Attaching Social Interactions Surrounding Software Changes to the Release History of an Evolving Software System

by

Olga Baysal

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

©Olga Baysal, 2006

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Open source software is designed, developed and maintained by means of electronic media. These media include discussions on a variety of issues reflecting the evolution of a software system, such as reports on bugs and their fixes, new feature requests, design change, refactoring tasks, test plans, etc. Often this valuable information is simply buried as plain text in the mailing archives.

We believe that email interactions collected prior to a product release are related to its source code modifications, or if they do not immediately correlate to change events of the current release, they might affect changes happening in future revisions.

In this work, we propose a method to reason about the nature of software changes by mining and correlating electronic mailing list archives. Our approach is based on the assumption that developers use meaningful names and their domain knowledge in defining source code identifiers, such as classes and methods. We employ natural language processing techniques to find similarity between source code change history and history of public interactions surrounding these changes. Exact string matching is applied to find a set of common concepts between discussion vocabulary and changed code vocabulary.

We apply our correlation method on two software systems, LSEdit and Apache Ant. The results of these exploratory case studies demonstrate the evidence of similarity between the content of free-form text emails among developers and the actual modifications in the code.

We identify a set of correlation patterns between discussion and changed code vocabularies and discover that some releases referred to as minor should instead fall under the major category. These patterns can be used to give estimations about the type of a change and time needed to implement it.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor, Professor Andrew Malton, who was very helpful throughout my studies. His constant encouragement and support made this work successful. This work was financially supported by the Software Telecommunications Group through the Ontario Research and Development Challenge Fund (ORDCF).

I am also very grateful to my readers Professor Michael Godfrey and Professor Daniel Berry for their time to review this thesis and for their helpful suggestions and detailed comments that have greatly improved this work.

Special thanks to Dr. Ian Davis for proving the base facts to my case study and for many helpful discussions about evolution of LSEdit.

My warmest thanks to all current and former members of the Software Architectural Group, University of Waterloo who have participated in my research study. I greatly appreciate their valuable input, their time and effort. Many thanks to my friend and colleague Xinyi Dong, without whom I won't be able to complete this thesis in time. Thank you for all our fruitful discussions on correlating public discussion with architectural changes, all your valuable technical writing advices and making long working days at the lab more enjoyable.

I am deeply indebted to my family - my мама and папа, sister Елена and brother Виктор. In spite of being located on the other continent for the past few years, they yet gave my tremendous support by inspiring, encouraging and praying for me.

I am very thankful to my new family and friends in Canada. They all have accepted me with warmth and openness. I love you all and will never forget the time and support you gave when I needed them most. I am very lucky and proud to be a member of Kuzucuoglu family.

Finally, I would like to thank a very special person, my husband, for all his support, patience, understanding and unconditional love. He is the one who encouraged we the most in pursuing my Master's and making it all happen by putting aside his own career.

Dedication

To you, my dear parents and my loving husband Timuçin, I dedicate this thesis.

Contents

1 Introduction **1**

1.1 Motivating Idea 2

1.2 Problem Definition 4

1.3 Overview of Proposed Solution 5

1.4 Organization 6

1.5 Contributions 7

2 Background and Related Work **9**

2.1 Software Architecture and Software Change 9

2.1.1 Software Architecture vs. Detailed Design 9

2.1.2 Planning for Change and Reuse 11

2.1.3 Architecture in Practice 13

2.2 Mining Software Repositories to Detect Change Patterns 15

2.3 Recovering Trace Links Between Code and Documentation 18

3 An Approach for Correlating Emails with Source Code Changes **21**

3.1 Change Event–Topic Correlation 22

3.2 Conceptual Similarity Method 24

3.2.1 Building the Discussion Vocabulary 27
3.2.2 Building the Changed Code Vocabulary 29
3.2.3 Comparison of the Vocabularies 32
3.3 Summary 38

4 Empirical Case Studies **39**
4.1 LSEdit Case Study 39
4.1.1 Change Event–Topic Correlation 43
4.1.2 Conceptual Similarity Method 45
4.2 Apache Ant Case Study 51
4.2.1 Conceptual Similarity Method 53
4.3 Discussion 60
4.3.1 Comparison of Case Studies 60
4.3.2 Correlation Patterns 61
4.3.3 Weaknesses of Our Approach 63
4.4 Summary 64

5 Future Work **66**

6 Conclusions **69**

A LSEdit Results **79**

List of Tables

3.1 The output of *diff* represented as a delta 26

4.1 Size of release history and email archives for LSEdit and Apache Ant . . . 40

4.2 Sizes of discussion vocabularies for Ant 53

4.3 Size of changed code vocabulary for Ant 54

4.4 Correlation matrices for regular vocabulary 57

4.5 Correlation matrices for new vocabulary 58

4.6 Correlation matrices for repeated vocabulary 58

4.7 Comparison of the results for LSEdit and Ant 61

4.8 Comparison Table 62

4.9 “Major” minor releases for LSEdit and Ant 63

A.1 Size of discussion vocabularies for LSEdit 79

A.2 Size of changed code vocabularies for LSEdit 83

List of Figures

3.1 A scatter plot depicting change event-topic correlation 23
3.2 Building discussion vocabulary 28
3.3 Finding new topic vocabulary 29
3.4 Finding repeated topic vocabulary 30
3.5 Building changed code vocabulary 31
3.6 Finding correlation between two vocabularies 33
3.7 How discussion vocabulary affects future changes 34
3.8 How many changes were discussed earlier. 34
3.9 Correlation of new topic vocabulary with changed code vocabulary 35
3.10 How new topic affects source code changes. 35
3.11 Correlation between repeated topic and changed code vocabularies 36
3.12 How repeated topic relates to the code changes 36
3.13 Finding maintenance vocabulary 37

4.1 LSEdit Release Distribution 41
4.2 Email Distribution Across Release History of LSEdit 41
4.3 Change event distribution for LSEdit 42
4.4 LSEdit Change-Topic Correlation 44
4.5 LSEdit Change-Message Correlation 44

4.6 Ratio of New and Repeated Topics in Discussion Vocabulary for LSEdit . . 46
4.7 Regular 47
4.8 New 49
4.9 Repeated 49
4.10 Release Distribution of Apache Ant 51
4.11 Email Distribution Across Release History of Apache Ant 52
4.12 Ratio of New and Repeated Topics in Discussion Vocabulary for Ant . . . 55
4.13 Results for Ant 56

Chapter 1

Introduction

The architecture of a software system, however that term be defined [15, 23, 34], must provide terms of reference, a common mental model [20], for discussion of feature implementation, design change, change impact, and migration tasks. A well-documented architectural view is a critical element of the development of a software system, and because of the wide range of stakeholder concerns throughout a long and large software development history, much work in software architecture has concentrated on documenting multiple architectural views [7, 23, 41].

Architectural reverse engineering is the semi-automatic recovery of architectural views from available artifacts, especially source code. Existing techniques of architectural reverse engineering [29, 31, 33, 41] to discover the evolution architecture of an existing software system are based on the mining source code control repositories.

1.1 Motivating Idea

Imagine that there were a tool that could store a record of all public interactions preceding each release of a software system and collected during its development process. At any time, the tool could suggest to a developer what would be the amount and the type of changes in the upcoming version, and show the location of the code where the next modifications will occur. This intelligent application would essentially predict the future behavior of software changes based on the size and length of the current discussion, number and role of its participants, and most importantly, the issues that were brought up by the participants.

Thus, developers armed with such a powerful tool, would spend less time managing changes at the architectural level, the maintenance task that might become very costly as this type of changes affect larger parts of the system and thus, they are more expensive to implement.

Although the idea of developing such an application sounds very promising, the current research in the area of distinguishing architectural changes leaves much to be desired. Therefore, this thesis is aimed at providing a possible path forward for designing techniques and approaches to monitor, plan, and predict software changes.

Regardless of the software system and the development process, there is always a lot of useful information produced during that process. For example, the interactions and communications among developers can be a useful source of information about the software. In fact, communications by means of electronic mail is the only possible way for the developers working on an open source project, to interact with each other remotely. An open source product is designed, developed and maintained through community cooperation. Participants of an open source culture modify the product and redistribute it to the community [36, 52]. These interactive communities contribute to open source project

through electronic media. Therefore, these media consist of the discussions on a variety of issues surrounding the evolution of the open source software product such as reports on bugs and their fixes, new feature requests, design change, refactoring tasks, test plans, etc. Even end users are able to contribute to the open source project by writing a problem report or a request for a new functionality and submitting it electronically.

Most of the time this information is lost as developers ignore the enormous amount of mailing listing archives that can be used to understand the nature of the changes. Our hypothesis is based on the belief by Nedstam *et al.* [32] that "an architectural change does not only need to be technically sound, but it also needs to be anchored firmly in the organization". Architectural changes can originate from various sources but they are always initiated by the architects, developers and managers. Thus, we believe that *electronic media surrounding the evolution of a software system can be used to make recommendations about the nature of the changes that are likely to happen next.*

Most software system documentation such as design documents, user manuals, maintenance journals, consists of free-text documents [1]. Such documents contain free-form natural language text and carry valuable information about the application domain. For example, user manuals include technical regulations. Free text is often used to explain the content of instructions, for example, comments in the source code, or to make understanding easier, for example, user manuals provide assistance to non-technical readers.

Public interactions consist of free-text documents expressed in a natural language and are conducted between software developers, architects, project managers, users, etc. This data is collected from mailing listings, forums, bug reports, new feature requests, and so on during the evolution process of a system.

Discovering a correlation between the free-text email archives associated with the development and maintenance cycle of a software system and its source code can be helpful

in number of ways:

- Impact Analysis

  Nowadays, virtual teams, whose members work from different locations and time zones, are very popular [46]. Developers of such teams often communicate through electronic mails by submitting requests for new features, bug reports or even exchanging ideas about possible architectural enhancements of a system. Thus, any suggested change is first described in an email message and will later propagate to the source code. Establishing the correlation between the content of the mail and the source code will help developers to identify the chunks of code affected by the proposed change [2].

- Maintenance

  As mentioned earlier, maintenance cost can be reduced by monitoring interactions among developers and foreseeing the severity of upcoming changes.

- Guidance for Software Development

  Techniques to monitor and predict software changes assist developers managing these changes and give guidance for software development.

1.2 Problem Definition

Discovering and understanding non-trivial relations between different artifacts of a software system is certainly interesting and important in overall understanding of the system. Our research idea originates from real life observation. When people tend to discuss, intensively and widely, different issues about a phenomenon, the phenomenon will undergo some significant changes in the future.

Unfortunately, the archives of public communications surrounding software changes, are often left behind and the knowledge contained in this resource is simply forgotten or disregarded.

Thus, our research question was therefore refined to: how can we make use of human interactions surrounding the evolution of a system in order to detect and predict architectural changes of the system? What is the correlation, if any, between the content of the email archives and the actual modifications of the source code?

In this work we present a novel approach of making recommendations about software changes during the evolution of a system by making use of the collected mailing list archives.

1.3 Overview of Proposed Solution

Unlike other reverse engineering techniques, finding correlation between free-text discussions and source code components cannot be done using compiler techniques due to the difficulty of applying syntactic analysis to natural language sentences.

In our approach we used the techniques of the Natural Language Processing(NLP). Such techniques have been suggested to benefit research in reverse engineering [22]. Despite their success in many areas, NLP methods are little used in software engineering. Similar to Biggerstaff [5] and Antoniol [1], our assumption is that developers use meaningful names in programs for the classes, methods, functions, types, and variables. These names of program items are mapped to the content of emails in order to find common concepts—words that are common for the vocabularies of the emails and source code. For this work, the vocabulary of a document is the set of words appearing in the document.

For each released version of a software system, we generated two vocabularies:

1. vocabulary of the changed code;

2. vocabulary of the discussion surrounding the changes.

To compute a correlation model, we compare each discussion vocabulary of a certain release against the vocabulary of actual code changes for the same release and for the whole collection of the following releases in the release history.

