al. 2011]. However, our work is distinguished by focusing in stereotype of modified methods by adaptive commits and comparing between adaptive and non-adaptive commits in term of stereotype distributions occurring in these commits.

6.6 Summary

We presented a case study to identify stereotype of adaptive modified methods for commits in open source KOffice system. Our investigation includes stereotype of modified methods, stereotype category distribution, and commit categorization based on stereotype-based size, and number of stereotype categories.

The investigation results show that approximately 60% of adaptive modified methods are of type *Command collaborator*, and nearly 85% of those methods are categorized as *Collaborational* or *Structural_Mutator* methods. Also, the obtained results show that all adaptive modified methods are classified and none of them are empty methods. Additionally, the adaptive maintenance tasks never modify any creational or degenerate methods.

What is more, we found that large portion of adaptive maintenance tasks touched methods of same role and responsibilities rather than non-adaptive activities.

If these characteristics hold for a broad range of systems, we can hypothesize that knowledge extracted from stereotype of modified methods provide us with a basis for the automatic identification of adaptive commits.

CHAPTER 7

Automatic Identification of Adaptive Commits using LSI

As described in CHAPTER 4, we examined the vocabulary of the associated adaptive commit messages. There are number of distinguishing commonly terms used in these messages. Specifically, the terms *port*, *support*, *fix*, *remove*, and *replace* were all used in high frequency within the associated commit log messages.

If such characteristics hold for a broad range of systems, we can hypothesize that a large portion of adaptive changes, for a specific adaptive maintenance tasks (e.g., porting of Qt) can be founded via automated or semi-automated methods. That is, we can identify all large changes involving an API migration with commit log messages of a certain type. Therefore, our vocabulary examination supports possible efforts to automatically identify such adaptive commits using information retrieval methods. The log messages can be compared via information retrieval methods such as vector space models or latent semantic indexing.

In this chapter, we offered an adaptive change identification approach by means of using information retrieval method, latent semantic indexing (LSI) [Deerwester et al. 1990; Marcus et al. 2004]. The main principle of such support is the existence of specific identifiable vocabularies in the adaptive commit log messages that were discovered by our previous investigation. Here, we shall adopt the attractive LSI approach to enhance

the commit classification in order to truthfully tease the adaptive commits from other maintenance types.

7.1 LSI -Based Approach

Identifying adaptive commits in large software systems is a very costly task. Specifically, manually examining version histories to identify adaptive changes is a very hard task, especially to the large corpus of commits already performed [Hindle et al. 2009]. For instance, it took several months to manually uncover the adaptive commits during the time period of 1/1/2006 to 12/31/2010, where there were nearly 38,000 commits. Thus, researches directly aim at enhancing the productivity of software developers as regards to the identification of such commits.

As in the state of art, Information retrieval (IR), such as LSI, is a valuable methodology that can be leveraged to help categorization and clustering textual units based on various similarity concepts [Deerwester et al. 1990; Marcus et al. 2004]. Thus, LSI can be applied on the text messages, the textual description of the undertaken change, of the associated commits of version control system. The main hypothesis of our approach is that the outcome of the LSI clustering involves at least one cluster that is related to accomplished adaptive commits. That is, LSI can place the vast majority of adaptive commits in one cluster because of similarity terms and concepts. Then, we can ask for the relevant adaptive commits in the form of an explicit query, which is derived from the keywords of resultant clusters.

We begin with an overview of the LSI and the associated text clustering, followed with details on how the LSI is applied to allow for adaptive commits extraction.

7.1.1 Overview of Latent Semantic Indexing

Among IR approaches, the LSI is a corpus based statistical technique which is used for inducing and representing characteristics of the meanings of natural language of words and messages to identify their main concepts and usages [Deerwester et al. 1990; Marcus et al. 2004].

LSI is considered as one of the better techniques [Binkley and Lawrie 2010] capable of retrieving the relevant data that is associated with a specific query. Furthermore, LSI is a can deals with synonymy and polysemy. That is, LSI provides the users the ability to determine the semantic similarity between a set of documents.

When using the LSI approach, the corpus for the system that would be inquired, must be build, where the corpus should consist of a set of documents. These documents can represent system components of different granularity levels (i.e., functions, methods, interfaces, and classes). The following subsection gives moiré details about corpus building process.

7.1.2 The Approach

As mentioned before, we offer an adaptive commit recognizing approach by means of employing the IR methods. Among IR techniques, LSI is considered as one of the valuable methodologies able of recognizing the correct data that are relevant to a user query [Binkley and Lawrie 2010]. Therefore, our adaptive commits recovery process we present centers on the LSI component. An overview of our approach is presented in Figure 7-1.

As shown in Figure 7-1, after we extract the version history commits using *SVN* log command, we build a corpus regarding those commits. In this corpus, each document represents a commit message with its author. To create the targeted corpus, following steps are taken.

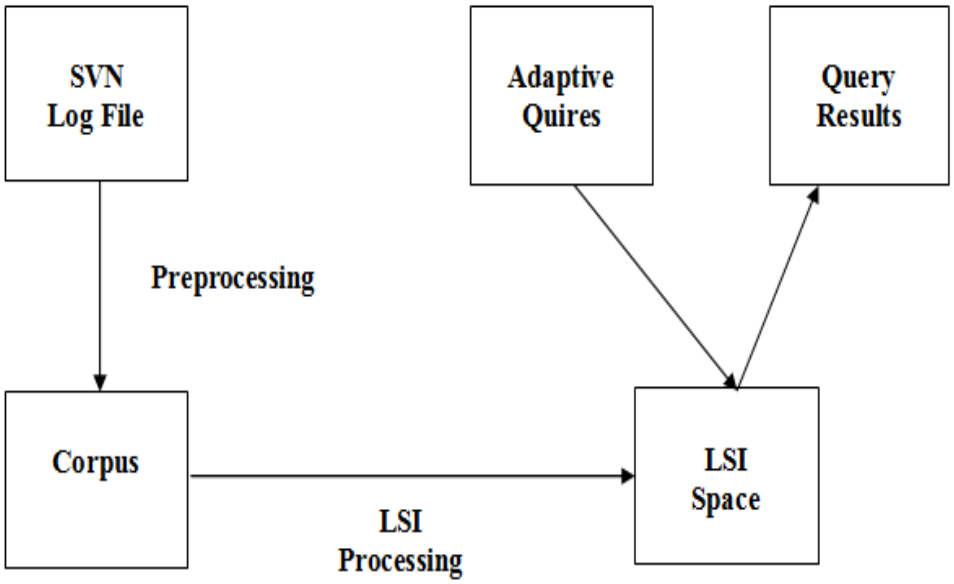


Figure 7-1. Using LSI to identify the adaptive commit

7.1.2.1 Corpus Creation

An important step when using LSI for querying issues is constructing the corpus. A well-built corpus helps in locating the relevant documents (effectiveness measure). To create the commit corpus, we use the following sequence of steps:

- Extraction of commits from version history of the system.

- Extraction of text message and the author of each commit.
- Identifier (term) separations, separations using the standard separators presented in [Maletic and Marcus 2000; Marcus et al. 2004; Revelle and Poshyvanyk 2009]
- Removing stop words, using a list of predefined words
- Divide into documents (Commit level)

At the end, the commit coups consists of a set of documents, where each document is associated with one commit and represents the textual description of that commits along with the person who performed the change. That is, when the preprocessing is completed the system commits history is represented by a set of documents, $S = \{d_1, d_2, ..., d_n\}$, where d_i is a document related to one commit message and its author.

7.1.2.2 Corpus Indexing

Then, preprocessing is applied to the resultant corpus to convert it into an input for LSI. The resultant of such preprocessing is a co-occurrence matrix of vocabulary × documents. Then, the Singular Valued Decomposition (SVD) [Salton and McGill 1983], is applied to reduce the dimensionality of this matrix by exploiting the co-occurrence of related terms. Reduction of dimensionality represents the vital latent aspects of the corpus.

The dimension of the vector is a parameter of the algorithm. It is usually between 100 and 300 [Marcus et al. 2004]. Here, each document d_i in system S will have a

corresponding vector V_i. At this point, the similarities between any two documents $sim(d_i,d_j)$, can be identified by measuring the *cosine* similarities between their associated vectors [Marcus et al. 2004]. By studying and analyzing these similarities, we can identify the semantic similarity regarding system documents and the relations (i.e., main concept) connecting them.

7.1.2.3 Automatic Extraction Queries

The goal of this part is aimed at establishing how well we can automatically generate queries from the terms specified by each generated topic.

When using the LSI, there are two ways in which we can formulate queries for identifying a given concept. Firstly, the query is formulated by the users based on the change request and their knowledge of the software components (i.e., source code). Second option for querying is to create queries that contain words from the resultant LSI clusters source code. These queries can be generated automatically. In our proposed approach, we follow the second option.

Once the query is formulated, a document q representing the query is created and mapped onto the LSI space. Then LSI creates a corresponding vector V_q and retrieves the set of all documents in the system, ranked by the similarity measure to the input query. At this point, the developer will inspect a subset of the documents in the suggested order and identify which ones are actually parts of the concept.

In order to determine how many documents to inspect, we partition the search space based on the similarity measure. Each partition, at step *i*, is made up of documents

d that are greater than a specific threshold δ [Marcus et al. 2004]. Here, we use $\delta = 0.65$ as our threshold since it gives good results.

The main contribution of this work, as mentioned before, is to identify the adaptive commits. In this approach we promote and employ the LSI for automatic queries generation. In details, as a first step our approach perform topic modeling for the commits corpus, more specifically, we use 10 topics. Next, we use two automatic styles for formulating the query. Firstly, the approach uses all terms of each topic as a separate query. We called this query model as TopicAllTerms (TAT).

As a second query formulating style, we generate another set of queries similar to the ones above, but with looking for suitable terms to be included within the query of each topic. In [Kuhn et al. 2007], the authors proposed an approach to decide what the most important terms for each cluster are. The authors found that querying based on specific terms of a topic is better than just using the top most similar terms of that topic. These specific terms include all terms that are very relevance to the current cluster but not common to all other clusters. To compute the relevance of a term to a topic, authors compare the similarity to the current cluster with the similarity to all other clusters. In our work, we follow the same idea by formulating query for a given topic based on selecting the suitable terms, which are strongly related to this topic rather than the remaining topics. For this purpose, we developed a TermAverageModel (TAM) formulating model.

In TAM, the following formula ranks high the terms that are very relevant to the current topic rather than other topics:

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

n=number of topics contains term *i*.

Σ Tijr = Sum of all term *i* relevancy across all topics.

GAvgi: Total Average of term *i* relevancy.

LTijr : local relevancy for term *i* in topic *j*.

Then, term *ti* is used in the query of topic (*j*) using the following decision condition:

If (LTijr >= GAvgi)

Add (Tij)

Else

Discard (Tij)

7.2 Evaluation

To assess the accuracy of our LSI-based approach, we evaluate the results of our automatic clustering of version history commits with the golden set that was obtained by our previous manual investigation of adaptive commits, see CHAPTER 3. Therefore, we applied our approach to the three open source systems (KOffice, Extragear/graphics, and OSG systems).

The evaluation process addresses two main questions. Firstly, can LSI generate clusters contain at least one cluster that is related to the adaptive commits? Or, is the LSI able to place the vast majority of adaptive commits in one group (i.e. topic). Secondly,

Can we automatically generate query that can identify most of adaptive commits? The details of the evaluation study and the obtained results are now presented.

7.2.1 LSI Topic Extraction Results

To extract the adaptive commits of interest, the LSI is asked for performing topic modeling for the commits corpus, more specifically, we use 10 topics. Thus, the first part of the evaluation study is aimed at establishing how well our approach can automatically generate topics contains at least one topic related to the adaptive maintenance.

The software systems that are used for our analysis are those systems that were manually examined for identifying the adaptive commits. The LSI space is created from all version history commits of the studied systems, as discussed in section 7.1. Table 7-1 contains the details of the generated commit corpus for each studied system as well as the dimensionality used for the LSI subspace.

Table 7-1. Details of the LSI generated Corpora for the three studied systems.

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

The clustering of the commits gives another dimension to view relationships among messages of system changes. That is, grouping commits together within a topic

often represents some semantic relationship within the grouping [Maletic and Valluri 1999; Kuhn et al. 2007]. Consequently, once discovered, commits can be represented in a few words in terms of the associated topic keywords.

Here, we conducted the LSI clustering on the generated corpus. Table 7-2, Table 7-3 and, Table 7-4 show the discovered 10 topics along with the top conceptual terms included in each topic. For each topic, the tables show the words that relate to that topic ordered by their relevancies. For example, as shown in Table 7-2, for topic *1*, the word *fix* is the most relevant word to this topic with 0.705 relevancies. Additionally, there is a gap between the relevancies between terms of each topic, where such gap is usually small between the top two terms while it increases with respect to the rest of that topic terms.

When we investigate the terms for each topic, we can see that each topic represents a maintenance category, where the terms of this topic reflect the undertaken maintenance activities. Moreover, the LSI links related tasks in one topic as what developers do in the reality. For instance, the developers do fixing once there were error, crash, and warning. After this fixing process, the developer should re-compile the resultant source code to discover any possible propagated errors. These consequence actions are placed in one topic as shown in Table 7-2, Table 7-3 and, Table 7-4. Moreover, these activities represent the main undertaken tasks during the corrective maintenance. Therefore, we can conclude that corrective commits are grouped in one topic for the three studied systems.

Before further discussing our results regarding adaptive maintenance, we should reiterate that adaptive commits have specific identifiable vocabularies in the commit log

messages (i.e. *port*, *support*, *replace*, and *remove*), see CHAPTER 4. One can argue that in this case the LSI clustering process constructs clusters contains at least one topic associated with the adaptive maintenance. This is because, as shown in Table 7-2, Table 7-3 and, Table 7-4 , these adaptive identifiable vocabularies are placed in at least one topic for each system (i.e. topics *8* and *9* for KOffice, topic *6* for Extragear/graphic system, and topic *5* for OSG system). Moreover, the top term of each of these topics are either *port* or *support*, which are the terms with highest frequencies in the adaptive commits, while these frequencies are very small in the non-adaptive commits.

As mentioned previously, there is at least one topic consists of the main tasks of adaptive maintenance. Therefore, our hypothesis is that using the LSI with commit corpus will yield generating a cluster associated with adaptive commits.

7.2.2 Adaptive commits Allocating Results

From previous clustering results, our hypothesis is that the LSI generates at least one topic that is associated with the adaptive maintenance. For this to be a sound hypothesis, the basic prerequisite is to examine if the vast majority of the adaptive commits can be retrieved using the terms of such topic. Here, we use the two query models, discussed in subsection 7.1.2.3, to help allocating commits for each generated topic and validate our hypothesis.

The evaluation methodology is to conduct two sets of experiments. In the first experiment, we queried the commit corpus with 10 quires using TAT model, where for each topic, see Table 7-2, Table 7-3, and Table 7-4 , we generated a query using all terms

of that topic. We call this experiment the TAT. In the second experiment, we generated the above 10 queries using TAM model. We call this experiment the TAM.

Table 7-2. Topic extraction results for KOffice system. Each topic has 5 terms, where each term has a corresponding relevancy rank.

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)	remove (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 7-3. Topic extraction results for Extragear/graphics system. Each topic has 5 terms, where each term has a corresponding relevancy rank.

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

Table 7-4. Topic extraction results for OSG system. Each topic has 5 terms, where each term has a corresponding relevancy rank.

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

In both experiments, we used the most common measure in the experiments with IR methods: *recall*. Here, our interest is allocating the adaptive commits using such queries. Therefore, for a given query *q*, *Ni* documents will be inspected in step *i*. Among these *Ni* documents, we will identify how many adaptive commit are correctly retrieved (*Ci*), where there is *Ri* correct adaptive commits, which were discovered by our manual identification in CHAPTER 3. That is, the recall measure can be defined as follow:

Recall = #of correct & retrieved documents (Ci) / # of correct documents (Ri)

Figure 7-2, Figure 7-3, and Figure 7-4 show the recall measurements for the 10 quires from both experiments. The recall results show that the majority of the adaptive commits can be allocated by one specific query, 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 system, using either TAT or TAM models.

In contrast, few adaptive commits can be retrieved using other topics, and so these topics addressed other type of maintenance such as corrective, adding new features, or enhancements tasks going on in parallel. That is, applying the LSI on the commit corpus can generate at least one topic, which is related to the adaptive maintenance activities. Moreover, these obtained results provide strong evidence of the semantic similarity between undergoing adaptive changes (i.e., having specific identifiable vocabularies in their commit log messages).

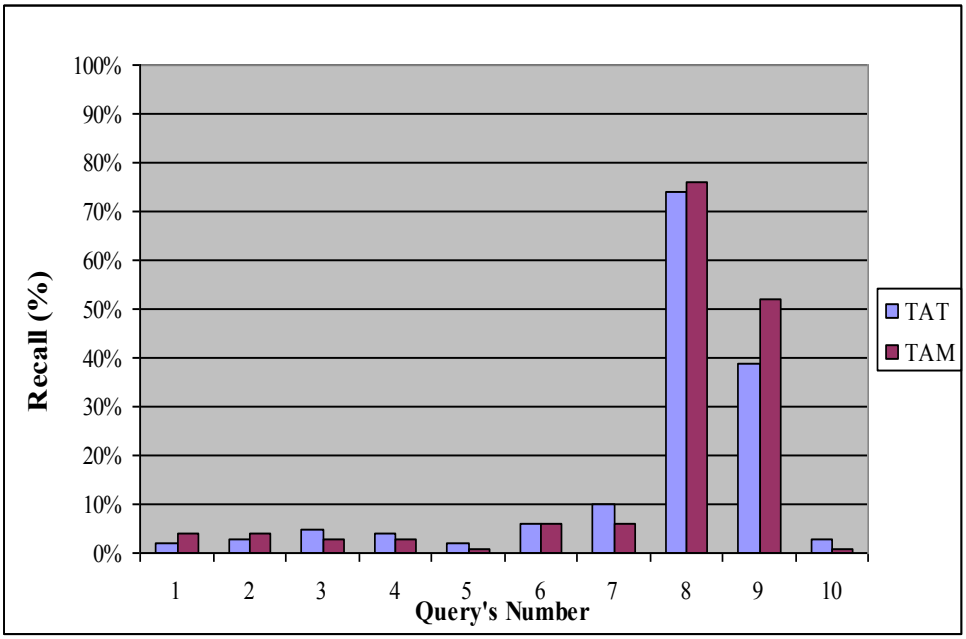


Figure 7-2. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for KOffice system.

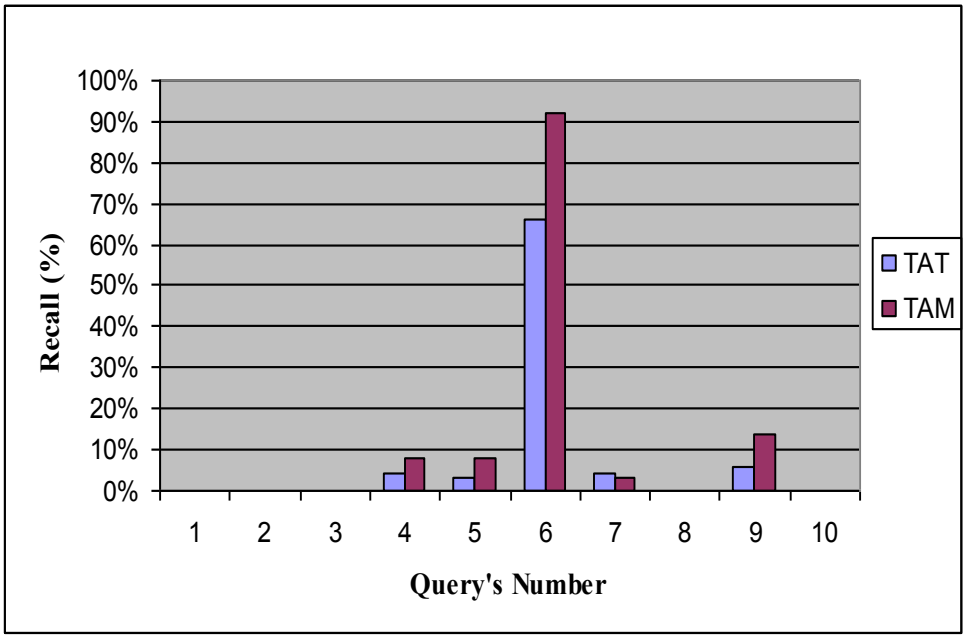


Figure 7-3. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for Exrtragear/graphic system.

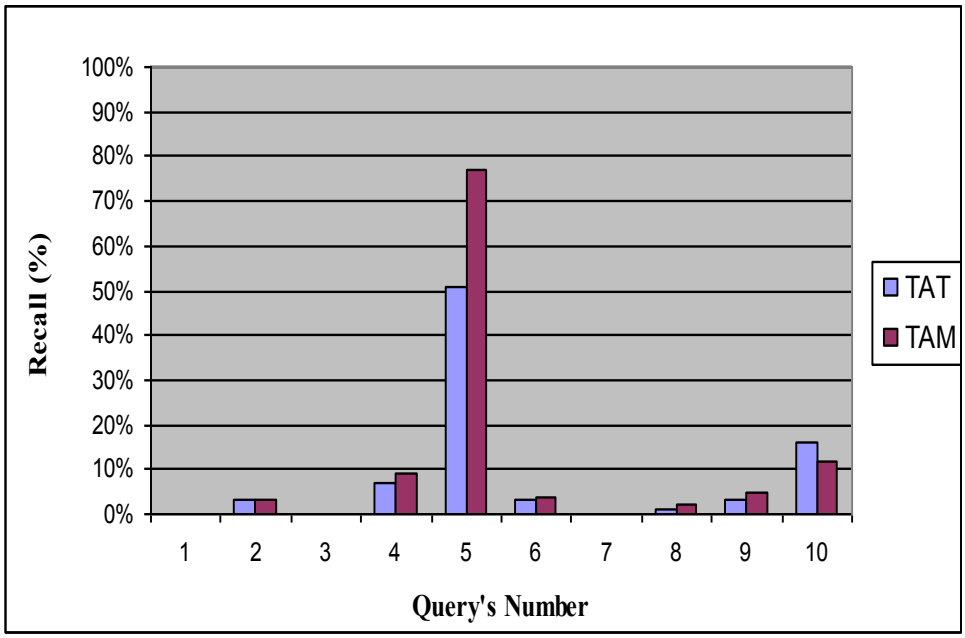


Figure 7-4. Recall values for each query. Query number (*i*) is formatted from topic number (*i*) using TAT and TAM models for OSG system.

On the other hand, TAM model generates better quires, with more than 70% recall, for the adaptive topics (i.e., topic *8* for KOffice, topic *6* for Extragear/graphics, and topic *5* for OSG) comparing with the TAT model. This is because; these topics contain terms that are strongly similar to other topics and generates many correlations. For instance, consider Table 7-2 for the KOffice system. As we can see, the term *add* has a rank of 0.202 with topic *8* (i.e., adaptive topic), while it has a rank of 0.780 with topic *4* (non-adaptive topic). Similarly, the frequency of term "*add"* in KOffice adaptive commits is 14.6%, while the frequency is 19.45% in non-adaptive commits, see CHAPTER 4. Therefore, using term *add* with the query of topic *8* (TAT model) returns

poor results. That is, TAM model helps enhancing the recall values slightly than using all terms in each adaptive topic.

We now take a closer look at a set that is the union of all relevant adaptive commits from the all queries produced by the TAM model. As shown in Table 7-5 , the resultant set hold about 90% of the adaptive commits for each examined system. Given this trend, the LSI can be leveraged to help automatically identify adaptive commits within a version history. This can then form a basis for lower the cost and save the developer's time of identifying adaptive commits for large-scale systems. That is, instead of inspecting all commits in the version history, developers need only to search the commits most relevant to the executed queries. For instance, instead of investigating nearly 26,000 commits for the Extragear/graphic system, developers examine about 2000 commits, which are retrieved by the 10 quires, to allocate and identify nearly 93% of the adaptive commits.

Table 7-5. The Recall values of the adaptive commits using the TAM union set for each studied system.

System	Recall (%)
KOffice	90.1%
Extragear/graphics	93.7%
OSG	87.3%

Now, we examine the relation between all adaptive commits that were retrieved by each query. Here, we address the question: can we use the LSI to classify the adaptive commits based on the supported undertaken task (i.e., tasks that were performed along

with the main adaptive migration tasks)? To answer this question we investigate the involved tasks within all commits returned by each query.

The inspection results are very interested. Commits, which share same supported tasks, were returned by the same query. For instance, all commits returned by query of topic *1* for the KOffice system involves migrating the API code to fix previous error and crashes, which occurred while using the old API code. Thus, LSI is valuable approach to classify the adaptive commits based on the type of the undertaken tasks

7.3 Threats to Validity

There are some threats that may affect the validity of our work results and the ability to generalize obtained results. The open systems that we examined are prime examples of successful open-source development that involve a lot of APIs modifications. However, we do not claim that the proposed approach likely be able to function with a high accuracy to a broad range of systems, such as the closed source system.

Essentially, developers have none-standard as when to commit. Some commit every small successful change early due time constraints, where others wait until the completion of the whole problem. On the other hand, developers don't use one commit to solve one problem, neither contiguous commits. Thus, some adaptive maintenance tasks are accomplished via several commits, and some commits contain adaptive and other types of maintenance actions.

7.4 Related Work

Clearly, a number of approaches have been proposed for a variety of purposes that utilize the metadata stored in software commits using Information retrieval (IR) methodologies for the understanding of software development and evolution process.

Mockus and Votta [Mockus and Votta 2000] proposed an IR-based approach to classify modification requests and commits based on the keyword usage in the textual abstract of the change, where this textual description is essential to understanding why that change was performed. For instance, the appearance of specific terms such as new is an indication that the undertaken change is an adaptive task. The proposed approach proved the helpful of using textual description field to identify the reason for a change and other properties of the change. However, there were numerous cases reported by the authors, where the undertaken changes could not be correctly classified using the appearance of specific terms.

Canfora and Cerulo [Canfora and Cerulo 2005] presented an IR-based method to identify the set of source files impacted by each proposed change request. The method links the change request description and the set of historical source file revisions modified by similar past change requests. The method is evaluated by applying it on bug reports in the Bugzilla repository and the CVS commit messages for several open-source systems.

Dirichlet allocation (LDA) was applied over the commit messages in [Cubranic and Murphy 2003]. In this approach, LDA was able to derive the topics that are being performed by developers at any given period of time, s from a corpus of commit-log

comments. Additionally, the approach allows developers to explore the evolving stream of each topic of development occurring over the time.

LSI has been applied to many software engineering problems such as traceability, program comprehension, and software reuse. Maletic et al. [Maletic and Valluri 1999] were the first to use LSI to cluster source code documents. In this proposed approach, the clusters produced by LSI represent an abstraction of the source code based on a semantic similarity. Therefore, by grouping similar components together a broader understanding of the software system may be achieved. Recovering traceability links between source code and external documentation was proposed by Marcus and Maletic [Marcus and Maletic 2003]. The obtained results prove the capability of applying the LSI with regards to preprocessing and/or parsing of the source code and documentation with lower cost and high flexibility.

Measuring the cohesion of the content of a bug report using LSI is offered in [Dit et al. 2008]. In this approach, the LSI is applied to measure the textual coherence of the user comments in bug report. The obtained results on a subset of Eclipse bugs reports shows that the LSI can help as an indicator of the textual coherence of comments in bug reports, which is also confirmed by the developer.

For the task of feature and concept location in source code, LSI was used in several efforts [Marcus et al. 2005; Poshyvanyk et al. 2006; Poshyvanyk and Marcus 2007]. For instance, in [Poshyvanyk et al. 2006] , the authors proposed a novel hybrid approach to combine LSI with a dynamic technique for feature identification. The approach allows the elimination of the knowledge-based filtering. This has the advantage

that the user does not need to acquire extensive knowledge of the target system a-priori. The authors applied the proposed approach in a new case study for bug location in Mozilla.