A high score indicates a high probability that a particular list of concepts discussed prior to a release is relevant to the actual code modifications of that release. We interpret concept similarity as an indication of the existence of correlation between the two artifacts, mailing list archives and release history. Later, the behavior of the calculated correlation is analyzed to find the patterns of correlation between the two artifacts. Detecting correlation patterns can be used in predicting future software changes.

1.4 Organization

Chapter 2 presents related and background work including the existing techniques on identifying software changes from the source code, or using data mining methods to make recommendations on change prediction based on the past events occurred in the system. A brief overview of the current work in applying AI techniques in software engineering such as finding dependencies between documentation and source code is given in this chapter as well.

Chapter 3 describes the novel approach of correlating the history of public discussions about a particular software system with the system's source code change history in order to understand the nature of software changes and perhaps to forecast future modifications. We described the process of developing correlation model using NLP techniques to find relation links between free-form text messages and structured statements of the source code. We speculate that communications recorded prior to a particular product release,

may not correlate to this release's change events, but might affect the changes happening in future revisions.

Chapter 4 provides case studies and the results that were obtained after applying our approach on two open source software systems, LSEdit [40] and Apache Ant [14]. We then compare the results of two case studies, reveal a list of correlation patterns and discuss the weaknesses of our approach.

Chapter 5 discusses possible directions of future work in the area of identifying and predicting software changes.

Finally, Chapter 6 outlines the contributions of our work.

1.5 Contributions

Our primary goal has been to create a method of identifying architectural changes. With this goal in mind, we discovered latent, information-rich relations between the source code changes and the public communications involving those changes.

The major contribution of this thesis is the proposal of a novel approach to make recommendations about the essence of software changes by mining both the release history and mailing list archives. Our approach is based on correlating the vocabulary extracted from e-mail messages with the vocabulary of source code changes.

We have contributed to the understanding of the nature of software changes by attaching social interactions surrounding those changes during the development process of a system.

The empirical contribution of the thesis is the application of proposed approach on two open source projects LSEdit and Apache Ant. The case studies demonstrate that our method can be used in understanding the architecture of a large software system.

Furthermore, we believe that by monitoring social interactions during the development process, in particularly newly introduced concepts, we could tell about the nature that is architectural or non-architectural, and the location of future changes.

Chapter 2

Background and Related Work

This chapter provides an overview of the existing research in different areas related to our study. We begin by discussing the usefulness and importance of software architecture, providing the foundation for our research. We provide the summary of the data mining approaches to detect change patterns, as well as the techniques used to recover traceability links between different artifacts.

2.1 Software Architecture and Software Change

We start the discussion with describing our understanding of the notion of software architecture, its change and importance during the lifecycle of a system.

2.1.1 Software Architecture vs. Detailed Design

Software architecture is a growing but relatively young discipline, thus, there is no single, precise definition.

Several definitions of software architecture can be found in research literature.

According to Garlan and Shaw [15], software architecture "involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns".

Perry and Wolf [34] defined architecture as a set of architectural elements that have a particular form, explicated by a set of rationales. The term "architecture" is often used as a synonym for design of a system [13], but it is not what we mean under software architecture.

A well received definition of software architecture was suggested by Bass *et al.* is "the structure of the system which comprises software elements, the externally visible properties of those elements, and the relationships among them" [4]. This definition intentionally does not specify what the elements of the system and relationships among them are. For example, it could be an object, a database, a graphical user interface, a library or anything. Architecture is more of an abstraction of a system that hides details of the elements that do not affect the way how they relate to or interact with other elements.

Architecture is concerned with the selection of architectural elements, their interactions to provide a framework for the design. Design is concerned with detailed interfaces of the design elements, their algorithms and data types needed to support the architecture [34].

There, in practice, the difference between architecture and design lies in the degree of an abstraction. Software architecture provides the overall picture of the system's components and their interactions at the highest level of abstraction, where only main subsystems and their dependencies are revealed. At the design level this picture is more detailed. Each subsystem is broken down into smaller modules and relationships among these modules are exposed.

For the purpose of software reusability, architectural reuse is more beneficial, as it does not involve details about how elements of a system communicate with each other.

Elements interact with each other by means of interfaces, which carry domain specific details. Architecture is not interested in these low level, non-architectural details, therefore architectural reuse is more desirable as it can be applied at a large-scale.

2.1.2 Planning for Change and Reuse

When designing an architecture for a product or a family of products, it is crucial to build structures that will allow a system to reach its quality goals. One of the main goals is to prepare software for change [3]. Planning for change results in reduction of maintenance cost and increase in quality attributes.

When preparing software for change, an architect will try to predict possible changes and design the architecture in such a way that all elements engaged in the change will be put together in a single component. Software quality metrics, like cohesion and coupling, can also assist architect with managing future changes. Architectural components with high cohesion and low coupling increase probability that future changes will affect only a single component [4]. Architecture should support design that any scheduled change would require minimum effort to implement it and thus, make a system easier to evolve and maintain.

Development of new software systems from reusable blocks is possibly easier and faster than building them from the scratch. Two decades of previous research in software engineering was dedicated in finding techniques to support software reuse, component interactions, domain-specific architectures, etc.

Earlier reuse research was focused on reuse of code-level entities like classes, subroutines and data structures. But software reusability is not limited to the reuse of a source code only. In fact, various artifacts of a software lifecycle, including designs, test data and documentation, can be reused. According to Jones [21], there are ten potentially reusable

kinds of artifacts of software projects such as:

- source code,
- architectures,
- data,
- designs,
- documentation,
- estimates or templates,
- human interfaces,
- plans,
- requirements,
- test cases.

Research literature offers various definitions, metrics, taxonomies, and reusability systems to assist with reusability issues on different artifacts.

Reuse at the architectural level is more common when developing a family of products [8]. A family of products includes software systems that share common features and parts. Most successful software organizations today plan product lines rather than a single product. Therefore, software architecture needs to address product family concerns. Architecture designed for the entire family becomes the main asset of the company. These architectures provide design decisions that can apply across the family and can apply to individual systems, providing variation in their features and capabilities.

Multiple releases of a single system support enhancements on the functional level or improve quality attributes of a system. They can also be considered as a product family. Transferable abstractions [4] that are the architectures that can be moved from one release to another, are most beneficial if they are exploited earlier in the lifecycle. Such abstractions will ensure gradual evolution and easy maintenance of a software system.

At the end, product line architectures always show big payoffs in cost, time and product quality [4].

2.1.3 Architecture in Practice

There are many reasons to study software architecture, including:

- Early design decisions

  It is the most difficult to get the right architectural design at the fist stage of a development process and it is the hardest and most costly to change it later. The right decisions about architecture form the ground for the successful path in the system development, its deployment and maintenance.

- Assistance with reasoning about and managing changes

  It is well know [24], that around 80 percent of software cost occur after its deployment. Most work of developers fall into the maintenance phase. Every software system changes, sometimes with difficulty. Architectural changes are most difficult to implement. They affect the way elements interact with each other in a system, and most probably they will require changes all over the system. Non-architectural changes are more desirable as they are easier to implement. Thus, assessing proposed changes and their risks require good understanding of the current design.

- Systematic reuse

  Software architectures can be applied to the systems with similar functional requirement and quality attributes. Architectural decisions can be reused across multiple systems, in other words, at large scale.

- Communication vehicle for stakeholders

  This reason addresses the major importance of the architecture. Each stakeholder of a software system - user, developer, tester, project manager, architect - expects different characteristics from a software systems that are all affected by the architecture. For example, users want more reliable and secure systems, while customers concern with the cost of a project and its rapid delivery.

According to Holt, software architecture is best thought of as a mental model that servers to facilitate development and understanding of a system and communication between its stakeholders [20]. Architecture provides different stakeholders the means and the basis for communication, discussions, negotiations and understanding.

Since a software architecture provides a mental model for discussions on design change, feature implementation, migration tasks, the question is, "Is the reverse true?" Can we use communication during development process to reason about software architecture and its changes?

Our hypothesis is that *interactions* between stakeholders can certainly be used to reason about software changes, give recommendations about their nature and may be even predict changes to happen next.

2.2 Mining Software Repositories to Detect Change Patterns

Most maintenance tasks focus on managing all sorts of changes happening to the system. It is important that software maintainers understand structural and architectural evolution that the system has experienced in the past.

Data mining is an automated extraction of hidden information from a large data set. It usually involves searching for patterns in large volumes of information. Data mining, sometimes called market basket analysis [50], is used often to recover valuable information from the resources initially collected for other intents and purposes. Hand described data mining as "the process of secondary analysis of large data aimed at finding unsuspected relationships which are of interest or value to the database owners" [18].

Software artifacts are normally stored in and managed from software repositories. For example, source file versions are located in source code repositories, email messages are kept in mailing archives, problem reports and feature requests are recorded in bug-tracking systems, design documents are included in project documentation. Analysis of these repositories of information can certainly benefit to a developer. Hence, data mining techniques can be applied to these repositories in order to retrieve useful information about latent dependencies between various types of artifacts.

Considerable amount of research has been done in the area of data mining where historical data was analyzed to learn about the nature of software changes [30, 42, 55], to find change patterns [46, 47, 56] and even to predict future changes [19, 39].

Detecting structural changes and change patterns is recognized by the research community as one of the most difficult task in architectural reverse engineering.

Godfrey and Tu [42] investigated a way to detect and model structural changes such as

moving and renaming, by performing origin analysis [41]. Origin analysis is used to reason about where, how and why the design changes have occurred in the system. The approach is based on a detailed analysis of call relations and entities at the function level. Beagle tool [42] was implemented to support origin analysis of a structural evolution of a software system. Later Zou and Godfrey [16] has extended the Beagle tool with the techniques of applying origin analysis to detect function and file merges and splits that have occurred from one version to another. They demonstrated that locating merges and splits is helpful in discovering some of the original context of the design changes.

Wu investigated the punctuated evolution [55]. He observed that software architecture mainly changes during the punctuation periods that are the periods of sudden and discontinuous change. Punctuated evolution can be determined by analyzing changes to structural dependencies at the file level and functional growth in number of files. He measured file level dependency change based on either incoming or outgoing dependencies. Based on these metrics, he presented the development history of a software system using technique of evolution spectrograph, a color-coded graph. This tool can be used to highlight major change events across historical sequence of software releases.

Zimmermann *et al.* [47] presented a data mining approach over Concurrent Versions System(CVS) repositories to recommend source code that is relevant to a given source code fragment. Their approach is based on the association rules to identify changes, detect coupling fine-grained entities and to predict future or missing changes.

Ying *et al.* [56] suggested to use market basket analysis techniques to assist developer with identifying relevant source code during modification task. They determined change patterns, sets of files that were changed together frequently in the past, by applying data mining techniques on the historical data of the source code. Their approach consists of three stages. First, they extracted useful information that is what files were checked in

together, from the software configuration management (SCM) system. Then they applied frequent pattern mining algorithm to find change patterns from the source code. Each change pattern consists of the names of the source files that have been changed together frequently in the past. These change patterns are used to give recommendations on the files relevant to a particular change task by providing a name of the file that is more likely to be involved in this task. In contrast to Zimmermann, Ying can suggest only files, not finer-grained entities like functions or variables.

Mockus *et al.* [30] studied a large legacy system to test the hypothesis that a textual description of a change retrieved from the historic version control data can be used to determine the purpose of software changes and to understand and diagnose the state of a software project. They discovered four types of changes: adding new functionality that is adaptive, fixing faults, defined as corrective, restructuring code to accommodate future changes as a perfective type, and code inspection changes that involve both corrective and perfective changes. Their classification of changes showed the strong relations between size and type of maintenance task and the time required to make a change.

Shirabad *et al.* [39] used machine learning techniques to extract models from the past experience that can be used in future predictions. They showed that data obtained from software update record can be used to find relations between files to predict whether change in one file may require the change in another file. Their experiments concluded that combining text based features with syntactic attributes from source code and problem reports improves the results. In our approach we also combined text-based such as comments, and syntactic attributes such as class names and method names, of a source code while building the vocabulary vector of the change code from one version to the next.