Wide ranges of research have been proposed on categorizing maintenance commits based on type of the undertaken changes using commit metadata. Hindle [Hindle et al. 2009] proposed an automatic classification of large changes in software systems into different categories of maintenance tasks. This proposed classification is based on using the word distribution of a commit message, commit author, and modules modified. The authors show the capability of using commit message to provide valuable information about the maintenance class of a commit, where terms of this message can identify the why the undertaken change was occurred.

Commit size, which is derived from the number of touched files was used to classify repository commits in [Hattori and Lanza 2008]. The authors, also, classify commits by the types of development and maintenance activities based on the content of their textual message.

To the best of our knowledge, this is the first work in using LSI to help the process of identifying the adaptive change commits. There is no other work in the literature to cluster version history commits of large scale systems using LSI based on textual description of the accomplished change.

7.5 Summary

A mean to automatically identify the adaptive change commits occurring in a software repository is proposed. Our approach is based on using an information retrieval

method, latent semantic indexing (LSI), to locate adaptive commits of interest in the change log file.

This LSI-based approach uses the textual description of the undertaken change to categorize all the commits into numerous topics. LSI can derive commit clusters contains at least one topic related to adaptive maintenance. Then, these adaptive topics can be used to locate the adaptive commits of interest, using two options of the commit location techniques. One, using all terms of each topic as a separate query and the second is based on deriving query from strongly relevant terms of each topic, as identified by the TAM model.

We evaluated our LSI-based approach on an oracle of commits from three large open source projects. Our evaluation results show the capability of using LSI to generate a topic that is strongly associated with adaptive commits. Then, when applying our querying approaches, we achieve about 90% recall for locating the adaptive maintenance commits in the change log file. Moreover, the LSI has the ability to classify the adaptive commits based on the undertaken supported tasks.

CHAPTER 8

Automatically Uncovering for Traceability Links from Adaptive Commits

Recovering traceability links between the varieties of software artifacts is essential for many large-scale systems to support the quality software maintenance tasks including the adaptive maintenance process for high quality systems. Especially, this is in the case of safety critical system and organizations that adopt standards such as SIL (Safety Integrity Level) or IEC 61508 from the European Functional Safety standards [Alhindawi et al. 2013].

Particularly, these links help understanding the rule and responsibility of each method or file exist in the system. Additionally, such links are useful to support the software comprehension between the source code and other documentations including design, requirements, and test case documentation.

Traceability link recovery has been a vital subject of study for many years within the software engineering community [Kagdi et al. 2007]. Therefore, a number of approaches have been developed to help uncovering the traceability links in large scale software systems [Spanoudakis and Zisman 2005]. However, no single proposed approach is completely center in uncovering a set of traceability links for a specific type of software maintenance task. Here, we look at traceability links regarding a particular adaptive maintenance task that involves the migration of an API.

In this chapter, we present two approaches to discover traceability links concerning the adaptive maintenance. Firstly, we present a means to uncover a set of traceability links in a semi-automated manner via frequent itemset mining. We utilize some manually generated information, from CHAPTER 3, along with frequent itemset mining to uncover a set of traceability links for a specific type of software maintenance task.

Secondly, we show how TraceLab [Dit et al. 2012; Keenan et al. 2012] components can be used to uncover links between adaptive changed documents (e.g., non-source code artifacts) and source code. Our TracLab approach is based on running experiments, similar to those found in [Antoniol et al. 2002; Marcus and Maletic 2003], of using the information retrieval method, Latent Semantic Analysis (LSI) for such recovery process. Additionally, TraceLab will be used as a validation approach for the traceability links that are uncovered from frequent itemset mining appraoch.

The remainder of this chapter is organized as follows. Section 8.1 describes the frequent itemset mining approach that we use for uncovering traceability links from adaptive commits. Section 8.2 details how we validate the uncovered links using TraceLab components. Next are the threats to validity and related works. Finally, our conclusions are presented.

8.1 Uncovering Traceability Link

Recovery of traceability patterns is vital for a variety of investigation for several years of software engineering researches [Gotel and Finkelstein 1994]. A number of techniques have been proposed to assist in the discovery of traceability patterns in open

software systems .However, no approach was proposed to uncover a set of traceability links for a specific type of software maintenance. Here, we look at particular adaptive maintenance tasks, where our approach is to discover sets of files that frequently co-change in adaptive commits that involved the migration of an API.

Our goal is to uncover traceability between source code and other artifacts. This includes user douments (e.g., HTML, XML/docbook, LaTeX and Doxygen), build management documents (automake, cmake, and makefile), HowTo guides (e.g., FAQs), release and distribution documents (e.g., ChangeLogs, whatsNew, README, and INSTALL guides), progress monitoring documents (TODO and STATUS), and extensible mechanisms (e.g., Python, Ruby, and Pearl bindings for an API). These artifacts can be considered software informalisms [Scacchi 2002].

We derive our traceability links by mining adaptive commits to uncover links caused by the adaptive maintenance process. The main goal of this is to help us assess the uncovering of the traceability links between source code files and other artifacts that related to adaptive maintenance tasks. Here, we analyze sets of files that frequently co-occur in adaptive commits by applying a frequent-pattern mining technique (i.e., sequential pattern mining) [Masseglia et al. 2005] . The work presented here is closely related to the approach that is presented in [Kagdi et al. 2007].

8.1.1 Commit Grouping Heuristics

The input data to frequent-pattern mining algorithms are in the form of transactions. A transaction refers to a group of commits that share a common property or occur in the same event (e.g., customer baskets or items checked out together in market-

basket analysis). In [Kagdi et al. 2007], the authors presented three heuristics for grouping related change-sets formed from version history metadata found in software repositories. These presented heuristics are as follow:

- Time Interval: All the change-sets committed in a given time interval represent a single group. The committing time for a given change-set can be extracted from the 'date' metadata. The number of groups is equal to the number of unique time durations over which the commits were committed.
- Committer: All the change-sets committed by a given committer are placed in a single group. The committer for a given change-set can be extracted from the 'author' metadata. Therefore, the number of groups is equal to the number of unique committers of an open source system.
- Committer + Time Interval: All the change-sets committed by a given committer within the same time interval represent a single group. Here, the number of groups is equal to the number of unique committers and time interval combinations.

In our work, we developed a C++ tool to place extracted adaptive commits into groups according to the mentioned heuristics. Here, we decide a calendar day as the time interval for the heuristic Time Interval. The author who contributed a change is mapped to a group for the heuristic Committer. The committers and day combinations are used to generate groups based on 'Committer + Time Interval' approach. We refer to the last heuristics as a *CommitterDay*.

As a case study, we examined three large open source systems, namely, KOffice, Extragear/graphics, and OSG systems. The number of groups and the total number of

adaptive commits involved in these groups for the three heuristics are shown in Table 8-1.

Table 8-1. Groups formed from the adaptive commits by the different heuristics.

Heuristics	KOffice	Extragear/graphics	OSG
Day	100	100	47
Committer	26	17	8
CommitterDay	104	115	48

8.1.2 Frequent Pattern-Mining Approach

According to [Kagdi et al. 2007], the *CommitterDay* is likely to produce better recall and precision and represents quite precise approach to uncover the existence of traceability links from mining software version history. For that reason, we follow the *CommitterDay* heuristic. The groups constructed by this heuristics represent the input transactions for the frequent-pattern mining algorithms. An example of such transactions is shown in Figure 8-1 . The ordered pattern found using this grouping implies that if a file is modified in a pattern by a committer within a given day, then the following or preceding files are likely to be modified by the same committer in the same day.

The number of transactions in which a pattern occurs is known as its *support*. The support of a pattern is the number of groups in which it appears. Therefore, if the support of a pattern is at least a user-specified minimum support, then it is a frequent pattern in

the considered dataset. Sequential pattern mining produces a partially ordered list of files for patterns and as such, we term these ordered patterns.

```
……
26 643858 2 /kchart/ChangeLog
            /kchart/dialogs/KCConfigSubtypeBarPage.cc
26 643862 3 /libs/kformula/ActionElement.cpp
            /libs/kformula/CMakeLists.txt
            /libs/kformula/fontstyle.cc
26 643880 2 /libs/kformula/BracketElement.cpp
            /libs/kformula/fontstyle.cc
……
38 528540 2 /libs/KOfficecore/KoGlobal.cpp
            /libs/KOfficecore/KoGlobal.h
38 528546 2 /kchart/dialogs/KCConfigLegendPage.h
            /kchart/dialogs/KCConfigSubtypeBarPage.cc
38 528568 2 /libs/kformula/ActionElement.cpp
            /libs/kformula/CMakeLists.txt
38 528569 1 /libs/kformula/fontstyle.cc
38 528798 2 /libs/KOfficecore/KoApplication.cpp
            /libs/KOfficecore/KoApplication.h
……
```

Figure 8-1. A snippet of input transactions from KOffice adaptive commits grouped by heuristic CommitDay.

To accomplish this uncovering process, we developed a sequential-pattern mining tool that is based on the Sequential Pattern Discovery Algorithm (SPADE) [Zaki 2001], which utilizes an efficient enumeration of ordered patterns based on common-prefix subsequences and division of search space using equivalence classes. Additionally, it utilizes a vertical input-transaction format (i.e., a set of transactions for each file versus a set of transactions consisting of files) for efficiency. Since our intention is to uncover traceability patterns, the developed tool prints only the uncovered change patterns consist of at least one source code file and at least one file of another type.

8.1.3 Uncovered Traceability Link Data Set

The developed mining tool was performed on the constructed transactions of the three examined systems. The mining execution was performed on several minimum support values. Table 8-2, Table 8-3, and Table 8-4 show the traceability patterns uncovered by the *CommitterDay* heuristics for the three examined systems. The number of uncovered traceability patterns (TP) found with a configuration of the minimum support (σ_{min}) and the minimum, maximum, and average number of files in the uncovered patterns are presented in these tables. Examples of uncovered traceability patterns from adaptive commits are shown in Table 8-5.

Table 8-2. Traceability patterns uncovered from the commits of the KOffice system.

σ_{min}	**Adaptive Commits**				**Non-adaptive Commits**			
	TP	Number of files			TP	Number of files		
		Min	Max	Average		Min	Max	Average
3	89	2	6	3.2	1637	2	12	5.3
5	51	2	5	2.9	510	2	10	4.2
7	19	2	4	2.6	430	2	9	3.9
9	14	2	3	2.4	248	2	7	3.5
11	9	2	3	2.3	154	2	7	3.4

Table 8-3. Traceability patterns uncovered from the commits of the Extragear/graphics system.

σ_{min}	Adaptive Commits				Non-adaptive Commits			
	TP	Number of files			TP	Number of files		
		Min	Max	Average		Min	Max	Average
3	64	2	6	2.9	1510	2	11	4.3
5	45	2	4	2.7	473	2	10	3.8
7	23	2	3	2.5	227	2	8	3.7
9	12	2	3	2.3	147	2	7	3.4
11	6	2	3	2.3	108	2	7	3.1

Table 8-4. Traceability patterns uncovered from the commits of the OSG system.

σ_{min}	Adaptive Commits				Non-adaptive Commits			
	TP	Number of files			TP	Number of files		
		Min	Max	Average		Min	Max	Average
3	39	2	5	2.8	236	2	10	3.9
5	21	2	5	2.6	174	2	8	3.6
7	15	2	3	2.4	103	2	6	3.1
9	7	2	3	2.3	86	2	6	3.0
11	5	2	3	2.2	44	2	4	2.8

Table 8-5. Examples of traceability patterns uncovered from the *adaptive* commits of the examined systems.

System	Traceability Pattern
KOffice	{/kchart/dialogs/KCConfigSubtypeBarPage.cc} → {/libs/kformula/CMakeLists.txt, /libs/kformula/ActionElement.cpp} → {/libs/kformula/fontstyle.cc }
Extragear/graphics	{ /digikam/libs/dimg/filters/icctransform.cpp, /digikam/libs/dimg/filters/CMakeLists.txt } →{/kcoloredit/src/widgets/colorinfo/colorinovisual.cpp , /kcoloredit/src/CMakeLists.txt }
OSG	{/src/osg/Texture.cpp , /src/osgPlugins/dds/ReaderWriterDDS.cpp} →{/src/OpenThreads/common/Config.in}

On the other hand, Table 8-6 and Table 8-7 present the frequency distributions of uncovered patterns, from both types of commits, in terms of sizes (e.g., number of documents in a pattern) at minimum support of *5*. This frequency can serve as an indicator for the number of documents (from different artifacts) that are likely to co-change because of undertaken adaptive maintenance. For adaptive commits, the results show that the majority of uncovered traceability patterns consist of only two documents (the minimum pattern size). However, this result is not valid for the non-adaptive commits, as we can see in Table 8-7.

Table 8-6. Distribution of patterns with regards to the number of documents per pattern for the *adaptive* commits of the examined systems at minimum support of 5.

TP Size	Frequency		
	KOffice	Extragear/graphics	OSG
2	43.2%	48.9%	59.1%
3	33.3%	37.8%	28.6%
4	19.6%	13.3%	9.5%
5	3.9%	0.0%	2.8%
6 or more	0.0%	0.0%	0.0%

Table 8-7. Distribution of patterns with regards to the number of documents per pattern for the *non-adaptive* commits of the examined systems at minimum support of 5.

TP Size	Frequency		
	KOffice	Extragear/graphics	OSG
2	25.6%	30.1%	34.5%
3	23.7%	27.2%	25.5%
4	20.7%	16.3%	21.4%
5	14.8%	16.0%	10.8%
6 or more	15.2 %	10.4 %	7.8 %

Moreover, from Table 8-6 , patterns with as many as *6* documents were never occurred during the adaptive maintenance. But, as shown in Table 8-7, non-adaptive commits involve patterns with *6* or more files with nearly 10%. Thus, the size of traceability pattern can serve as an indication to recognize the adaptive commits from the non-adaptive set.

Next, we further examine the idea of using the traceability patterns for identifying the adaptive commits in the version history systems. At this time, we address the question: is the mined traceability pattern restricted to or depend on the maintenance type? To answer this question we need to find the uncommon patterns between the traceability patterns uncovered from the adaptive commits and those patterns extracted from the non-adaptive commits. The usefulness of such finding can be seen in "how well" the existence of a traceability pattern predicts the type of undergone maintenance change.