Hipikat [46] is a tool that gives recommendation about the project information a developer should consider during a modification task. This project information is formed

from a number of different artifacts, including source code versions, change tasks reports, newsgroup messages, email messages and documentation. When presenting recommendation to a developer, relationships links are used to determine relevant artifacts to the task being performed. Similar to Hipikat, we tried to relate the source code release history with the email messages collected during the development process. However, while Hipikat makes recommendation about similar change completed in the past, we aim providing recommendation about future changes.

Hassan and Holt [19] used historical source control systems to predict change propagation. Change propagation is used to determine how changes made in a particular file, will effect modifications in other files, called co-change files. Co-change files are those that need to be modified as soon as new feature or bug fix take place in a system. When new changes happen to the file, other files need to be modified at the same time to keep the system updated and consistent with these new changes. They presented some heuristics for change propagation, as well as the approach to study various change propagation models [19].

Our approach involves discovering non-trivial relationships between artifacts of different types such as source code changes and email discussion surrounding those changes. We applied data mining techniques on the release history of a software system and email archives collected during the lifetime of this system for the purpose of recovering useful information about the correlation between these resources.

2.3 Recovering Trace Links Between Code and Documentation

Several researchers have investigated relationships between software artifacts. Existing literature [1,25,28] has shown that Information Retrieval (IR) methods can be successfully

used in recovering trace links between software artifacts of different types. These methods are based on finding textual similarity between the artifacts. The main assumption behind this is the fact that most of the software documentation is text based and that developers use meaningful source code identifiers.

Antoniol *et al*. [1] have proposed a semi-automatic approach for recovering trace links between free-text documentation such as manual pages and functional requirements, and source code classes. They used IR techniques to rank the documents against the query consisting of the source code identifiers. Two IR models, probabilistic and vector space model, were applied and experimentally evaluated in two case studies. They tried to map domain concepts found in documentation to code fragments by applying exact string matching algorithm. The results of these studies were assessed by IR metrics, precision and recall. Both models showed promising results. However vector space model will be a better choice in the case of the smaller size of the software engineering documentation.

Marcus and Maletic [28] used Latent Semantic Indexing (LSI), an extension of the vector space model, that searches for concepts rather than searching for terms. They applied their model on the same case studies [1] and compared the performances of LSI with the vector space and probabilistic models. Their results showed a very good performance of applying LSI model without the need for morphological analysis such as stemming, of the terms which was essential for the vector space and probabilistic models to reach similar results.

De Lucia *et al*. [25] used the LSI model also for trace link recovery to deal with any type of software artifacts, including requirement and design artifacts, test cases and code classes.

Our approach also aims at finding relations between two artifacts, mailing list archives and source code. However, we did not apply some predefined IR models, for example, VSM or LSI, but rather we use NLP methods in text analysis to identify correlation

corpus between code and electronic interactions among developers. In order to determine correlation between two artifacts, they need to have common concepts in their vocabularies. Thus, correlation process is only possible if both artifacts are written in the same natural language, for example, English. Human communications are mainly represented as natural language texts, while source code can hardly be defined as English prose. Therefore, our main assumption is that developers use natural language in writing source code as well as their domain knowledge in defining source code identifers.

Chapter 3

An Approach for Correlating Email Interactions with Source Code Changes

Although free-form natural language text documentation was found to be useful to recover trace links [1], it is unknown whether communications between stakeholders, represented in a form of free-text documents, can help to reason about the nature of changes. IR methods deal with text categorization problems by determining a set of documents relevant to user query. Since we are not interested in retrieving email messages to match a particular user query and to rank retrieved data, but rather in discovering correlation between electronic documentations and changes in the source code, we had to design an algorithm that is lightweight and able to compare the data of these two artifacts.

Our method to find correlation between source code changes and natural language communications is based on textual similarly between a discussion vocabulary and a changed code vocabulary. Thus, our premise is that developers use their domain knowledge to

give meaningful names to their identifiers of program entities and are consistent with their usage.

This chapter characterizes the input data, describes two correlation approaches and gives details for each step of the methods.

3.1 Change Event–Topic Correlation

Since the evidence of correlation between source code changes and email interactions is unknown, our first approach aimed at finding out whether there is a potential association between source code and email discussions.

A scatter plot was used to determine how discussions around the changes relate to the actual modifications in the source code.

A scatter plot, also known as a scatter diagram or a scatter graph, is a graph used in statistics to visually display and compare two or more sets of data by displaying points, each having a coordinate on a horizontal and a vertical axis [53]. The idea is that two data sets are lined up along X and Y axes, and dots are used to indicate the presence of similarity between two sets.

First data set represents the number of topics that are threads, extracted from the content of the discussions. The second data is obtained from the source code and portrays the number of change events occurred between two sequential releases. We used class-level granularity to measure change events by counting the number of classes that have been changed such as deleted, added or modified. Both data sets are then transferred to the scatter plot, in which "number of topics" is assigned to the vertical axis and "number of changes" to the horizontal axis. A single dot on the scatter plot, for example at point (20,10), would represent a release with twenty source code changes and whose discussion

consists of ten topics. The scatter plot representing all releases would demonstrate a visual comparison of the two sets of data and be used to conclude what kind of relationship exists between them.

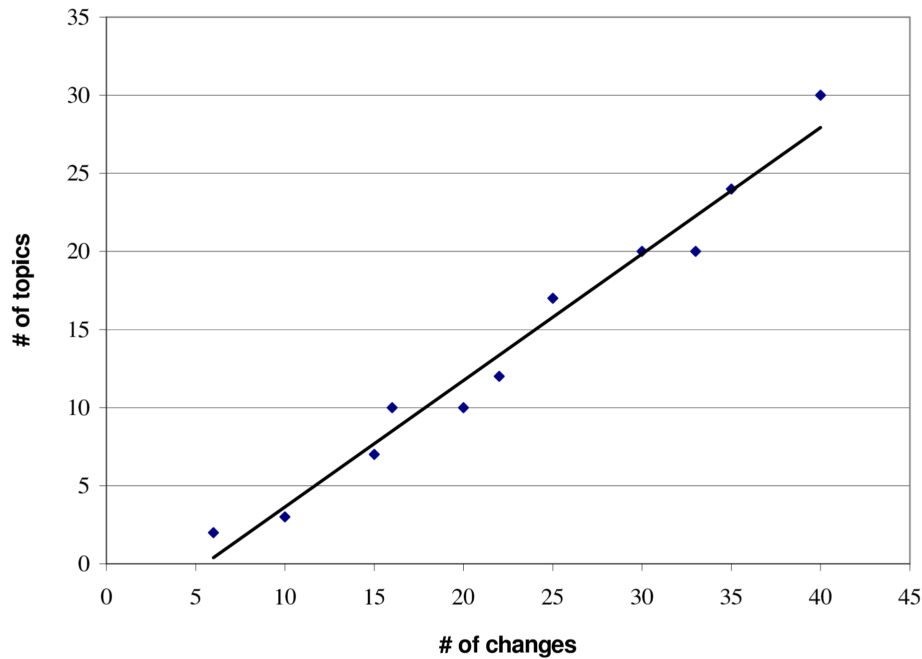

Figure 3.1: A scatter plot depicting change event-topic correlation

A positive that is rising, trendline in a scatter plot will represent a high similarity, while the negative relationship will be indicated by a falling diagonal, showing a low degree of similarity. Linear trendline is automatically added to a chart, as we used MS Excel application to chart data. The trendline shows a slope of data points and can be calculated using the following formula:

$$y = m * x + b, \tag{3.1}$$

where m = SLOPE(y,x) and b = INTERCEPT(y,x).

Since we believe that email discussions and code changes are correlated, we expect to see many dots along the rising diagonal indicating a high degree of similarity between them.

Figure 3.1 shows an example of the expected relationship between discussion topics and modifications in the code. This scatter plot demonstrates a positive linear relationship between two data sets—more topics engaged in the discussion, in other words, longer discussions, correspond to larger amount of changes found in the system, and few topics that are shorter discussions, lead to less changes.

Thus, scatter plots are helpful in detecting the degree of correlation such as strong or weak, between two variables and in indicating how much one variable is affected by another. However, they are not able to suggest what is the similarity between two variables and how one affects another. Consider the case, when most of the interactions contain various bug reports. Obviously, changes affected by this sort of discussions would simply be bug fixes and nothing else. And on the contrary, bigger changes are not necessary affected by big debates, the system might undergo some planned, not spontaneous or immediate, changes, for example, restructuring tasks.

In fact, the results of our case study, shown in Figure 4.4, reveal this kind of evidence confirming the weakness of this approach. Chapter 4.1 discusses these findings in details.

3.2 Conceptual Similarity Method

Conceptual similarity method uses the terms extracted from the discussions and identifiers extracted from the code to find correlation between natural langauge electronic communications and source code changes. According to Definitions 1 and 2, these terms form a discussion vocabulary, and the identifiers form a changed code vocabulary. Later these

vocabularies are compared in order to determine their common concepts - terms that appear in both vocabularies. A term becomes a concept if it is defined in the code as well as represented in human communications. This set of common concepts represents the correlation between these vocabularies.

The input data consists of the archives of electronic mails and release history of a software. Both the source code and the mailing archives need to be decomposed into the proper granularity to define the document, which later will be used to construct vocabularies.

As a general practice in IR, when dealing with natural language, a paragraph or section is used as the granularity of a document. Since the content of email messages is usually smaller than a section or a paragraph of a document, obviously message granularity is too small. Similar, in source code the concepts like function, module, class or file can define granularity level. For example, Maletic used functions in procedural code [26] and class definitions in OO code [27].

In our work, we aim at mapping electronic interactions to the source code changes. Therefore, each file is treated as a textual document. This allows us to compare corresponding files across release history and to compute a difference, defined as delta, representing source code changes between two sequential releases.

A *release history* of a software system, denoted by R, is a set of versions deployed during the development process of the system. R = $\{r_1, r_2, ..., r_k\}$, where k is the total number of released versions, k = |R|.

A mailing archive consists of a large amount of email messages. Each email message can refer to different structures of the source code like a function, a method or a class and so on. As we are interested in matching code modifications of a complete release with the electronic discussions that caused them, email messages have to be organized to form a discussion of that release.

```
171c4
< * Copyright (c) 1999 The Apache Software Foundation. All rights
– – –
> * Copyright (c) 2001 The Apache Software Foundation. All rights
153a688,690
> public boolean execute() throws BuildException {
> attributes.log("Using classic compiler", Project.MSGVERBOSE);
> Commandline cmd = setupJavacCommand();
```

Table 3.1: The output of *diff* represented as a delta

Since the message granularity is too small to be used in our work, we define a discussion document as follows: a *discussion document* d is a set of email messages originated between two sequential releases. Discussion document d_i consists of all the email interactions that occur between release r_i and its preceding release r_{i-1}. The email interactions prior to the first release are not considered because the first release is the starting point for identifying changes and its preceding discussion is omitted.

Hence, a *discussion corpus* $\bar{D}$ is defined as a set of discussion documents $\bar{D} = \{d_1, d_2, ..., d_n\}$. The total number of documents in the corpus is n = $|\bar{D}|$ = k-1. Therefore, to form a discussion corpus, email messages are arranged into documents, one document for each release to allow the linking between release discussion and version modifications.

Each source file is considered as a textual document. To be able to relate a discussion document with the source code, the content of all the files that represent a software system, is joined together into one document.

A *source code document* denoted by c, is a set of all the source files for a single release. Therefore, c_i represents a source code document for release i.

We used fine grained analysis of release repositories to recover the history of source code modifications indicated by lines that have been added, deleted and changed during the evolution of a source file. Each release represented by the source file document is compared to its predecessor by running Unix utility *diff*. Table 3.2 shows the output of a diff command performed on two source code documents. In the example, the line 171 of the first document was changed to line 4 in the second document. And at line 153 three new lines were added. The result of this comparison is stored in *deltas*. Each delta contains a line by line difference, such as added, deleted and changed lines, between two source code documents, thus $\Delta_i = c_i$ - c_{i-1}. Therefore, a *corpus of code changes* $\bar{C}$ is a set of deltas $\bar{C}$ = $\{\Delta_1, \Delta_2, ..., \Delta_m\}$. The total number of deltas in the corpus is m = $|\bar{C}|$ = k-1, k $\in$ R.

The process of finding correlation between the discussion document and source code changes embedded in delta consists of the following steps:

1. Building the discussion vocabulary by extracting terms from the discussion document.

2. Building the changed code vocabulary by extracting identifiers from delta.

3. Comparing discussion and changed code vocabularies.

Let us explain each step in detail.

3.2.1 Building the Discussion Vocabulary

Definition 1 *A discussion vocabulary D, also referred as regular discussion vocabulary, is a set of terms extracted from a discussion document, $D \leq d$.*

Figure 3.2 shows the process of constructing a discussion vocabulary for each document in the discussion corpus.

This stage is performed in five steps:

1. First, each attachment or email header containing a date and time of a message, a subject, the name of an author, a recipient is removed as it does not carry information.

2. In the second step, each number or punctuation, such as a comma, period, quotation mark, bracket, hyphen, is eliminated.

3. In the third step, each capital letter is transformed into its lower case letter.

4. The next step includes sorting and duplicate removal.

5. Finally, a list of stop words [44] is applied to eliminate most common English words that are articles, prepositions, etc.

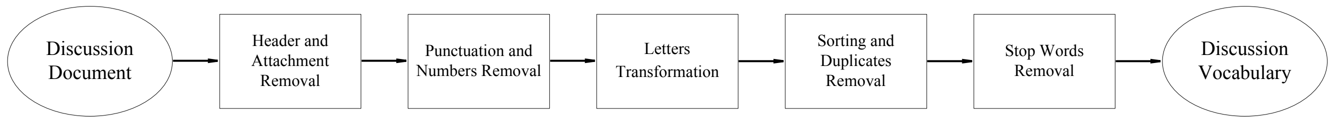

Figure 3.2: Building discussion vocabulary

Analyzing discussion documents, we introduce a few more types of vocabulary that we expect carry valuable information.

New topic vocabulary is denoted as N and calculated as a relative complement of the discussion vocabulary of a preceded release in the discussion vocabulary of a current release:

$$N_i = D_i - D_{i-1}. \tag{3.2}$$

New topic identifies all the new words that appear in the release discussion but not in the previous release.

Figure 3.3 displays the method of obtaining new topic vocabulary. The boxes represent discussion documents D_1 and D_2. New topic vocabulary is shown as the dotted area $N_2 = D_2$ - D_1.

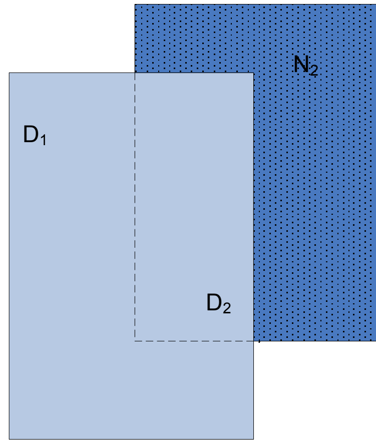

Figure 3.3: Finding new topic vocabulary

Repeated topic vocabulary described as P, consists of all the terms found in both discussion vocabulary of the current release and discussion vocabulary of the previous release. It is defined as:

$$P_i = D_i \bigcap D_{i-1}. \tag{3.3}$$

Repeated topic defines all the common words that discussion documents share with each other.

In Figure 3.4 a repeated topic vocabulary is demonstrated as the intersection between the boxes filled with horizontal lines.

3.2.2 Building the Changed Code Vocabulary

Definition 2 *A changed code vocabulary or simply change vocabulary C is a set of identifiers extracted from a delta, $C \leq \Delta$. Changed vocabulary C_i contains all the identifier names obtained from a Δ_i.*

Domain knowledge and concepts are embedded in the source code through identifier

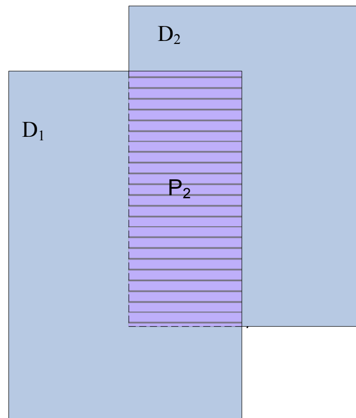

Figure 3.4: Finding repeated topic vocabulary

names and comments. Identifiers are textual tokens that name program entities, such as variables, types, classes, functions, methods, etc. Comments are used in the code mainly to explain developers' intentions about a certain function or an algorithm [48]. They are particularly important in open source projects when the code is shared between many developers who may never have met. Comments provide a better understanding and guidance throughout the code.

Therefore, we use identifier names and comments to map the source code to human communication.

Figure 3.5 demonstrates the process of building changed code vocabulary:

1. Identifier extraction separates the names of identifiers such as classes, methods, and comments, from the rest of the source code.

2. Identifier separation splits identifiers into two or more simple words, for example, a class name DataInputStream would be split into three separate words - Data, Input and Stream. Identifier separation enriches the corpus and improves the results for the reason that separated identifiers are closer in form to natural language words

used in communication.

3. Numbers and punctuation, including special symbols like #, %, $ etc., removal purges all non-alphabetic symbols.

4. Letters transformation changes capital letters into lower case ones.

5. Sorting and duplicate removal gets rid of repeated words and groups remaining words alphabetically.

6. Stop words removal eliminates useless words from the vocabulary.

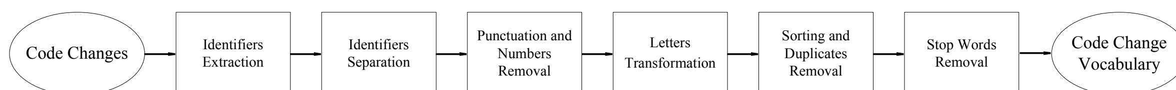

Figure 3.5: Building changed code vocabulary

The process of building discussion vocabulary as shown in Figure 3.2, slightly differs from the process of building discussion vocabulary that is presented in Figure 3.5. Both processes include text normalization activities, such as: punctuation and numbers removal, letters transformation, sorting and removal of duplicates and stop words. However, when dealing with discussion document we consider all the words as terms that build a discussion vocabulary, but for each code document we are interested in extracting only specific identifiers to build a changed code vocabulary. Identifiers [51] are names assigned to the program entities like variables, types, classes and so on. We use such identifiers to refer to the higher-level concepts found in the electronic discussions, making correlation process possible.

3.2.3 Comparison of the Vocabularies

The final stage of the approach deals with comparing two generated vocabularies. In order to compare discussion vocabulary with the changed code vocabulary, we first need to define a measure of correlation, also called as similarity or association, between two vocabularies.

IR offers various measures of similarity between two documents [45]. The simplest definition is presented as:

$$|\, X \bigcap Y \,| \tag{3.4}$$

where X and Y are the documents.

We adopted this definition (3.4) to state the measure of correlation between two vocabularies.

A *correlation* between discussion vocabulary and changed code vocabulary is a set of terms that two vocabularies have in common, also called *common concepts*, corr(D,C) = D $\bigcap$ C.

Thus, the evidence of correlation or association is based on the number of common concepts, in other words, on the presence or absence of terms in the vocabularies.

Figure 3.6 illustrates the process of comparing two vocabularies and determining their *common concepts* - terms that appear in both vocabularies. A box shape represents a discussion vocabulary D of some release i and a circle represents a changed code vocabulary C of a release j. A set of common concepts or correlation s_{ij} between vocabularies D_i and C_j is the intersection of two shapes shown as a pattern-filled area.

We next present an enhancement of a correlation measure, given in Definition 3.2.3, for two vocabularies. Following this, we propose two new correlation measures as:

$$corr_D(D,C) = \frac{|\, D \bigcap C \,|}{|\, D \,|}, 0 \leq corr_D(D,C) \leq 1 \tag{3.5}$$

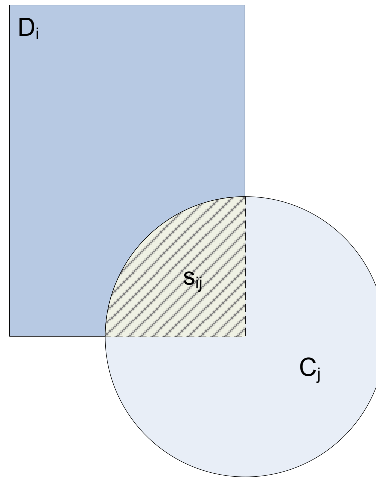

Figure 3.6: Finding correlation between two vocabularies

and

$$corr_C(D, C) = \frac{\mid D \bigcap C \mid}{\mid C \mid}, 0 \leq corr_C(D, C) \leq 1 \qquad (3.6)$$

where $corr_D$ is *correlation with respect to D* and $corr_C$ is *correlation with respect to C.*

The first measures how D relates to the C, while the second measures how C has been faced in D. In other words, correlation $corr_D$ corresponds to how much of the source code changes are discussed by stakeholders. While the second one $corr_C$ determines how much of the discussed issues are actually found in changes.

To find out how a discussion might affect future changes that is $corr_C$, is demonstrated in Figure 3.7. Here, the strength of correlation between discussion vocabulary D_1 and change vocabularies C_1, C_2 and C_3 respectively, varies. The larger area of the intersection between the box and the circle, the stronger correlation between the vocabularies. The highest degree of correlation is shown between discussion vocabulary D_1 and changed code vocabulary C_3.

Figure 3.8 displays $corr_D$, indicating how much of a change in a release was actually

mentioned in earlier discussions. Such findings can prove that older email interactions affect newer releases if the degree of correlation between newer change code vocabulary and older discussion vocabulary is higher than the one between the former one and newer discussion vocabulary. In the example, the correlation between C_3 and D_2, denoted as s_{23} is higher than the correlation between C_3 and D_3 expressed by s_{33}.

Next, we examine how new topic relates to the source code changes. In Figure 3.9, correlation s_{22} shows that a new topic affects more than one third of all the modifications in the source code.

Figure 3.10 displays correlation of a new topic vocabulary N_2 between two discussions with changed code vocabularies C_2 and C_3. A set of common concepts s_{22} is larger than a set s_{23}, revealing a stronger association of new topic vocabulary with changed code

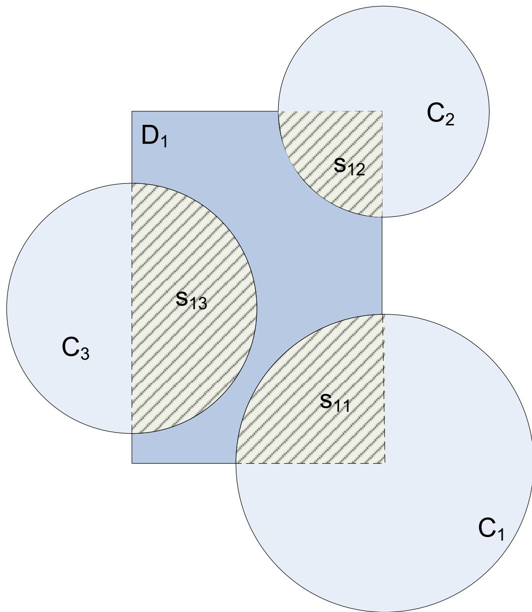

Figure 3.7: How discussion vocabulary affects future changes

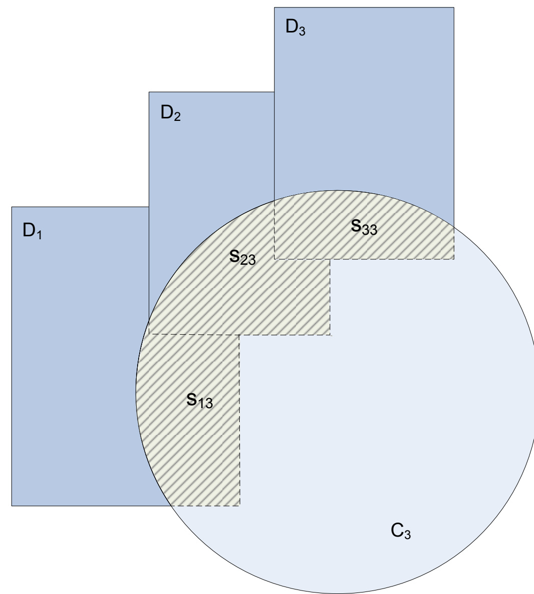

Figure 3.8: How many changes were discussed earlier.