Table 8-8 shows the ratio of the uncovered traceability patterns from adaptive commits, which are not occurred in the non-adaptive commits. As we can see that there are considerable ratios of traceability patterns from adaptive commits that are not valid during the non-adaptive maintenance tasks. Furthermore, this ratio increases with higher minimum supports. For instance, for Extragear/graphics system, 50% of uncovered patterns from adaptive commits were never mined from the non-adaptive dataset at minimum support of *11*. As a result, the occurring of many treaceability patterns can be useful to recognize the type of undertaken change (i.e., adaptive or non-adaptive).

Also, our results demonstrate that using traceability patterns for software-change prediction should be correlated with the maintenance type of accomplished changes.

Table 8-8. Ratio of uncovered traceability patterns from adaptive commits, which are not valid for the non-adaptive commits of the examined systems.

σ_{min}	Ratio		
	KOffice	**Extragear/graphics**	**OSG**
3	30.3%	18.8%	23.1%
5	27.4%	28.9%	23.8%
7	31.6%	30.4%	20.0%
9	28.6%	41.6%	28.6%
11	44.4 %	50.0 %	40.0 %

Mining adaptive commits of KOffice system uncovered 89 non-source code files, which have a traceability links at minimum support of *3*. As we see in next section, these identified traceability links will be used in conjunction with TraceLab components to validate "how well" we discover the existence of a traceability links between source code files and other artifacts from adaptive commits, where the uncovered non-source code files that have traceability links will be used to generate queries in the validation process.

8.2 Validation

The main goal of the validation phase is to assess if our uncovering approach can accurately identify traceability links between source code and other documentation associated with a specific adaptive maintenance task (e.g., API migration) occurred in a

given software system. Firstly, we present the conducted experiments in details. Later, we show the identification assessment results.

8.2.1 TraceLab - Based Validation Experiment

To address this validation phase, an experimental TraceLab is being used to support traceability research. TraceLab [Dit et al. 2012; Keenan et al. 2012] is an environment where traceability experiments can be easily constructed and reproduced using reusable components. It uses a visual modeling environment to set up experiments with the components. It also allows experiments to be easily repeated by other researchers [Alhindawi et al. 2013].

In the experiments presented here, we have attempted to use the TraceLab components developed in [Alhindawi et al. 2013], to give us a means of relevant assessment to our obtained results in the previous section. These TracLab components center on the LSI component including LSI Space Builder, LSI Querier, LSI Data Importer, and LSI Data Exporter. For further detail on these components, we refer to [Alhindawi et al. 2013].

Here, we examined the open source system KDE/KOffice as a case study. The input data consists of the source code and the external documentation, which have traceability links with source code files as uncovered at minimum support of *three* in the previous section. Table 8-9 contains the size of the examined system, as well as the dimensionality used for the LSI subspace and the determined vocabulary, see [Alhindawi et al. 2013].

Given that the number of external non-source code documentations is much smaller than the number of source code files, we attempted to trace the links from the external system documentations to the source code. That is, a typical query will be used to find out which parts of the source code are described by a given external documentation, where this external documentation is being used to generate such query.

Table 8-9. Elements of the KOffice source code, documentation and LSI settings used in the experiments.

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

8.2.2 Validation Metrics

Here, a way to determine the "goodness" of the obtained results is needed. We used the most common measures in experiments with information retrieval (e.g., LSI) techniques: recall and precision.

Usually, for a given document d_i, a number N_i of documents will be retrieved in step i using the similarity measure and the defined threshold. Among these N_i documents, the user will identify that $C_i \leq N_i$ of them are actually related or similar to d_i, where there

are a total of $R_i \geq C_i$ documents that are truly similar to d_i. Using such numbers, the recall and precision measures for d_i can be defined as follows:

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

$$Precision = (\#of\ correct\ \&\ retrieved\ documents\ (C_i) / \#\ of\ retrieved\ documents\ (N_i))\ \%$$

The two measures will have values between [0, 1]. If recall = 1, it shows that all the correct traceability links are recovered. While, if the precision = 1, it shows that all the recovered links are correct. With above detentions in mind, the result of higher precision is a lower recall and vice versa.

8.2.3 Validation Results

Table 8-10 shows the obtained results from recovering the traceability links between interested external documentation and source code for KDE/KOffice. Here, the first column represents the threshold (Cosine) value being used; column 2 represents the total number of recovered (correct and incorrect) links; and the last two columns are the precision and recall measures calculated at each used threshold. Figure 8-2 shows a snapshot for the results of running one query sample over 2000 document.

The overall results can be inspected with twofold objectives. Firstly, the obtained results from an experiment with TracLab components shows that mining the adaptive commits found in software repositories can be leveraged to help automatically recover traceability links , which are associated with specific adaptive maintenance task occurring while migrating the system to new APIs, using sequential pattern mining techniques. Secondly, the results of the experiment align well with the original findings from frequent

itemset mining and form a basis for running a variety of experiments to uncover traceability links using TracLab. That is, TraceLab provide an automatic solution for identifying the traceability links using LSI, for a specific type of adaptive maintenance task.

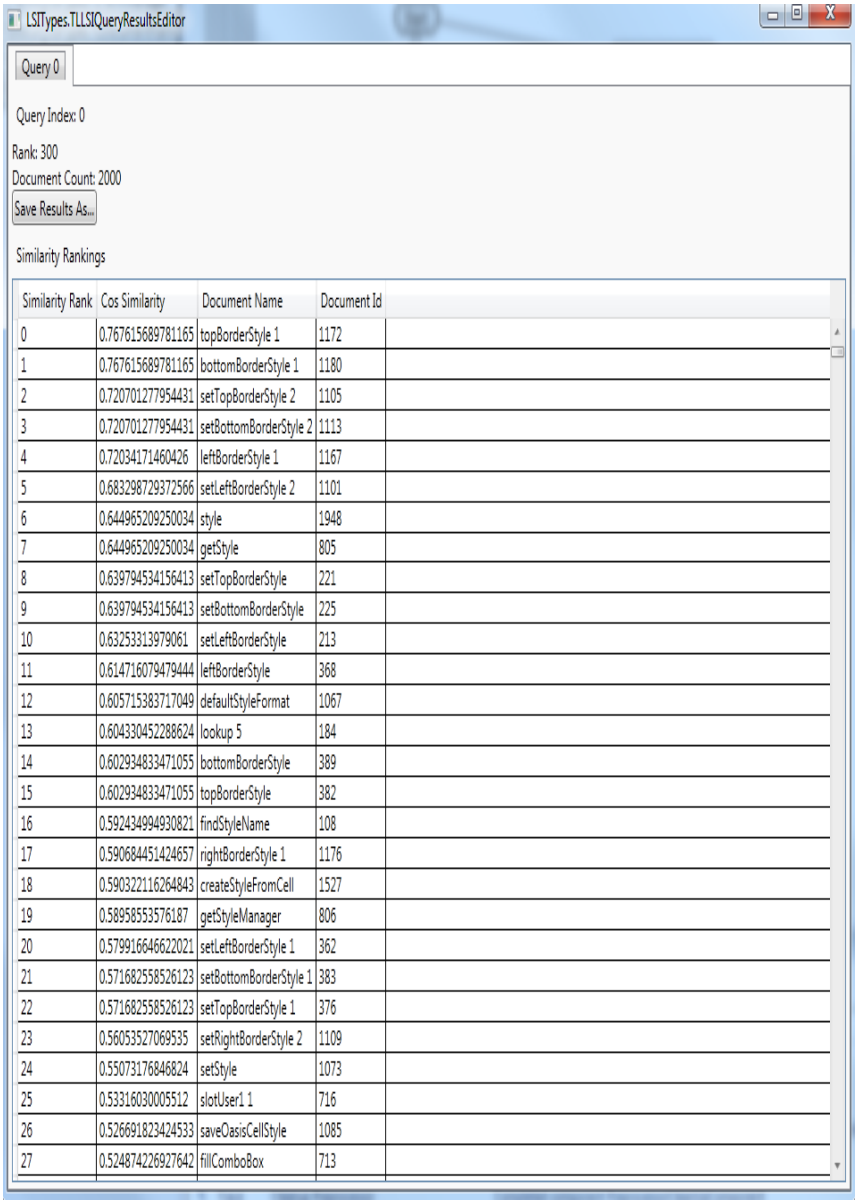
Table 8-10. Recovered links, recall, and precision using cosine threshold for KOffice system

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%

8.3 Threats to Validity

Regarding threats to validity, a number of threats may affect the accuracy and the ability to generalize our approach. The manual inspection to identify the adaptive commits may affect the accuracy of our approach.

The assessment of our approach is obtained from examining prime examples of successful open-source development that involve a lot of APIs modifications. Therefore, we do not argue that the proposed approach likely be able to operate with a high accuracy to a wide range of systems, such as the closed source systems.



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 8-2. A snapshot for the results of running one query sample (Results with first 2000 documents & LSI Dimensionality = 300.

8.4 Related Work

There are two distinct areas of research that are directly related to our work, namely traceability link recovery and TraceLab-based solutions for conducting and creating experiments to assist software engineering researches.

Traceability recovery between source code files and other types of files is essential for developers wishing to learn which external documentations are somehow linked to the current source code file being changed. Hence, a number of efforts have been spent on recovering and extracting the traceability links in existing software systems.

In [Egyed 2003], the author proposed an automated approach to recover and generate traceability interrelationships between software models, scenarios, and code. In this approach, traceability information can be observed by using test scenarios that are typically defined during system development. The incremental behavior of the proposed approach makes it appropriate for both forward and reverse engineering purposes. Spanoudakis et al. [Spanoudakis et al. 2004] developed an automated tool , called traceMaintainer , to modify the existing traceability relations when changes have been made to UML analysis and design models. This tool is based on a set of predefined rules, which identify each change as ingredient steps of broader development activities. Murphy et al. [Murphy et al. 1995] provides a formal characterization of reflexion models to assist in recovering the traceability links between source code and high-level models. This approach helps developers to understand how these high-level models maps to the

specific part of the source code. Consequently, this knowledge supports an engineer in performing a software maintenance task.

Cleland-Huang et al. [Cleland-Huang et al. 2005] proposed a goal centric approach to identify traceability links between non-functional requirements and performance models based on event-notifier design pattern. The proposed approach helps developers to make informed decisions regarding the implementation of every proposed change by analyzing the impact and goals of this proposed change using associated traceability links.

Mining software repositories for uncovering traceability is proposed by Kagdi et al. [Kagdi et al. 2007]. Here, the authors present a heuristic-based approach to recover traceability links between software artifacts through the examination of a software system's version history. The proposed approach is based on sequential-pattern mining algorithms, and can be applied to the commits in the software repositories for uncovering highly frequent co-changing sets of artifacts. The authors evaluated their work on a number of versions of the open source system KDE. The evaluation results demonstrate highly precision predictions of certain types of traceability links.

Several LSI-based approaches are introduced to recover the existence of traceability links in software systems. Marcus and Maletic [Marcus and Maletic 2003] used LSI to automatically identify traceability links from system documentation to program source code. The LSI is applied to all the comments and identifier names within the source code to produce semantic meaning with respect to the external system documentations. A set of experiments and results were presented and compared to a

vector space model (VSM) based recovery technique. The experimental results show that the proposed approach requires less processing of the source code and documentation, implicitly, less computation.

De Lucia et al. [De Lucia et al. 2004] presented a traceability recovery method and tool based on LSI in the context of an artifact management system. The study results show that the proposed approach is able to recover links between source code, test cases, and requirements documents. Jiang et al. [Hsin-yi et al. 2008] presented a LSI-based technique, called incremental latent semantic indexing (iLSI), to automatically manage traceability link evolution and update the links in evolving software. The presented tool use traceability links from previous versions to recover those links for the current version. Therefore, the approach saves the effort and time in recovering traceability links. A number of other researchers [Hayes et al. 2006; Marco et al. 2006; Lormans 2007; McMillan et al. 2009] have also applied LSI for traceability link recovery.

On the other hand, TraceLab is designed to empower conducting experiments for several investigations within the software engineering community. Czauderna et al. [Czauderna et al. 2011] uses TraceLab to evaluate the effectiveness of computing idf using only on the local terms in the query, versus computing it based on general term usage as documented in the American National Corpus. The conducted experiments show the capability of TraceLab to model and execute realistic traceability experiments. An integrated approach for combining orthogonal techniques, such as VSM and Relational Topic Modeling (RTM), using TraceLab components is offered in [Keenan et al. 2012]. In this work, an experiment was executed on six datasets and the outcomes show that

combining RTM with information retrieval methods significantly outperformed stand-alone methods. In [Dit et al. 2012] ,the authors proposed a TraceLab solution for creating, conducting, and sharing experiments in feature location. The authors used TraceLab to evaluate and compare the effectiveness measure between the using VSM and LSI in performing the feature location process. Additionally, the proposed solution provides TraceLab templates and components for rapid creating of future experiments in feature location researches.

Our work is distinguished by utilizing the Tracelab components in recovering traceability links for a specific type of software maintenance, namely adaptive maintenance activities.

8.5 Summary

A means to automatically uncover traceability links consisting of source code files and other artifacts for a particular adaptive maintenance task that involves the migration of an API is presented. In our approach, using the manually identified adaptive commits as an oracle, we build a *CommitterDay* heuristic based approach that uses frequent pattern mining tools to uncover sets of files with different kind of usage that frequently co-occur in adaptive commits.

The obtained traceability links are validated by comparing them with uncovered links extracted by applying the TraceLab-based solution centers on using an information retrieval method, namely Latent Semantic Indexing (LSI), on a set of external documentation and source code of KDE/KOffice system.

The evaluation results are promising enough to demonstrate TraceLab and frequent itemset mining as a solution that can be used to conduct traceability link uncovering research for adaptive maintenance, and aid the growth in the traceability linking area.

CHAPTER 9

Conclusions and Future Work

The dissertation addresses a very relevant problem faced by almost all organizations that depend on large software systems. It is focused on better understanding how software evolves in the context of adaptive maintenance through addressing several research issues. The first is a large case study examining version histories to identify adaptive changes. The second is the understanding the distinguishing characteristics of adaptive commits. Lastly, our third issue is the developing methods to automatically identify adaptive commits and their associated traceability links.