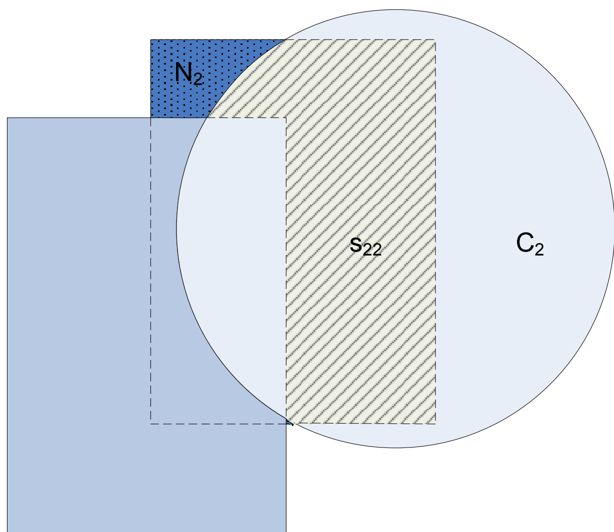

Figure 3.9: Correlation of new topic vocabulary with changed code vocabulary

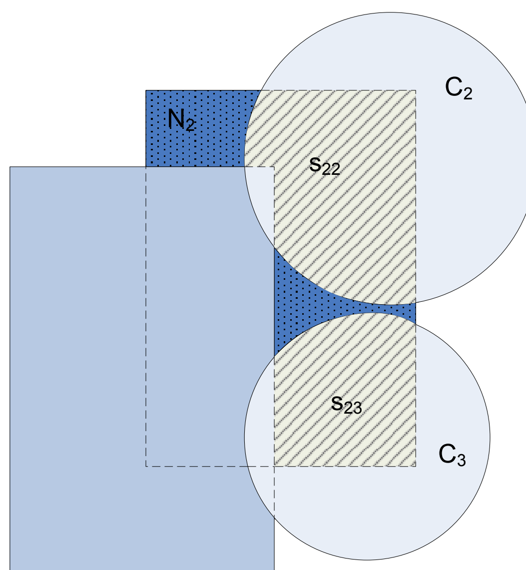

Figure 3.10: How new topic affects source code changes.

vocabulary C_2.

To determine a correlation between the repeated topic vocabulary and the changed code vocabulary, shown in Figure 3.11, we identify a set of terms that are present in both vocabularies $s22$. This type of correlation shows whether words that are repeated from one discussion to another reflect the changes in the source code.

Figure 3.12 illustrates a correlation of repeated topic vocabulary P_2 with the changed code vocabularies C_2 and C_3 respectively.

We calculate correlation values between changed code vocabularies and discussion vocabularies of different types such as regular, new topic, repeated topic, for the complete release history of a system. To store the generated data we use matrices.

A correlation matrix [49] is computed to indicate the strength of the relationships between discussion vocabularies and changed code vocabularies for the complete release

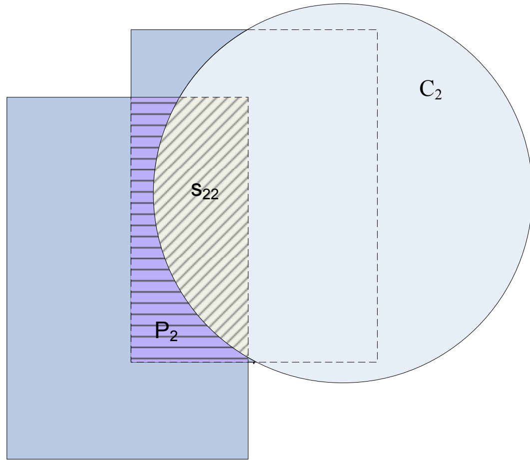

Figure 3.11: Correlation between repeated topic and changed code vocabularies

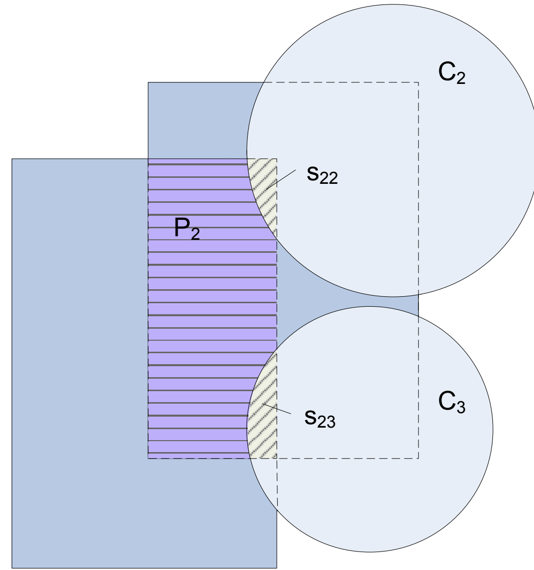

Figure 3.12: How repeated topic relates to the code changes

history of a system.

Definition 3 *A correlation matrix is a $k \times k$ matrix $S = (s_{ij})$, where (s_{ij}) is corr(D_i,C_j). A correlation matrix S is an upper or lower triangular matrix, which is shown in 3.7, where entries below or above, for lower triangle, the main diagonal are zeros $s_{ij} = 0$ if $i > j$:*

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & . & . & s_{1k} \\ 0 & s_{22} & s_{23} & & & s_{2k} \\ 0 & 0 & s_{33} & & & . \\ . & & & . & & . \\ . & & & & . & . \\ 0 & . & . & . & 0 & s_{kk} \end{pmatrix} \tag{3.7}$$

Zero values are explained by the fact that every correlation matrix has a mirror-image quality above or below the diagonal, where the correlation between release i and release j

is always equal to the correlation between release j and release i. Thus, there is no need to show both triangles.

To locate the correlation for any pair of vocabularies, find the value for the row (discussion) and a column (changed code) for those two vocabularies.

Several correlation matrices are generated. Each matrix is characterized by the correlation measure from Definition 3.2.3, (3.6) and (3.5) and the type of a discussion vocabulary used at the comparison stage.

Analyzing discussion vocabularies, we can define what are the most discussed issues that were addressed during the evolution of a system, called maintenance vocabulary. *Maintenance vocabulary* is built by determining the terms that are shared across complete release history. Figure 3.13 depicts the process of finding maintenance vocabulary across several releases by revealing their common terms. Maintenance vocabulary denoted as M, is shown as an intersection of the three boxes representing discussion vocabularies D_1, D_2 and D_3.

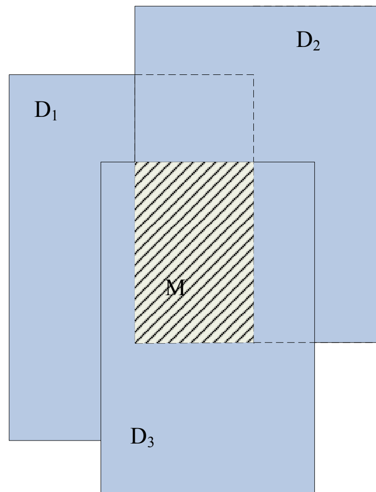

Figure 3.13: Finding maintenance vocabulary

3.3 Summary

This chapter explains two approaches for correlating mailing list discussions surrounding the change with the actual modifications of the source code.

The first approach compares two sets of data by representing their relevancy using scatter graph. The association is calculated based on correlating the quantity of topics or the mail count with the quantity of change events occurred in the code.

The second approach is based on determining the conceptual similarity between two vocabularies, extracted from the artifacts, by finding concepts that are shared between those vocabularies.

Chapter 4

Empirical Case Studies

Chapter 3 presents two approaches to correlate source code changes with communications by electronic media. Chapter 4 reports our application of the proposed correlation methods in two case studies. The goal is to assess how well our approach of correlating software changes with the email interactions among stakeholders performs on systems with different characteristics. The studies systems have different sizes of both the release history and email interactions. Table 4.1 summarizes the details of the systems. The results of the experiments are compared and analyzed to identify correlation patterns.

4.1 LSEdit Case Study

The first case study was a freely available graph visualization tool, called LSEdit [40], developed by the Software Architecture Group at the University of Waterloo. LSEdit (the name stands for Landscape Editor) is a tool used in reverse engineering to display and explore graphs representing software architecture. LSEdit is a Java-based system. Its size has grown from 137 files in release 6.0.1 up to 144 files in release 7.1.25, and LSEdit still

continues to evolve. Over the three and half years, the number of files remained almost the same, while the number of classes has increased significantly from 137 classes to 348 classes.

The goal of this case study is to determine the correlation between the source code modifications recorded in the release history and human communications among developers, users, and a programmer.

System	Number of Releases	Number of Email Messages
LSEdit	118	495
Apache Ant	16	67377

Table 4.1: Size of release history and mailing list archives for LSEdit and Apache Ant

Table 4.1 indicates the size of the release history and the number of collected electronic mails for two systems. For LSEdit, we examined only 91 sequential released versions starting from release 6.0.1 to release 7.1.25. Figure 4.1 illustrates that releases are not evenly distributed over time. For example, after one and half years there was only one version released. It was a huge rewrite of LSEdit to work under Swing framework.

Email archives were formed by collecting electronic mails from the current and former members of the Software Architecture Group lab. The collected data included emails from the main developer of a system, three professors who were actively involved in the development process of LSEdit, around twenty graduate students, and a number of other people from the academic environment participated in the past discussions about LSEdit. Since we were interested only in the interactions that surrounded the changes of a system, we manually analyzed the data removing all the irrelevant emails which have nothing to do with the maintenance tasks on LSEdit. After eliminating irrelevant emails, the discussion corpus for this case study contained only 495 emails. Collected emails were

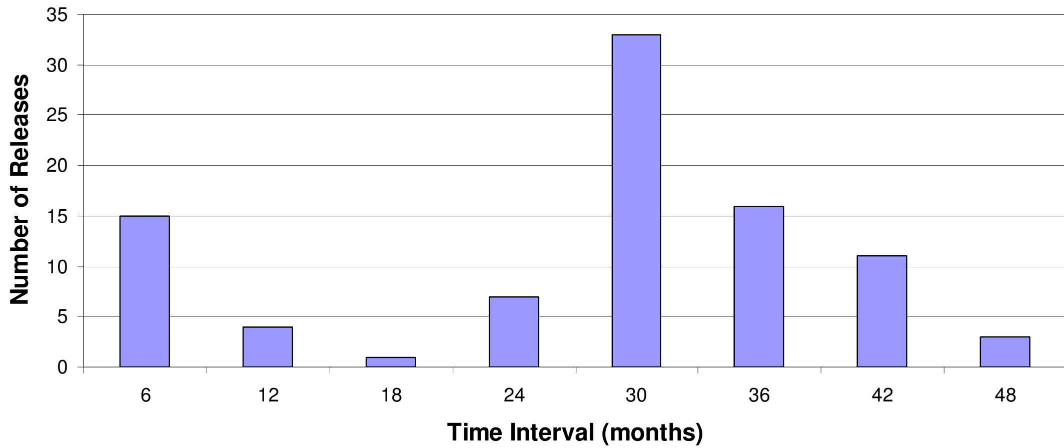

Figure 4.1: LSEdit Release Distribution

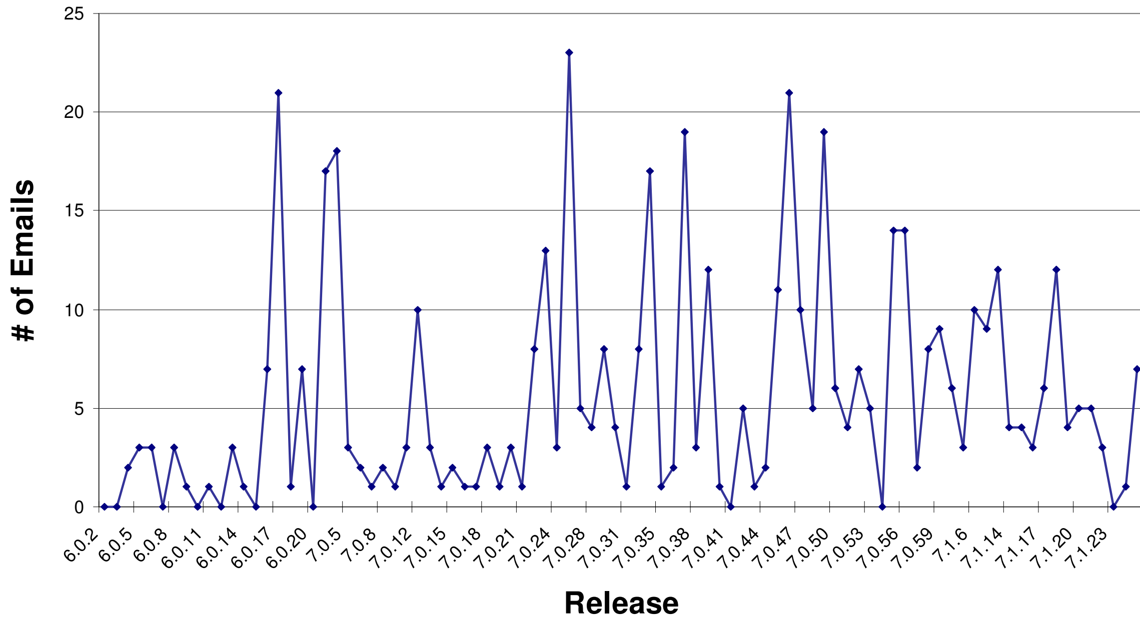

Figure 4.2: Email Distribution Across Release History of LSEdit

grouped according to the time frame they originated in. Time frame is defined by the date when a release was deployed. Hence, each of the 90 releases had a corresponding discussion

document consisting of the emails originated during the development period of that release. The first release is the starting point, and its corresponding discussion document contains no messages. For this reason, the first release is not included in the correlation process.

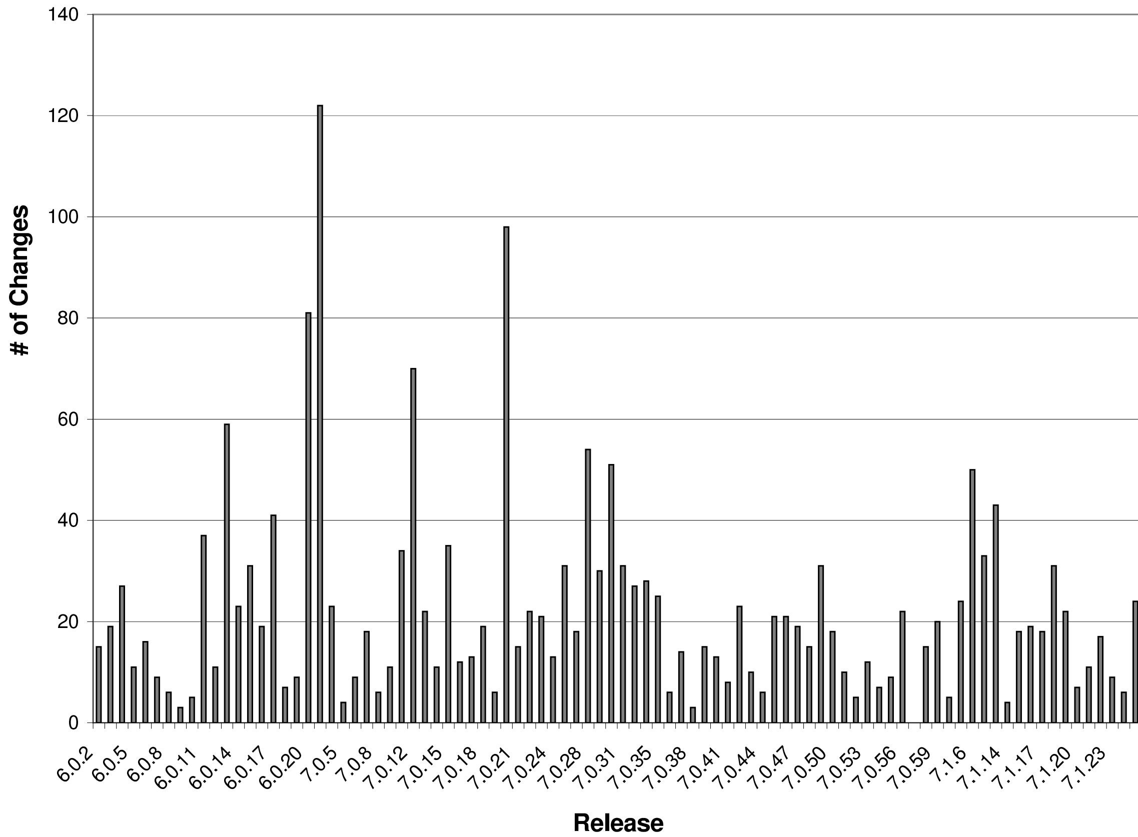

Figure 4.3: Change event distribution for LSEdit

Email distribution across release history of LSEdit is shown in Figure 4.2. Each point on the curve represents a single release. The vertical scale defines the size of the discussion document of a certain release or in order words, the number of emails that corresponds to that release. The figure shows that nine releases of LSEdit have a zero-sized discussion document. The largest discussion document consists of 23 emails and belongs to the release

7.0.25.

Figure 4.3 shows change distribution for all releases of LSEdit. To recover change event information we used class-level granularity. For object-oriented systems of small or medium size, a class is a very convenient unit to measure changes [6]. The total number of changes includes the number of added, deleted and modified classes. As in Figure 4.2, each bar delineates a release and a corresponding value of changes is indicated on the vertical axis.

Overall, LSEdit is a small-size system with a very poor discussion corpus but rich release history.

4.1.1 Change Event–Topic Correlation

The results of our change event–topic correlation are shown in Figure 4.4. There are 90 dots in the scatter plot, each of which corresponds to a single release. The rising trendline, bold and black line, indicates a positive correlation between number of topics and quantity of changes. However, it is obvious that there is no strong correlation among these two variables, because most of the dots are not distributed along the trendline as we would expect. On the contrary, we observe that dots are distributed in two directions. The two red lines show the actual trend lines in this scatter plot.

The presence of two trend lines can be interpreted as follows: the first trendline that is above the black trendline, defines the correlation between large number of topics with a small number of changes. This describes that modifications were originated from the user side, which explains the small amount of changes as users usually report bugs, and theses small changes represent bug fixes. The second trendline, shown below the main diagonal, demonstrates a lot of changes with the lack of discussions, which reveals that these changes might have been initiated from the developer of a system.

We also made an attempt to associate changes with the number of messages rather

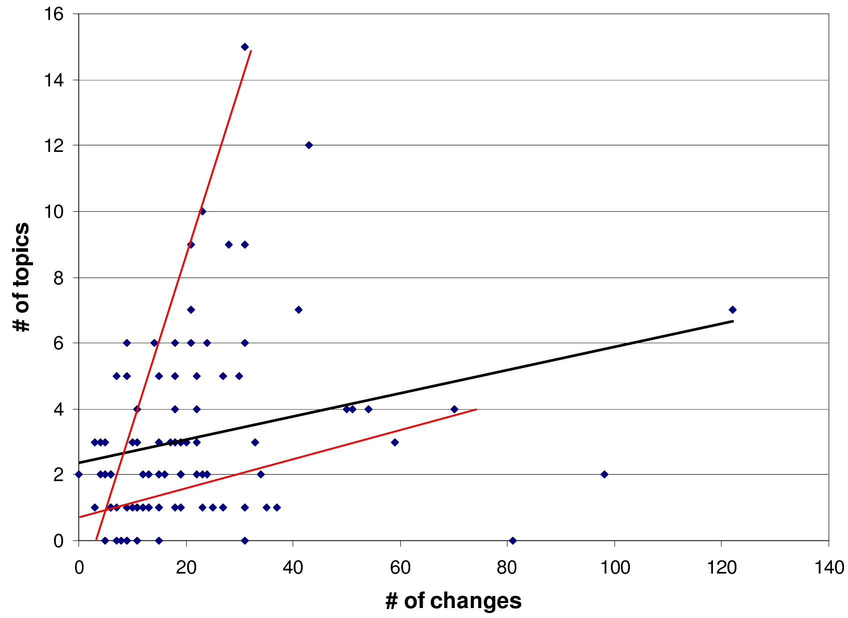

Figure 4.4: LSEdit Change-Topic Correlation

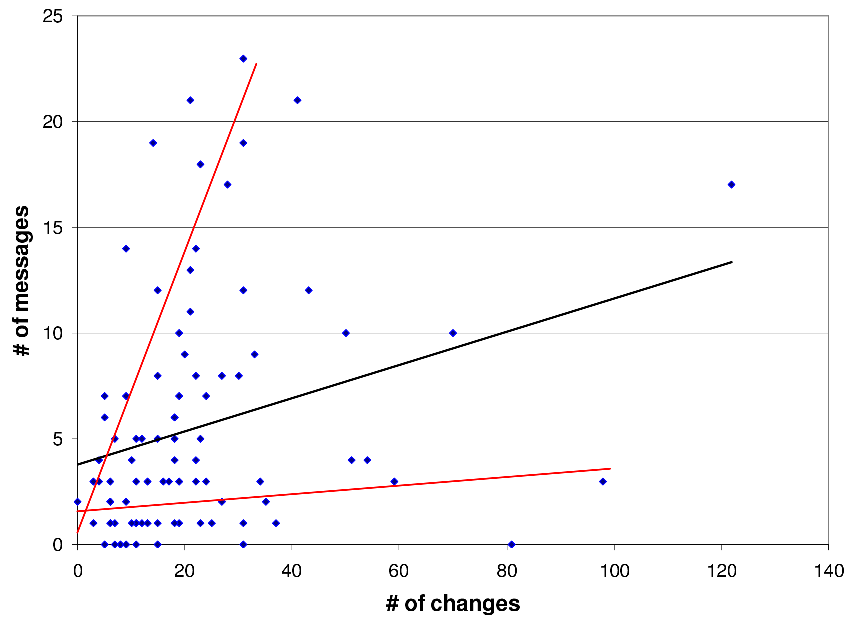

Figure 4.5: LSEdit Change-Message Correlation

than the number of topics. The results, shown in Figure 4.5, are similar to those depicted in Figure 4.4.

Both scatter plots demonstrated the weakness of this correlation method: comparing the quantity of topics or messages with the number of changes does not necessarily confirm the evidence of correlation between discussions and code changes. For this reason, we decided not to apply this method on the other software system.

4.1.2 Conceptual Similarity Method

Due to poor email communication during the development process of LSEdit, the size of its discussion vocabulary for each release is quite small, as illustrated in Table A.1. It varies from a minimum of 0 to a maximum of 969 terms per vocabulary. An average discussion vocabulary contains about 252 terms. Zero-sized vocabularies are quite common for the LSEdit case study, because such a large number of versions were released during the three and half year time interval. Figure 4.6 displays the the size of the discussion vocabulary for 26 out of 90 releases. It also depicts the content of the discussions with respect to the word quality, new words or repeated ones. The lower part of a bar represents new topic vocabulary, while the upper part represents repeated topic.

Figure 4.6 demonstrates that discussion vocabulary for each release mostly contains new terms. This can be explained by the fact that the size of a typical discussion of LSEdit is very small. Thus, emails contain interactions about the issues on new functionality, rather than on various problems and their fixes which would result in repetition of the same words.