9.1 Conclusion

Ideally, maintenance changes to a software system will be clearly documented in the change log or version history of that system. A case study was undertaken to gather data. We examined multiple years of version history of three open source systems, namely KOffice, Extragear/graphics, and OSG systems. Our examination was done manually and involved inspecting commit log messages, system documentation, development notes, and source code. The outcome of the study was a classification of undertaken commits into either adaptive commits or non-adaptive commits. The obtained results demonstrate that the minority of the commits during the studied time period had to do with the adaptive API migrations.

After identifying the commits involved in the adaptive changes we examined the main characteristics of large portion of accomplished adaptive tasks. We analyzed these adaptive maintenance commits to identify any commonalities or trends. Three characteristics of the commits are examined, namely the size of the commits, the vocabulary of the commits' log messages, and the developers who made the commits. From this study, we observed that a large portion of adaptive changes can be characterized as: involving known API's or language features (e.g., Qt 3 interfaces and Qt 4 interfaces), being system wide and on average large, having specific identifiable vocabulary in the commit log messages (e.g., port, support, and replace), and the developers who perform the adaptive maintenance tasks are few in number and appear to have a broad knowledge of the entire system.

A detailed study has been presented to understand and investigate the modified files from different software artifacts, which were changed by the occurred adaptive maintenance tasks. It was observed that adaptive tasks involved modification of both source and non-source code files. Also, vast majority of files were only changed either by adaptive or non-adaptive commits. Moreover, only two kinds of artifacts (e.g., build system managements and extensible mechanisms) occurred in the adaptive commits, while the non-adaptive maintenance affected all software artifacts.

An application of the method stereotypes to historical data and analysis of adaptive evolutionary patterns of commits stored in a KOffice version control system is presented. The method stereotypes distribution in a commit was used to categorize and analyze both adaptive and non-adaptive commits that impact the system evolution. It was

found that approximately 60% of adaptive modified methods are of type *Command collaborator*, and nearly 85% of those methods are categorized as *Collaborational* or *Structural_Mutator* methods. The observations also indicate that large portion of adaptive maintenance tasks touched methods of same role and responsibilities rather than non-adaptive activities.

An approach has been developed that use information retrieval techniques, namely Latent Semantic Indexing (LSI), to automatically identify adaptive change commits. This efficient LSI approach was developed to reduce the cost and increase the quality of identifying adaptive commits for large scale systems. The evaluation results show that the LSI performs well to construct commit clusters contains at least one topic related to adaptive maintenance. Our results show that the approach accurately retrieves relevant adaptive commits, with nearly 90% recall.

The two techniques, itemset mining and TraceLab-based components, were successfully used to automatically identify sets of traceability links that were resulted by the undertaken adaptive changes. We developed a *CommitterDay* heuristic based approach that uses frequent pattern mining tools to uncover sets of files with different kind of usage that frequently co-occur in adaptive commits. The obtained traceability links are validated by applying the TraceLab-based solution centers on LSI components. The results extend the idea of the existence traceability links produced by a particular adaptive maintenance task that involves the migration of an API.

To the best of our knowledge, this is the first work that examines adaptive maintenance in detailed and systematic manner. Furthermore, our work is the first to

uncover the characteristics hold for a broad range of adaptive commits from different large open source systems. We hope that the comprehensive study that was conducted during the prologue of this work would serve as a useful, introductory source to researchers and practitioners interested in the area of adaptive maintenance.

9.2 Future Work

The work presented in this dissertation forms the basis for a number of avenues of research in adaptive maintenance, and we plan to extend our work in a few directions. We plan to repeat the study with additional causes of adaptive changes, and we plan to extend our investigation by examining other open source system. Moreover, we will expand our study by investigating more characteristics regarding adaptive maintenance. This expansion will aid in providing a more detailed picture of adaptive changes and differences between adaptive and non-adaptive maintenance tasks.

In the field of stereotype investigation, we plan to extend our investigation by examining other open source systems. Also, we plan to use the concept of commit signature to categorize, analyze, and compare adaptive and non-adaptive commits. To achieve our main goal, we plan to design an automatically identifier of adaptive changes rooted in knowledge and information that can be extracted from stereotype of modified methods. We believe this will improve the adaptive maintenance understanding, estimation and managing.

We deeply plan repeat using LSI on locating other types of maintenance commits (i.e., corrective, perfective and inspection). With respect to the topic queries we plan 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. Additionally, we are trying to determine some good heuristics that the approach can use to determine the appropriate threshold value for investigating the retrieved ranked list to determine the stopping criterion.

In spite of the progress achieved, we will further employ the TraceLab environment to be used in supporting other software engineering researches, including classifying change commits, and predicting future maintenance activities.

There will be a supplementary attention to help the software research that concerns the invitation of adaptive transformation approaches. The main step of our next plan will be dealing with the variations on captured source code changes. To be precise, each recognized change category will be generalized in some manner in sequence to generate efficient transformation rules necessary for that category. The objective of this step is to reduce the migration cost and enhance the quality of generating adaptive transformation approaches for large software systems.

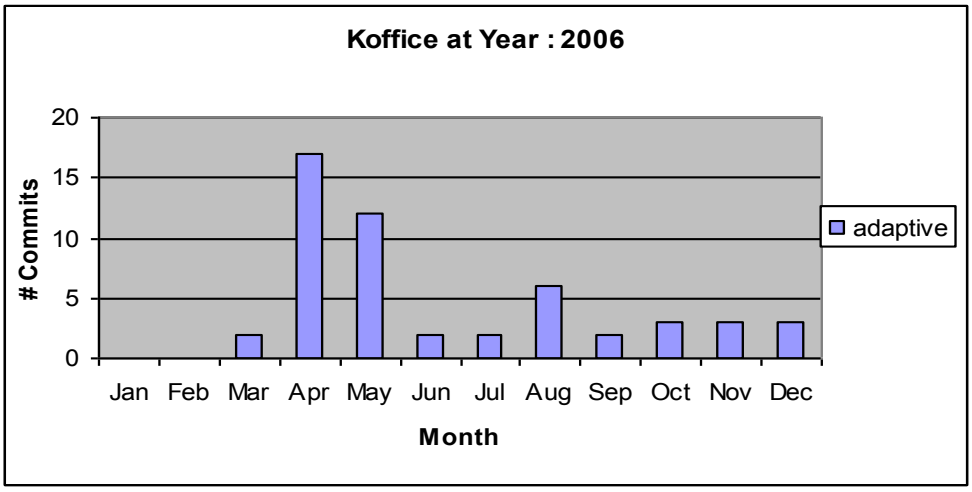
Likewise, an automated porting tool can be implemented using these identified code changes in order to automate the most tedious parts of the porting effort. The planned tool parses the source files and replaces the usage of old compiler/platform entities with their matching in the new version. For example, when porting the software system from Qt3 to Qt4, the planned tool will replace old Qt3 entities by their replacements in Qt4 as in replacing *QButton* by *AQbstractButton*.

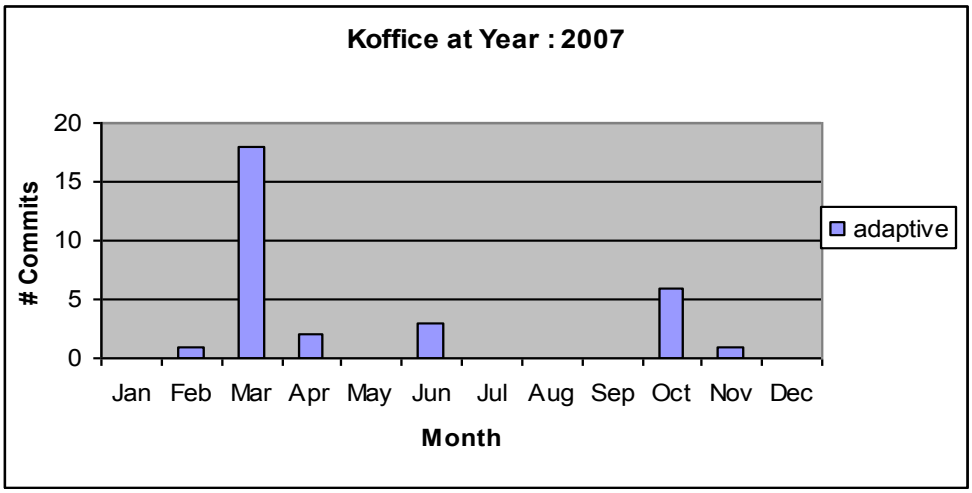
APPENDIX A

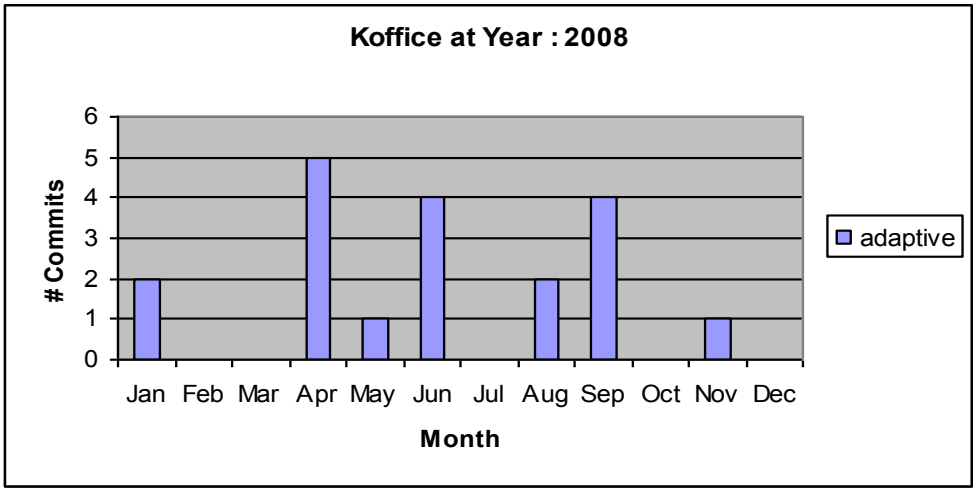
ADAPTIVE COMMITS DISTRIBUTIONS PER YEAR

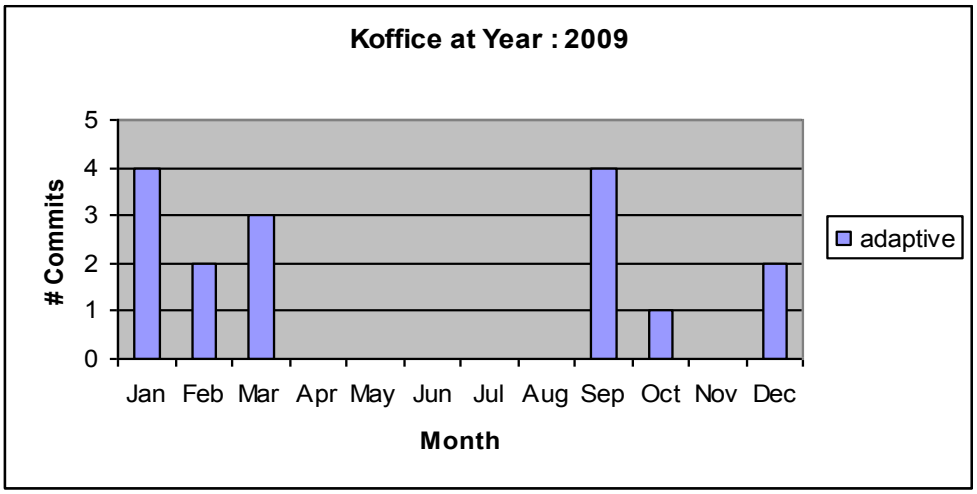
This appendix gives the number of adaptive commits accomplished at each month during the examination time period for both KOffice and Extragear/graphics systems. Additionally, the number of undertaken changes (adaptive and non-adaptive) is presented in this appendix.

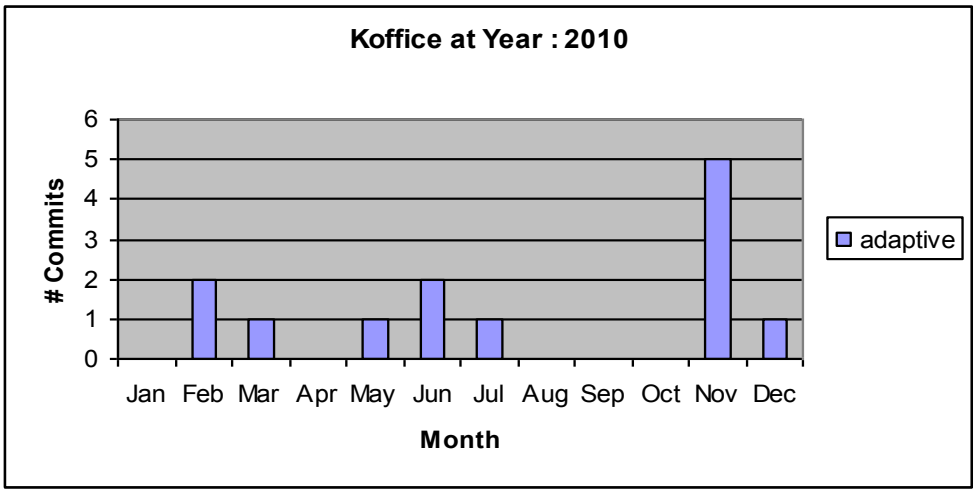
A.1 Distribution for KOffice



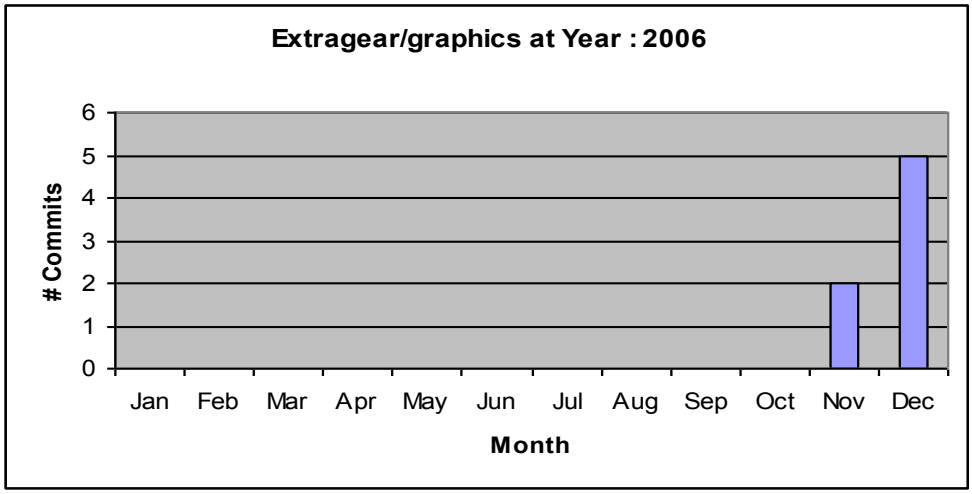

Koffice at Year : 2007
Commits
20
15
10
5
0
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Month
adaptive