Building changed code vocabulary, we decided to use class names, method names and comments embedded in the source code to ensure the generation of a rich enough vocabulary. Hence, source code identifiers were generated with the scripts that extract class

names, method names and comments.

Identifier separation process was concerned with the characteristics of a domain. When developers assign names to their program identifiers, they use their domain knowledge and expertise. Coding style of a developer also affects name selection. For OO domain such as Java in our case studies, there are two commonly used conventions for naming identifiers: one is using underscore "_" as a separate between several words, for example, text_box, and the other one is using letter capitalization for word separation, for example, textBox, TextBox, TextBOX. All identifiers that follow either of these rules are separated into basic words, for our examples text and box, text and Box, Text and Box, Text and BOX respectively.

Table A.2 displays the size of the changed code vocabulary that ranges from 0 to 1797 keywords per vocabulary. The average vocabulary for LSEdit consists of 229 terms. The number is not large, but neither is LSEdit, being a small-size system.

Figures 4.7, 4.8 and 4.9 show the results we obtained in correlating different types of

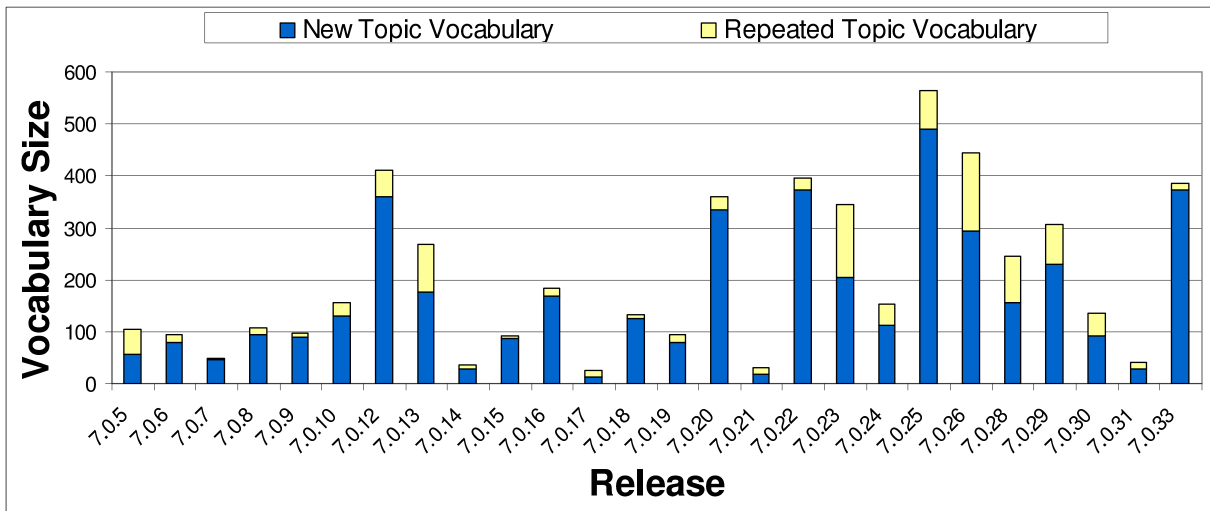

Figure 4.6: Ratio of New and Repeated Topics in Discussion Vocabulary for LSEdit

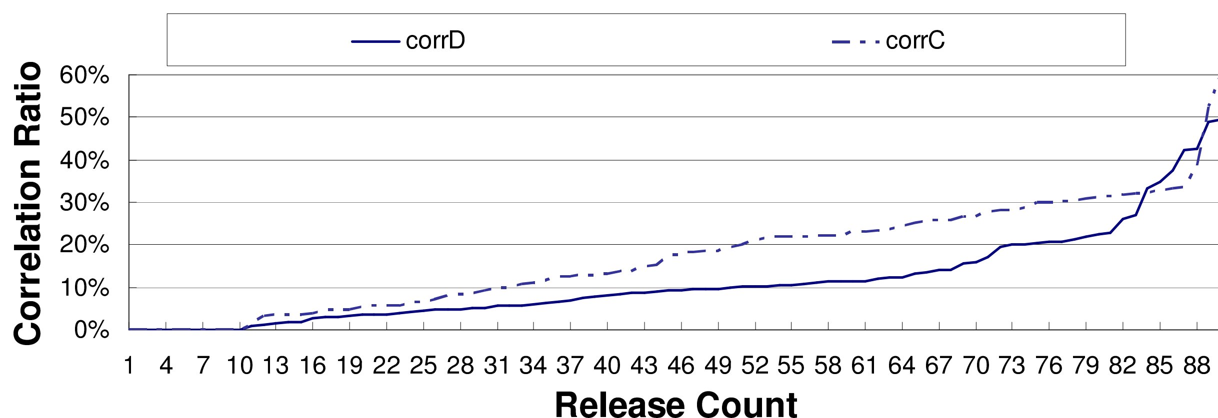

Figure 4.7: Regular

discussion vocabulary such as regular, new and repeated, with changed code vocabulary for LSEdit. In each figure a solid line depicts correlation with respect to discussion that is $corr_D$, and a dashed line shows correlation with respect to the changes that is $corr_C$. Figure 4.7 shows the comparison of $corr_D$ and $corr_C$ for the regular discussion vocabulary and corresponding changed vocabulary of the same release. The $corr_C$ values are slightly better than $corr_D$. We can see that 10 out of 90 releases have 0% correlation ratio, which was expected because several releases have an empty discussion vocabulary. This vocabulary type is quite common for LSEdit due to the fact that some releases were issued on the same or the very next day. Almost half of the total number of releases have correlation ratio $corr_C$ of over 20%, which means that in every other release at least one fifth of discussed issues is actually implemented. And only one release has a correlation value of over 50%. For the prediction purposes, we would be more interested in $corr_D$, indicating if discussion is able to anticipate the changes. The results are not very satisfying. For most releases it is typical that communication occurred prior to a release deployment, affects the modifications in that release with probability of less than 30%. In only two releases, this probability goes above 50%. The conclusion is that the regular type of discussion

vocabulary is not very useful in detecting big changes.

Let us take a look at the new topic vocabulary and its correlation with the code changes. Figure 4.8 shows the obtained results. The results are close to the ones in Figure 4.7. The number of releases with zero correlation between their discussion and code vocabularies is even larger, 15 releases. The discussion vocabularies of these 15 releases simply do not contain any new words or words that are peculiar only to the current release in regard to the discussion vocabulary of the preceding release. Without considering the first 15 releases, the outliers, 40% of all releases have their discussion correlated to the changes with correlation ratio over 10% for $corr_D$ and 62% for $corr_C$. Only 5% of the release history has $corr_D$ value above than 40%.

Figure 4.9 demonstrates the results of finding similarity between changed code vocabularies and repeated topic of discussion. The result for this type of discussion differ from those shown in two previous figures. The curve, representing $corr_D$, is higher $corr_C$ curve. However, the behavior of the $corr_C$ is very similar to the one described in Figure 4.8. There are more of the releases with zero correlation coefficient for both $corr_D$ and $corr_C$. Twenty five of the total number of releases did not show any similarity, because several releases had a zero-size changed code vocabulary, which is similar to the previous $corr_C$ curves. The main explanation of a bigger number of releases with 0% correlation coefficient is that common concepts between code changes and repeated topics are quite infrequent due to the fact most LSEdit discussions contain new topic terms rather then repeated ones, as shown in Figure 4.6.

As mentioned earlier, we are more interested in the $corr_D$, because it reveals how discussions affect changes. If we don't count zero-correlation releases, 78% of all releases have correlation value of over 10%, 20% over 30% and 9% of releases higher than 50%. So far, these results are the best for LSEdit case study. Therefore, we can conclude that

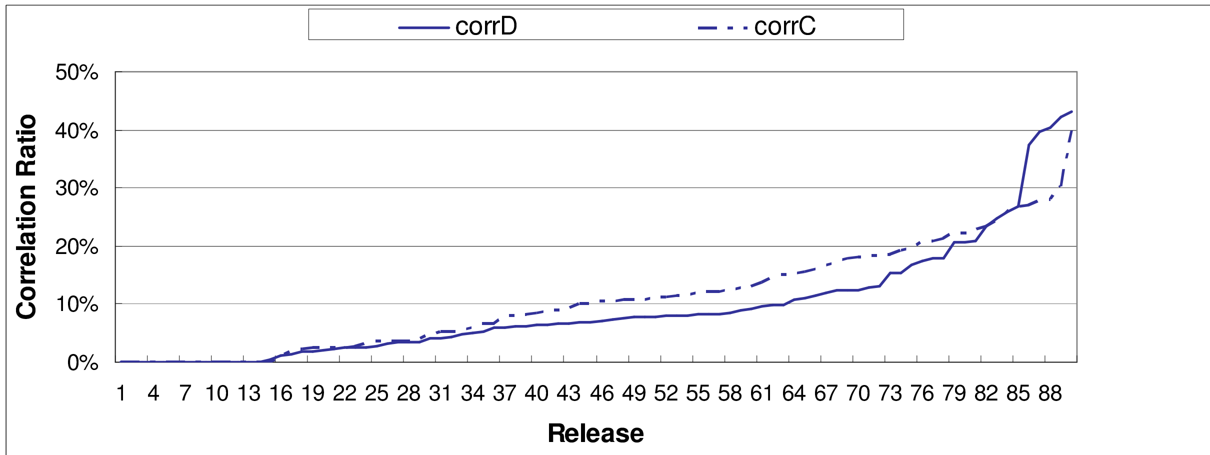

Figure 4.8: New

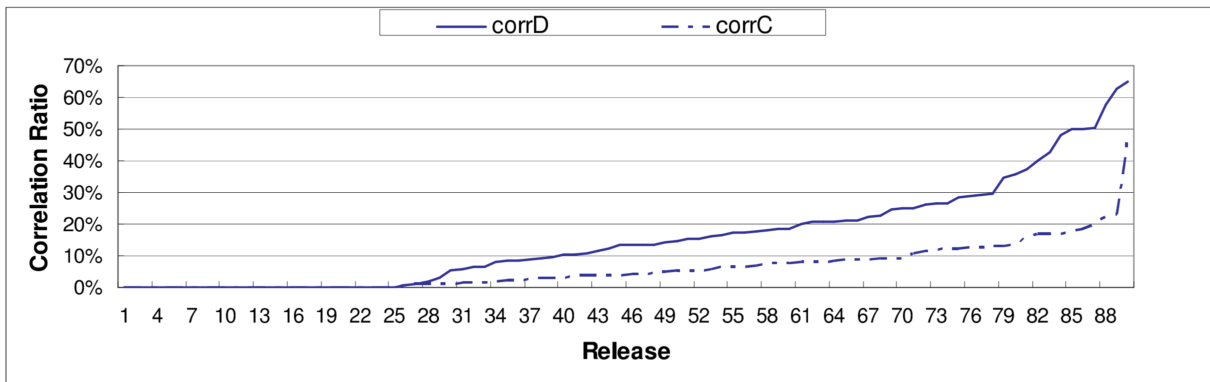

Figure 4.9: Repeated

repeated topic vocabulary has the highest correlation value with the changes in the code. This statement supports the hypothesis that discussions are related to the modifications in the code.

Next, we computed the similarity between discussion vocabulary of a certain release with the changed code vocabulary of all the releases that follow this release for a complete release history of LSEdit. We are unable to present the results due to space limitations, as it would require to display 90 $\times$ 90 correlation matrices. Rather, we summarize the interesting findings from the computed correlation values:

- Compared to minor releases, major releases hold higher correlation ratios between discussion and changed code vocabularies.
- A repeated topic is more likely than a new topic vocabulary to be implemented in major releases.
- A repeated topic is more likely than a new topic vocabulary to affect code changes.
- Discussions of minor releases always contribute to the changes in major releases.
- Code modifications implement new topic rather than repeated topic vocabulary.