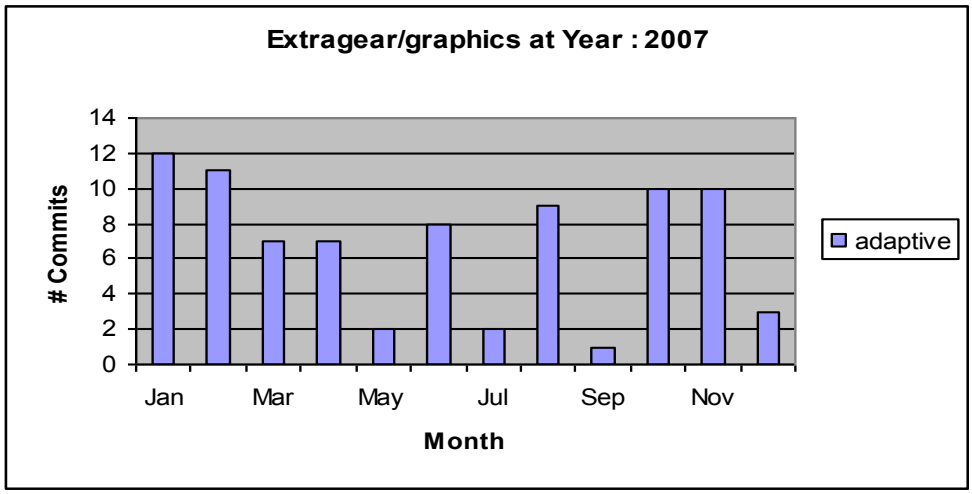

Koffice at Year : 2008
Commits
6
5
4
3
2
1
0
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Month
adaptive

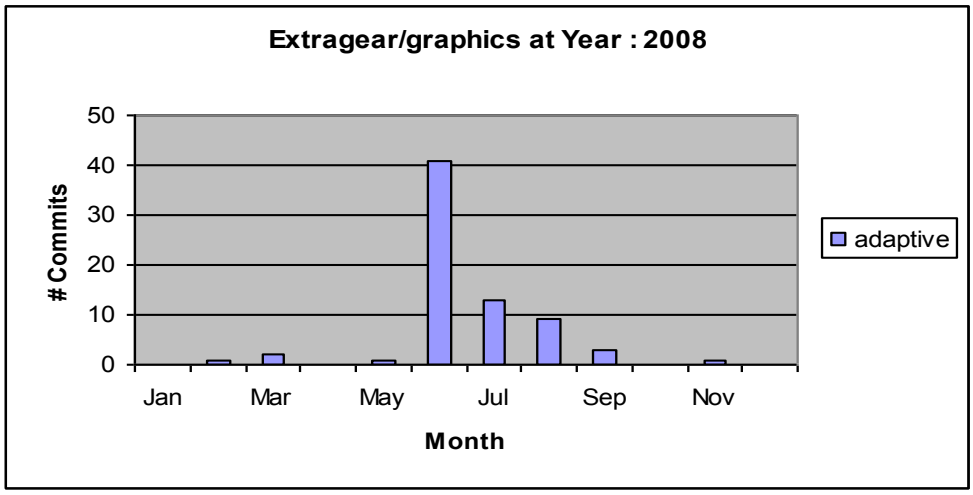

Koffice at Year : 2009
Commits
5
4
3
2
1
0
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Month
adaptive

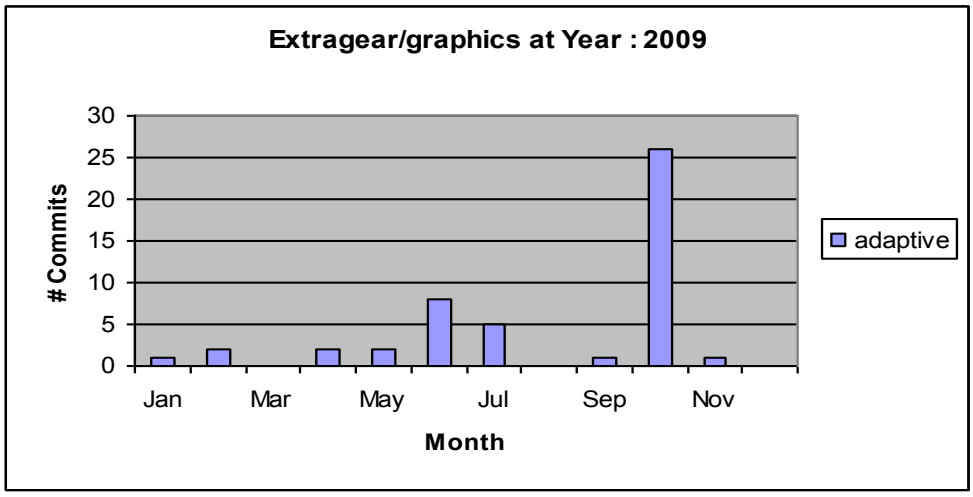

Koffice at Year : 2010
Commits
6
5
4
3
2
1
0
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month
adaptive

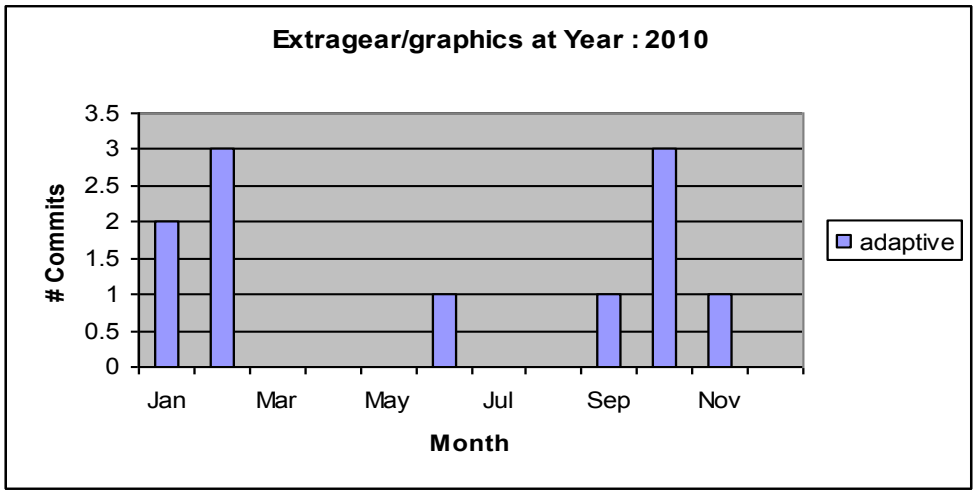
A.2 Distributions for Extragear/graphics






Extragear/graphics at Year : 2008
Commits
50
40
30
20
10
0
Jan
Mar
May
Jul
Sep
Nov
Month
adaptive

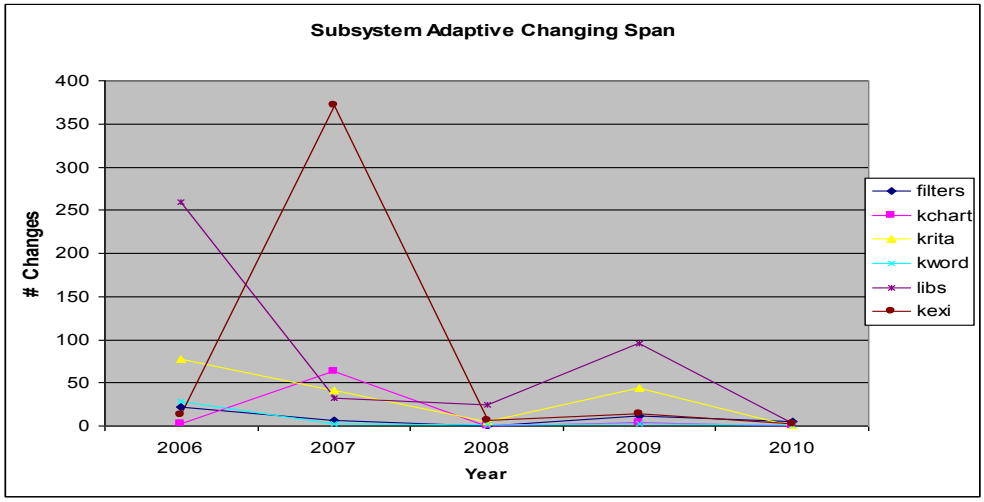

Extragear/graphics at Year : 2009
Commits
30
25
20
15
10
5
0
Jan
Mar
May
Jul
Sep
Nov
Month
adaptive


Extragear/graphics at Year : 2010
Commits
3.5
3
2.5
2
1.5
1
0.5
0
Jan
Mar
May
Jul
Sep
Nov
Month
adaptive

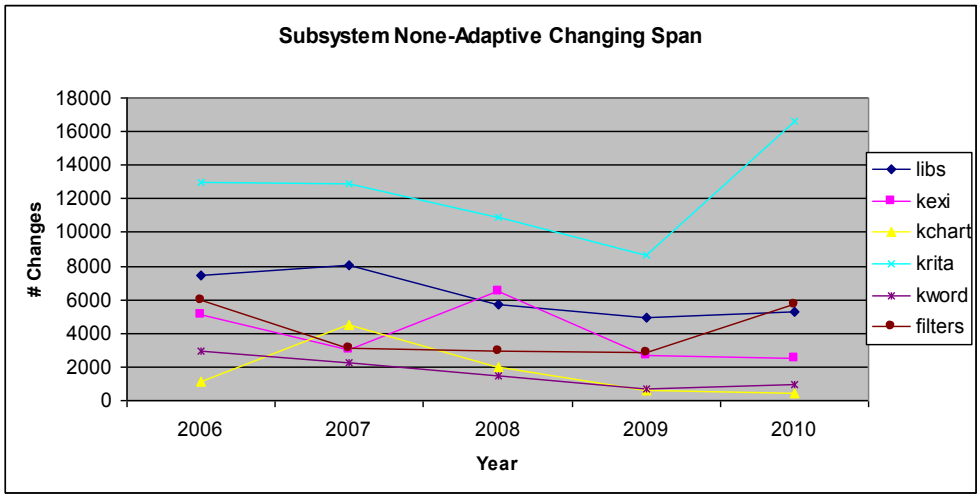
A.3 KOffice Subsystems Changing Span

A.3.1 Adaptive Changing Span

Subsystem	Adaptive Changes per Year				
	2006	2007	2008	2009	2010
Example	2	0	0	0	0
Filters	22	6	0	12	5
Karbon	8	0	0	0	0
Kchart	3	63	0	4	0
Kformula	8	1	0	1	0
Kplato	27	2	1	2	0
Kpresenter	4	1	1	9	1
Krita	77	41	5	44	0
Kspread	24	4	7	4	3
Kugar	4	0	0	0	0
Kword	29	2	1	3	1
Libs	259	32	25	95	3
Plugins	0	0	1	2	1
Shapes	0	11	0	0	0
Kexi	13	371	6	14	3
Kivio	4	0	0	0	0



A.3.2 Non-Adaptive Changing Span



APPENDIX B

ADAPTIVE CHANGES AT SOURCE CODE LEVEL

This appendix gives examples of adaptive changes occurred at source code level, where these changes are regarding the migration from Qt 3 to Qt 4. Then, we will present a classification of these changes for KOffice system, as a case study.

B.1 Approach

The approach to obtain the actual adaptive changes at source code statements is as follows:

- Find the changed files from the *path* dimension of each adaptive commits in the change log file.

Find the differences between the old and new versions of each changed file using UNIX *Diff* utility.

B.2 SVN Diff Command

Diff is a command that is used to display the differences between two versions of a given file. Thus, the command will before and after examples of the changes occurring by a commit. This command represents a UNIX utility. The syntax of this command is as follows:

svn diff -r Ro:Rn file_URL

Where:

– Rn : Commit(revision) number in the change log file

Ro: Rn - 1

B.3 Examples of Adaptive Commits for KOffice

Qt 4 -Class	QTime
Adaptive Changes	*Old:* QTime_var.toString("PThhHmmMssS")) *New:* QTime_var.toString("'PT'hh'H'mm'M'ss'S'"))

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
1183113	1	1	0	0	2	4

Qt 4 -Class	QListWidget
Adaptive Changes	*Old:* QListWidget_var.setSelectionMode (QAbstractItemView::MultiSelection) *New:* QListWidget_var.setSelectionMode (QListWidget::MultiSelection)

Revision Number	Modified Files	Revision Number	Modified Files	Revision Number	Modified Files	Revision Number
1134854	2	1134854	2	1134854	2	1134854

Qt 4 -Class	QUrl
Adaptive Changes	*Old:* openTemplate (const QString & url) *New:* QUrl url1(const QString & url); openTemplate (const QUrl & url1);

Revision Number	Modified Files	Revision Number	Modified Files	Revision Number	Modified Files	Revision Number
546610	3	546610	3	546610	3	546610

Qt 4 -Class	QRect
Adaptive Changes	*Old:* QRect range = QRect_var . normalize () *New:* QRect range = QRect_var . normalized ()

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
542403	9	57	2	0	61	64

Qt 4 -Class	QTextEdit

Adaptive Changes	*Old:*
	multiLine = new Q3MultiLineEdit(page);
	New:
	multiLine = new QTextEdit(page);

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
835248	4	15	0	0	32	66

Qt 4 -Class	QList & QHash
Adaptive Changes	*Old:* Q3ValueVector <KoXmlElement> userStyles; Q3Dict <KoXmlElement> masterPages; *New:* QList <KoXmlElement> userStyles; QHash <QString, KoXmlElement> masterPages;

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
616771	2	3	0	0	15	16

Qt 4 -Class	QRectF & QPointF
Adaptive Changes	*Old:* QRect clipRect (viewConverter()->normalToView (QRect_var).toRect()); *New:* QRectF clipRect(viewConverter()->normalToView (QRectF_var).toRect());

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
545061	8	67	0	0	221	337

Qt 4 -Class	QVariant
Adaptive Changes	*Old:* QVariant Result; Result = var ->….. *New:* QVariant Result; Result = pointer of var ->…..

Revision Number	Modified Files	Modified Methods	Added Methods	Deleted Methods	Hunk Size	Line Size
908255	1	1	0	0	1	1

B.4 Classification of Adaptive Changes

This section introduces a classification for adaptive changes that were occurred at source code level. This classification is based on the problem that is addressed by occurring change. Here, each subsection presents a category that is resulted from such classification.

B.4.1 Method Signature Change

Type	Method Signature Change
Description	Changes number of parameters, parameter types, and return value
Example	***Old:*** QBuffer::QBuffer (QByteArray byteArray) int QTimer::start (int msec, bool sshot) void QBoxLayout::addWidget (QWidget * widget, int stretch = 0, int alignment = 0)
	New: QBuffer::QBuffer (QByteArray * byteArray) void QTimer::start (int msec) void QBoxLayout::addWidget (QWidget * widget, int stretch = 0, Qt::Alignment alignment = 0)

B.4.2 Method Replacement

Type	Method Replacement
Description	Replacing an old class member method by it's equivalent in new platform
Example	***Old:*** bool QIODevice::at (Offset pos) QImage QPixmap::convertToImage () const int QString::find(const QRegExp & rx,int index=0)
	New: bool QIODevice::seek (Offset pos) QImage QPixmap::toImage () const int QString::indexOf(const QRegExp &rx,int index= 0)

B.4.3 Variable Type Change

Type	Variable Type Change
Description	Replacing classes that result in declaration statement changes, method replacement and new method usage, and header changes
Example	***Old:*** QMemArray <QRect> R Q3MemArray <QRect> R QWidgetStack *secondAnd3rdColumnStack
	New: Q3MemArray <QRect> R QVector <QRect> R QStackedWidget *secondAnd3rdColumnStack

B.4.4 Introducing New Class

Type	Introducing New Class
Description	Introducing new classes (Variables) in the new platform, which not exist in previous version.
Example	***Old:***
	New: QFlags QTreeView QModelIndex

B.4.5 Introducing New Global Methods

Type	New Class
Description	Introducing new global methods in the new platform, which not exist in previous version.
Example	***Old:***
	New: const T& qMax (const T &value1, const T & value2) void qSort (Container & container) T qAbs (const T & value)

B.4.6 Enum Value Renaming

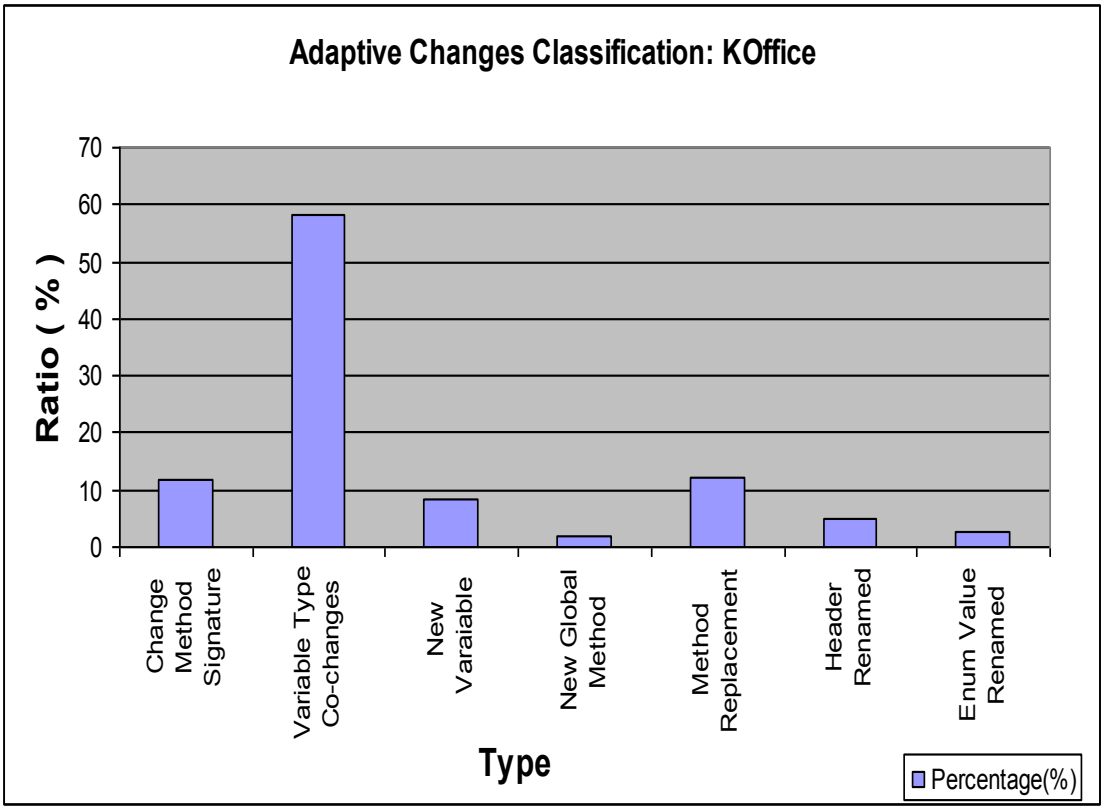
Type	Enum Value Renaming
Description	Renaming enum values in new platform.
Example	***Old:*** Qt::ControlButton Qt::SingleLine
	New: Qt::ControlModifier Qt::TextSingleLine

B.4.7 Header File Renaming

Type	Enum Value Renaming
Description	Renaming the include directives syntax in new platform.
Example	***Old:*** #include <qstringlist.h> #include <qlabel.h>
	New: #include <QStringList> #include <QLabel>

B.4.8 Classification Case Study: KOffice

Here, we categorized the occurring adaptive changes for KOffice system based on above classification. The results are as follow:

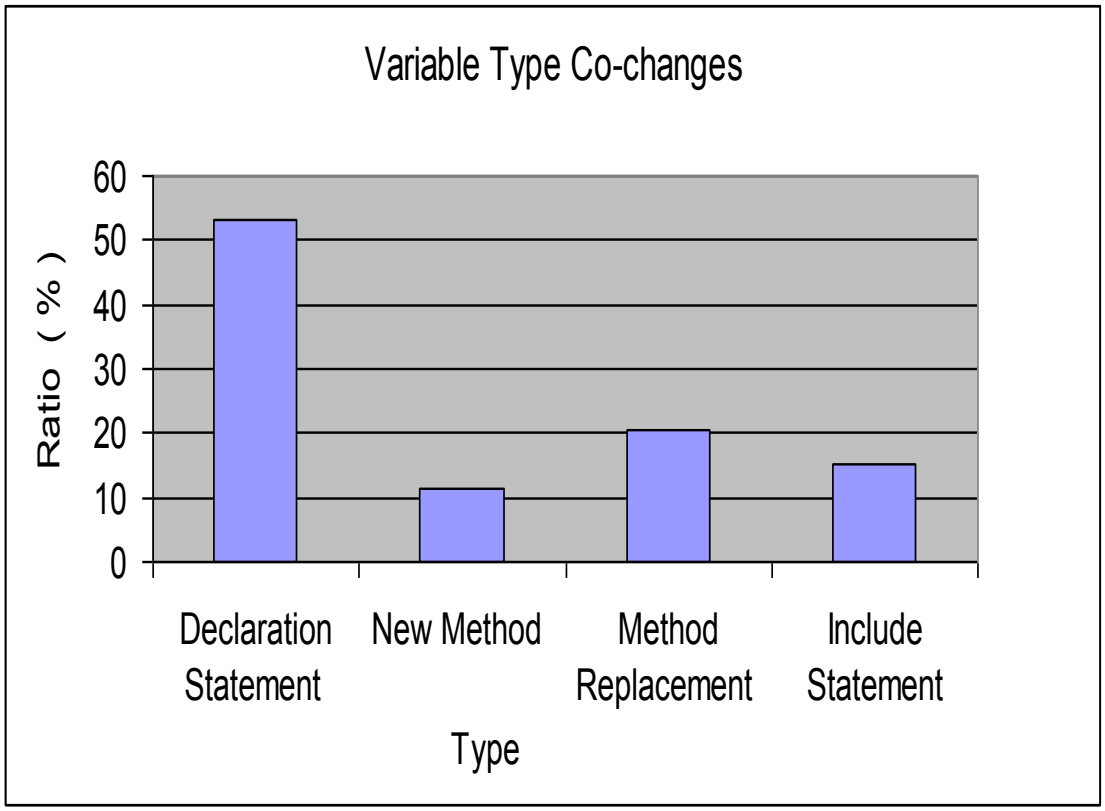


As we can see, the majority of occurring adaptive changes are placed in variable type changes. Now, changing variable type causes the following modifications:

- Declaration Statement Changing
- New Method Introducing
- Old Method Replacement

- Include statement

Therefore, we also classify the variable type changing into several categories based on mentioned effects. The results are shown below.



REFERENCES

[Alali et al. 2008] Alali, A., H. Kagdi and J. I. Maletic, "What's a Typical Commit? A Characterization of Open Source Software Repositories", in Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC), Amsterdam, Netherlands, 2008, pp.182-191.

[Alhindawi et al. 2013] Alhindawi, N., O. Meqdadi, B. Bartman and J. I. Maletic, "A TraceLab-Based Solution for Identifying Traceability Links using LSI", in Proceedings of the 7th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), California, USA, 2013, pp.79-82.

[Anbalagan and Vouk 2009] Anbalagan, P. and M. Vouk, "On predicting the time taken to correct bug reports in open source projects", in Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM), Alberta, Canada, 2009, pp.523-526.

[Andriyevska et al. 2005] Andriyevska, O., N. Dragan, B. Simoes and J. I. Maletic, "Evaluating UML Class Diagram Layout based on Architectural Importance", in Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), Budapest, Hungary, 2005, pp.1-6.

[Antoniol et al. 2002] Antoniol, G., G. Canfora, G. Casazza, A. D. Lucia and E. Merlo, "Recovering Traceability Links between Code and Documentation", Journal of IEEE Transactions of Software Engineering, vol. 28(10), 2002, pp.970-983.

[Bartolomei et al. 2010] Bartolomei, T. T., K. Czarnecki, x. La and R. mmel, "Swing to SWT and back: Patterns for API migration by wrapping", in Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM), NY,USA, 2010, pp.1-10.

[Binkley and Lawrie 2010] Binkley, D. and D. Lawrie, "Information Retrieval Applications in Software Maintenance and Evolution", Encyclopedia of Software Engineering, 2010.

[Buckley et al. 2005] Buckley, J., T. Mens, M. Zenger, A. Rashid and G. Kniesel, "Towards a taxonomy of software change: Research Articles", Journal of Software Maintenance and Evolution, vol. 17(5), 2005, pp.309-332.

[Canfora and Cerulo 2005] Canfora, G. and L. Cerulo, "Impact Analysis by Mining Software and Change Request Repositories", in Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS), Como, Italy, 2005, pp.29 - 38.

[Cleland-Huang et al. 2005] Cleland-Huang, J., R. Settimi, O. BenKhadra, E. Berezhanskaya and S. Christina, "Goal-centric traceability for managing non-functional requirements", in Proceedings of the 27th International Conference on Software Engineering (ICSE), Missouri, USA, 2005, pp.362-371.

[Collard et al. 2010] Collard, M. L., J. I. Maletic and B. P. Robinson, "A lightweight transformational approach to support large scale adaptive changes", in Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM), Timisoara, Romania, 2010, pp.1-10.

[Cossette and Walker 2012] Cossette, B. E. and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries", in Proceedings of the 20th ACM International Symposium on the Foundations of Software Engineering (SIGSOFT) North Carolina, USA, 2012, pp.1-11.

[Cubranic and Murphy 2003] Cubranic, D. and G. C. Murphy, "Hipikat: recommending pertinent software development artifacts", in Proceedings of the 25th International Conference on Software Engineering (ICSE), Oregon , USA, 2003, pp.408-418.

[Czauderna et al. 2011] Czauderna, A., M. Gibiec, G. Leach, Y. Li, Y. Shin, E. Keenan and J. Cleland-Huang, "Traceability challenge : using TraceLab to evaluate the impact of local versus global IDF on trace retrieval", in Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), Hawaii, USA, 2011, pp.75-78.

[De Lucia et al. 2004] De Lucia, A., F. Fasano, R. Oliveto and G. Tortora, "Enhancing an artefact management system with traceability recovery features", in Proceedings 20th IEEE International Conference on Software Maintenance (ICSM), Illinois, USA, 2004, pp.306-315.

[Deerwester et al. 1990] Deerwester, S., S. Dumais, T. Landauer, G. Furnas and R. Harshman, "Indexing by Latent Semantic Analysis", Journal of the American Society of Information Science, vol. 41(6), 1990, pp.391-407.

[Dit et al. 2008] Dit, B., D. Poshyvanyk and A. Marcus, "Measuring the Semantic Similarity of Comments in Bug Reports", in Proceedings of the 1st International Workshop on Semantic Technologies in System Maintenance (STSM), 2008.

[Dit et al. 2012] Dit, B., E. Moritz and D. Poshyvanyk, "A TraceLab-based solution for creating, conducting, and sharing feature location experiments", in Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC), Passau, Germany, 2012, pp.203-208.

[Dragan et al. 2006] Dragan, N., M. L. Collard and J. I. Maletic, "Reverse Engineering Method Stereotypes", in Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM), Pennsylvania, USA, 2006, pp.24-34.

[Dragan et al. 2010] Dragan, N., M. L. Collard and J. I. Maletic, "Automatic identification of class stereotypes", in Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM), Timişoara, Romania, 2010, pp.1-10.

[Dragan et al. 2011] Dragan, N., M. L. Collard, M. Hammad and J. I. Maletic, "Using stereotypes to help characterize commits", in Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), Virginia, USA, 2011, pp.520-523.

[Egyed 2003] Egyed, A., "A Scenario-Driven Approach to Trace Dependency Analysis", Journal of IEEE Transaction of Software Engineering, vol. 29(2), 2003, pp.116-132.

[El-Ramly et al. 2006] El-Ramly, M., R. Eltayeb and H. A. Alla, "An Experiment in Automatic Conversion of Legacy Java Programs to C#", in Proceedings of the 4th IEEE International Conference on.Computer Systems and Applications (AICCSA), Dubai/Sharjah, UAE, 2006, pp.1037-1045.

[Gall et al. 1998] Gall, H., K. Hajek and M. Jazayeri, "Detection of logical coupling based on product release history", in Proceedings of the 14th International Conference on Software Maintenance (ICSM), Bethesda, Maryland, 1998, pp.190-198.

[Gall et al. 2003] Gall, H., M. Jazayeri and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings", in Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE), Helsinki, Finland, 2003, pp.1-13.

[German 2004] German, D. M., "Using software trails to reconstruct the evolution of software: Research Articles", Journal of Software Maintenance and Evolution, vol. 16(6), 2004, pp.367-384.

[German 2004] German, D. M., "Mining CVS repositories, the softchange experience", in Proceedings of the 1st International Workshop on Mining Software Repositories (MSR), Edinburg, UK, 2004, pp.17-21.

[German 2006] German, D. M., "An empirical study of fine-grained software modifications", Journal of Empirical Software Engineering, vol. 11(3), 2006, pp.369-393.

[Gotel and Finkelstein 1994] Gotel, O. C. Z. and C. W. Finkelstein, "An analysis of the requirements traceability problem", in Proceedings of the 1st International

Conference onRequirements Engineering (ICRE), Colorado , USA, 1994, pp.94-101.

[Greevy et al. 2006] Greevy, O., S. Ducasse and T. Girba, "Analyzing software evolution through feature views: Research Articles", Journal of Software Maintenance and Evolution, vol. 18(6), 2006, pp.425-456.

[Hattori and Lanza 2008] Hattori, L. P. and M. Lanza, "On the nature of commits", in Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering Workshops (ASE), Italy 2008, pp.63-71.

[Hayes et al. 2006] Hayes, J. H., A. Dekhtyar and S. K. Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods", Journal of IEEE Transaction of Software Engineering, vol. 32(1), 2006, pp.4-19.

[Hindle et al. 2009] Hindle, A., D. M. German, M. W. Godfrey and R. C. Holt, "Automatic classication of large changes into maintenance categories", in Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC), Vancouver, Canada, 2009, pp.30-39.

[Hinkle et al. 1995] Hinkle, D. E., W. Wiersma and S. G. Jurs, "Applied Statistics for the Behavioral Sciences", Houghton Mifflin, 1995.

[Hsin-yi et al. 2008] Hsin-yi, J., T. N. Nguyen, C. Ing-Xiang, H. Jaygarl and C. K. Chang, "Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management", in Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Italy 2008, pp.59-68.

[Ivkovic and Kontogiannis 2004] Ivkovic, I. and K. Kontogiannis, "Tracing Evolution Changes of Software Artifacts through Model Synchronization", in Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), Illinois, USA, 2004, pp.252-261.

[Kagdi et al. 2007] Kagdi, H., J. I. Maletic and B. Sharif, "Mining Software Repositories for Traceability Links", in Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC), Alberta, Canada, 2007, pp.145-154.

[Kagdi et al. 2008] Kagdi, H., M. Hammad and J. I. Maletic, "Who Can Help Me with this Source Code Change?", in Proceedings 24th IEEE International Conference on Software Maintenance (ICSM), Beijing China, 2008, pp.157-166.

[Keenan et al. 2012] Keenan, E., A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein and D. Hearn, "TraceLab: an experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions", in Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2012, pp.1375-1378.

[Kozlov et al. 2007] Kozlov, D., J. Koskinen, J. Markkula and M. Sakkinen, "Evaluating the Impact of Adaptive Maintenance Process on Open Source Software Quality", in Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement (ESEM), Madrid, Spain, 2007, pp.186-195.

[Kuhn et al. 2007] Kuhn, A., S. Ducasse and T. Girba, "Semantic clustering: Identifying topics in source code", Journal of Information and Software Technology, vol. 49(3), 2007, pp.230-243.

[Kuzniarz et al. 2004] Kuzniarz, L., M. Staron and C. Wohlin, "An empirical study on using stereotypes to improve understanding of UML models", in Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC), Bari, Italy, 2004, pp.14-23.

[Lientz et al. 1978] Lientz, B. P., E. B. Swanson and G. E. Tompkins, "Characteristics of application software maintenance", Journal of Communications of the ACM, vol. 21(6), 1978, pp.466-471.

[Lormans 2007] Lormans, M., "Monitoring Requirements Evolution using Views", in Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR), Amsterdam, Netherlands, 2007, pp.349-352.

[Lovins 1968] Lovins, J., "Development of a Stemming Algorithm", Journal of Mechanical Translation and Computational Linguistics, vol. 11, 1968, pp.22-31.

[Maletic and Reynolds 1994] Maletic, J. I. and R. G. Reynolds, "A tool to support knowledge based software maintenance: the Software Service Bay", in Proceedings of the 6th International Conference on Tools with Artificial Intelligence (ICTAI), New Orleans, USA, 1994, pp.11-17.

[Maletic and Valluri 1999] Maletic, J. I. and N. Valluri, "Automatic Software Clustering via Latent Semantic Analysis", in Proceedings of the 14th IEEE international

conference on Automated software engineering (ASE), Florida, USA, 1999, pp.251-254.

[Maletic and Marcus 2000] Maletic, J. I. and A. Marcus, "Using latent semantic analysis to identify similarities in source code to support program understanding", in Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Vancouver, British Columbia, 2000, pp.46-53.

[Malik and Hassan 2008] Malik, H. and A. E. Hassan, "Supporting software evolution using adaptive change propagation heuristics", in Proceedings of the 24th EEE International Conference on Software Maintenance (ICSM), Beijing, China, 2008, pp.177-186.

[Marco et al. 2006] Marco, L., G. Hans-Gerhard, D. Arie van, S. Rini van and S. Andr, "Monitoring Requirements Coverage using Reconstructed Views: An Industrial Case Study", in Proceedings of the 13th Working Conference on Reverse Engineering (WCRE), Benevento, Italy, 2006, pp.275-284.

[Marcus and Maletic 2003] Marcus, A. and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing", in Proceedings of the 25th International Conference on Software Engineering (ICSE), Portland, Oregon, 2003, pp.125-135.

[Marcus et al. 2004] Marcus, A., A. Sergeyev, V. Rajlich and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of the 11th Working Conference on Reverse Engineering (WCRE), The Netherlands, 2004, pp.214-223.

[Marcus et al. 2005] Marcus, A., V. Rajlich, J. Buchta, M. Petrenko and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code", in Proceedings of the 13th International Workshop on Program Comprehension (ICPC), Missouri, USA, 2005, pp.33-42.

[Masseglia et al. 2005] Masseglia, F., M. Teisseire and P. Poncelet, "Sequential Pattern Mining: A Survey on Issues and Approaches ", Encyclopedia of Data Warehousing and Mining, 2005, pp.3-29.

[McMillan et al. 2009] McMillan, C., D. Poshyvanyk and M. Revelle, "Combining textual and structural analysis of software artifacts for traceability link recovery", in Proceedings of the 5th Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), British Columbia, Canada, 2009, pp.41-48.

[Minto and Murphy 2007] Minto, S. and G. C. Murphy, "Recommending Emergent Teams", in Proceedings of the 4th International Workshop on Mining Software Repositories (MSR), Minneapolis, USA, 2007, pp.5 -14.

[Mockus and Votta 2000] Mockus, A. and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases", in Proceedings of the 16th IEEE International Conference on Software Maintenance (ICSM), California, USA, 2000, pp.120 -130.

[Murphy et al. 1995] Murphy, G. C., D. Notkin and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models", Journal of ACM-SIGSOFT Software Engineering Notes, vol. 20(4), 1995, pp.18-28.

[Nita and Notkin 2010] Nita, M. and D. Notkin, "Using twinning to adapt programs to alternative APIs", in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, 2010, pp.205-214.

[Ostrand et al. 2004] Ostrand, T. J., E. J. Weyuker and R. M. Bell, "Where the bugs are", in Proceedings of the ACM international symposium on Software testing and analysis (SIGSOFT) Massachusetts, USA, 2004, pp.86-96.

[Poshyvanyk et al. 2006] Poshyvanyk, D., A. Marcus, V. Rajlich, Y.-G. Gueheneuc and G. Antoniol, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification", in Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC), Athens, Greece, 2006, pp.137-148.

[Poshyvanyk and Marcus 2007] Poshyvanyk, D. and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC), Alberta, Canada, 2007, pp.37-48.

[Pressman 1986] Pressman, R. S., "Software engineering: a practitioner's approach", McGraw-Hill, 1986.

[Purushothaman and Perry 2005] Purushothaman, R. and D. E. Perry, "Toward understanding the rhetoric of small source code changes", Journal of IEEE Transactions on Software Engineering, vol. 31(6), 2005, pp.511-526.

[Reiss 2002] Reiss, S. P., "Constraining software evolution", in Proceedings of the 18th International Conference on Software Maintenance (ICSM), Montréal, Canada, 2002, pp.162-171.

[Reiss 2006] Reiss, S. P., "Incremental Maintenance of Software Artifacts", Journal of IEEE Transaction of Software Engineering, vol. 32(9), 2006, pp.682-697.

[Revelle and Poshyvanyk 2009] Revelle, M. and D. Poshyvanyk, "An exploratory study on assessing feature location techniques", in Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC), Vancouver, Canada, 2009, pp.218-222.

[Ricca et al. 2006] Ricca, F., M. Di Penta, M. Torchiano, P. Tonella and M. Ceccato, "An empirical study on the usefulness of Conallen's stereotypes inWeb application comprehension", in Proceedings of the 8th IEEE International Symposium on Web Site Evolution (WSE), Pennsylvania, USA, 2006, pp.58-68.

[Robles et al. 2006] Robles, G., J. M. Gonzalez-Barahona, M. Michlmayr and J. J. Amor, "Mining large software compilations over time: another perspective of software evolution", in Proceedings of the 3rd international workshop on Mining software repositories (MSR), Shanghai, China, 2006, pp.3-9.

[Salton and McGill 1983] Salton, G. and M. J. McGill, "Introduction to Modern Information Retrieval. ", McGraw-Hill, 1983.

[Scacchi 2002] Scacchi, W., "Understanding the requirements for developing open source software systems", Journal of IEE Software Proceedings vol. 149(1), 2002, pp.24-39.

[Schach et al. 2003] Schach, S. R., B. Jin, L. Yu, G. Z. Heller and J. Offutt, "Determining the Distribution of Maintenance Categories: Survey versus Measurement", Journal of Empirical Software Engineering, vol. 8, 2003, pp.351-365.

[Sharif and Maletic 2009] Sharif, B. and J. I. Maletic, "The effect of layout on the comprehension of UML class diagrams: A controlled experiment", in Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT) Edmonton, Canada, 2009, pp.11-18.

[Sommerville 2004] Sommerville, I., "Software Engineering (7th Edition)", Pearson Addison Wesley, 2004.

[Spanoudakis et al. 2004] Spanoudakis, G., A. Zisman, E. Pérez-Miñana and P. Krause, "Rule-based generation of requirements traceability relations", Journal of Systems and Software, vol. 72(2), 2004, pp.105-127.

[Spanoudakis and Zisman 2005] Spanoudakis, G. and A. Zisman, "Software Traceability: A Roadmap", Handbook of Software Engineering and Knowledge Engineering, 2005.

[Spohrer et al. 1985] Spohrer, J. C., E. Soloway and E. Pope, "Where the bugs are", Journal of the ACM SIGSOFT Software Engineering Notes, vol. 16(4), 1985, pp.47-53.

[Srikant and Agrawal 1996] Srikant, R. and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", in Proceedings of the 5th International Conference on Extending Database Technology (EDBT), Avignon, France, 1996, pp.3-17.

[Staron et al. 2006] Staron, M., L. Kuzniarz and C. Wohlin, "Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments", Journal of Systems and Software, vol. 79(5), 2006, pp.727-742.

[Sunghun et al. 2008] Sunghun, K., E. J. Whitehead and Z. Yi, "Classifying Software Changes: Clean or Buggy?", Journal of IEEE Transactions on Software Engineering, vol. 34(2), 2008, pp.181-196.

[Swanson 1976] Swanson, E. B., "The dimensions of maintenance", in Proceedings of the 2nd international conference on Software engineering (ICSE), California, USA, 1976, pp.492-497.

[Waddington and Yao 2005] Waddington, D. G. and B. Yao, "High-Fidelity C/C++ Code Transformation", Journal of Electronic Notes in Theoretical Computer Science, vol. 141(4), 2005, pp.35-56.

[Wei Wu et al. 2010] Wei Wu , Y. G. Gueheneuc, G. Antoniol and M. Kim, "AURA: a hybrid approach to identify framework evolution", in Proceedings of the 32nd International Conference on Software Engineering (ICSE), Cape Town, South Africa, 2010, pp.325 - 334.

[Yusuf et al. 2007] Yusuf, S., H. Kagdi and J. I. Maletic, "Assessing the Comprehension of UML Class Diagrams via Eye Tracking", in Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC), Alberta, Canada, 2007, pp.113-122.

[Zaki 2001] Zaki, M., "SPADE: An Efficient Algorithm for Mining Frequent Sequences", Journal of Machine Learning, vol. 42(1-2), 2001, pp.31-60.

[Zhong et al. 2010] Zhong, H., S. Thummalapenta, T. Xie, L. Zhang and Q. Wang, "Mining API mapping for language migration", in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, 2010, pp.195-204.

[Zimmermann et al. 2004] Zimmermann, T., P. Weisgerber, S. Diehl and A. Zeller, "Mining Version Histories to Guide Software Changes", in Proceedings of the 26th International Conference on Software Engineering (ICSE), Edinburgh, UK, 2004, pp.563-572.

[Zimmermann et al. 2005] Zimmermann, T., A. Zeller, P. Weissgerber and S. Diehl, "Mining version histories to guide software changes", Journal of IEEE Transactions on Software Engineering, vol. 31(6), 2005, pp.429-445.