After the conclusion that repeated words seemed to be most helpful to predict code changes, we decided to recover a maintenance vocabulary for LSEdit. A maintenance vocabulary is formed by the words that are shared by every single release. Unfortunately, we were not able to compute it, even in the case when zero-size vocabularies are omitted. Again the failure to compute maintenance vocabulary is caused by empty discussion intervals during the evolution of LSEdit. Another reason is that the typical discussion vocabulary is too small for there to be common terms among them.

4.2 Apache Ant Case Study

The software system used for the second case study is Apache Ant [14]. Apache Ant is a Java build tool. Ant is as an evolving software system of a medium-size, which contains 666 files) and written in Java. We have chosen this system because it is a open source software under the Apache Software License of Version 1.1 and Version 2.0, and therefore all the development information is publicly available. But obviously, we are interested only in the source code distribution and the mailing list archives.

In our case study, we investigated the complete release history of Ant consisting of 16 versions, as shown in Table 4.1. Email archives accumulated during the development process of Ant tool, are of significant size, 67377 emails. We considered electronic communications among developers only. User mailing lists were not analyzed for the reason that users mainly concern about bug fix issues rather than issues of architectural nature.

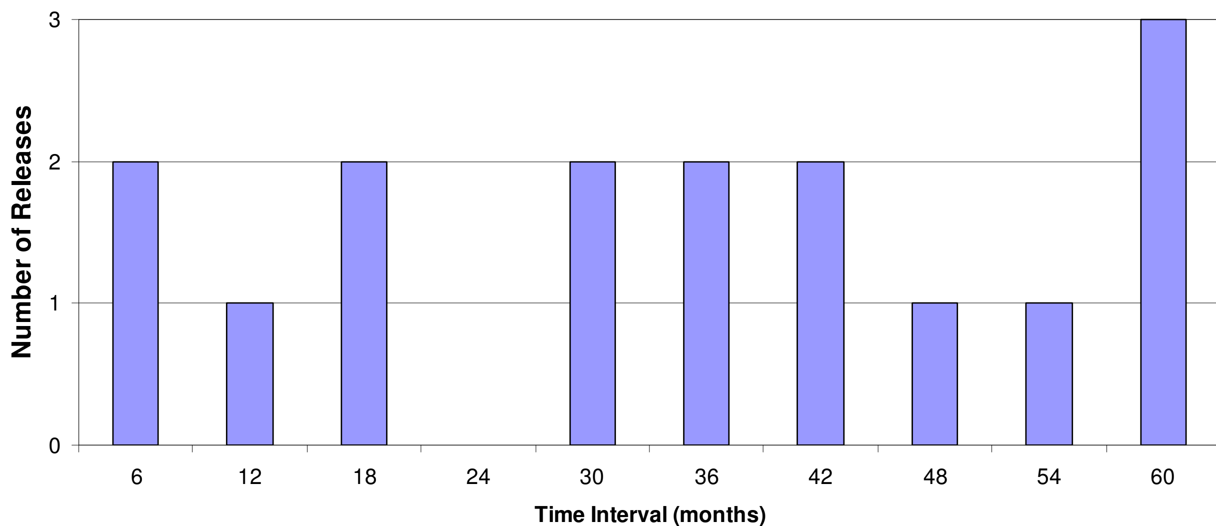

Figure 4.10: Release Distribution of Apache Ant

Figure 4.10 displays the distribution of release history of Apache Ant that was analyzed

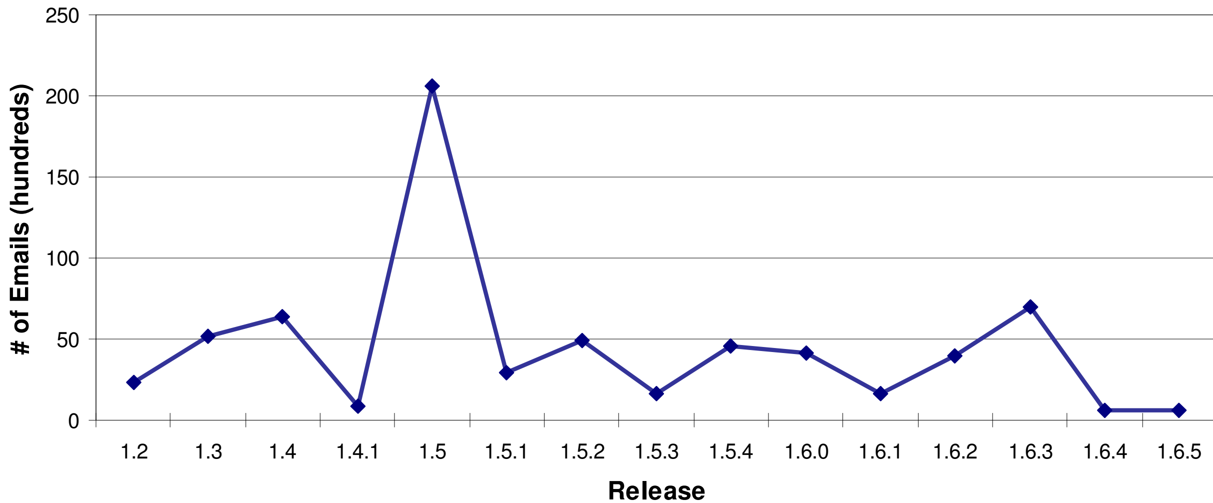

Figure 4.11: Email Distribution Across Release History of Apache Ant

in this study. Releases are almost evenly distributed over three years of Ant's lifecycle. After one and half year there was a longer time interval, which is expected in the case of delivering a major release. In fact, release 1.5 includes significant amount of new features-tasks. A new naming convention for Ant's releases was introduced also in release 1.5.

Email distribution across the release history of Apache Ant is shown in Figure 4.11. The largest discussion document, about Release 1.5, consists of over 20K email messages, while the smallest one, about Release 1.6.5, has 565 emails. On average, without accounting the highest peak of email distribution curve, the size of a discussion document is about 3K emails.

Before disclosing the results of our conceptual similarity method for Ant study, we need to mention that we did not apply the change event–topic correlation method on Ant due to the weaknesses of the approach. For more details refer to the Chapter 3.1 and Chapter 4.1.

Release	Regular	New	Repeated
1.2	6907	2690	4217
1.3	11318	6493	4825
1.4	11136	4810	6326
1.41	3975	787	3188
1.5	16508	13031	3477
1.5.1	7099	1616	5483
1.5.2	9106	4672	4434
1.5.3	4353	1173	3180
1.5.4	8336	5213	3123
1.6.0	7978	3582	4396
1.6.1	4727	1542	3185
1.6.2	7114	4030	3084
1.6.3	9798	5300	4498
1.6.4	2752	481	2271
1.6.5	2458	1160	1298

Table 4.2: Sizes of discussion vocabularies for Ant

4.2.1 Conceptual Similarity Method

Table 4.2 summarizes the sizes of three different types of a discussion vocabulary. The first column, Release, represents the release number, the next three columns, Regular, New and Repeated, represent the size of a regular, new and repeated discussion vocabularies respectively. Each regular discussion vocabulary is composed of both the new and repeated words. Hence, the sum of the total number of terms in new vocabulary with the total number of terms in repeated vocabulary defines the size of the regular discussion vocabulary. The largest discussion vocabulary contains 16508 terms and belongs to the release 1.5,

which also has the largest vocabulary document. The maximum size of regular discussion vocabulary is 16508, new vocabulary is 13031 and repeated vocabulary is 6326 terms. On average, a discussion vocabulary has 7571 terms. Comparing to LSEdit, having 252 terms in an average discussion vocabulary, it is extremely large.

Release	Size	Release	Size
1.2	2249	1.5.4	290
1.3	2965	1.6.0	5931
1.4	3699	1.6.1	395
1.4.1	3660	1.6.2	1303
1.5	5144	1.6.3	1324
1.5.1	227	1.6.4	12
1.5.2	819	1.6.5	21
1.5.3	5044		

Table 4.3: Size of changed code vocabulary for Ant

We observed that for most releases discussion vocabularies contain repeated topic vocabulary rather than new topic, as illustrated in Figure 4.12. It is expected from extensive discussions with large vocabularies to involve repetition of words.

Changed code vocabulary of Ant was constructed in the same way as the one for LSEdit, described in Chapter 4.1.2. We focused on comments, class and method names as source code identifiers to build changed code vocabularies for Ant. The size of the changed code vocabulary for each release is presented in Table 4.3. Each vocabulary ranges in size, starting from 12 to almost 6000 keywords per vocabulary. The average vocabulary contains about 2200 words, which is ten times bigger than the average size of changed code vocabulary for LSEdit.

Being a mid-size system, Ant has much richer vocabularies of both discussion and

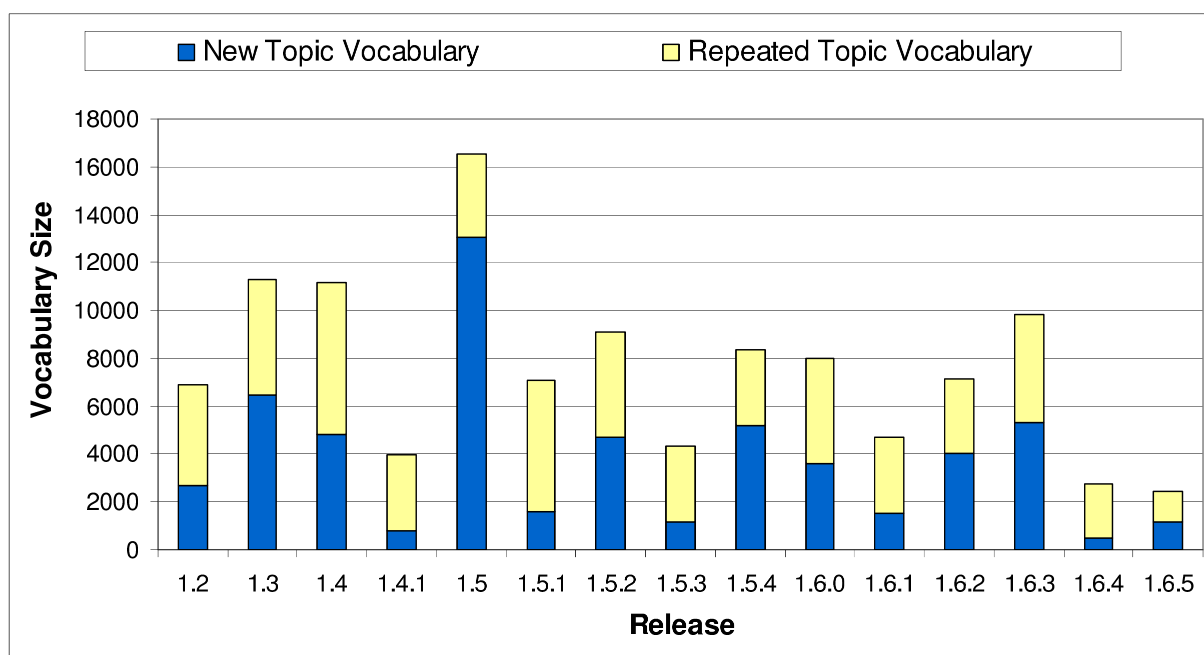

Figure 4.12: Ratio of New and Repeated Topics in Discussion Vocabulary for Ant

changes than LSEdit, so we expect to see a better performance of our correlation method. Figure 4.13 shows the results obtained in this case. Similar to Figures 4.7, 4.8 and 4.9, this figure represents the results of the correlation of changed code vocabulary with various types of the discussion vocabulary. We observed several things. First, we can see that regular and repeated topic vocabularies, denoted by dashed lines, have a high ratio of $corr_C$, which is more than 70% for regular one and above 60% for repeated one. This shows the evidence of correlation between the changes and the email vocabulary, the more modifications made in source code, the more changes occur in discussion vocabulary. However, the correlation $corr_C$ of these vocabularies with the code changes is not very high, 17% on average for regular and 28% for repeated vocabulary. Thus, it is hard to tell about the influence of the content of the email interactions on the source code changes.

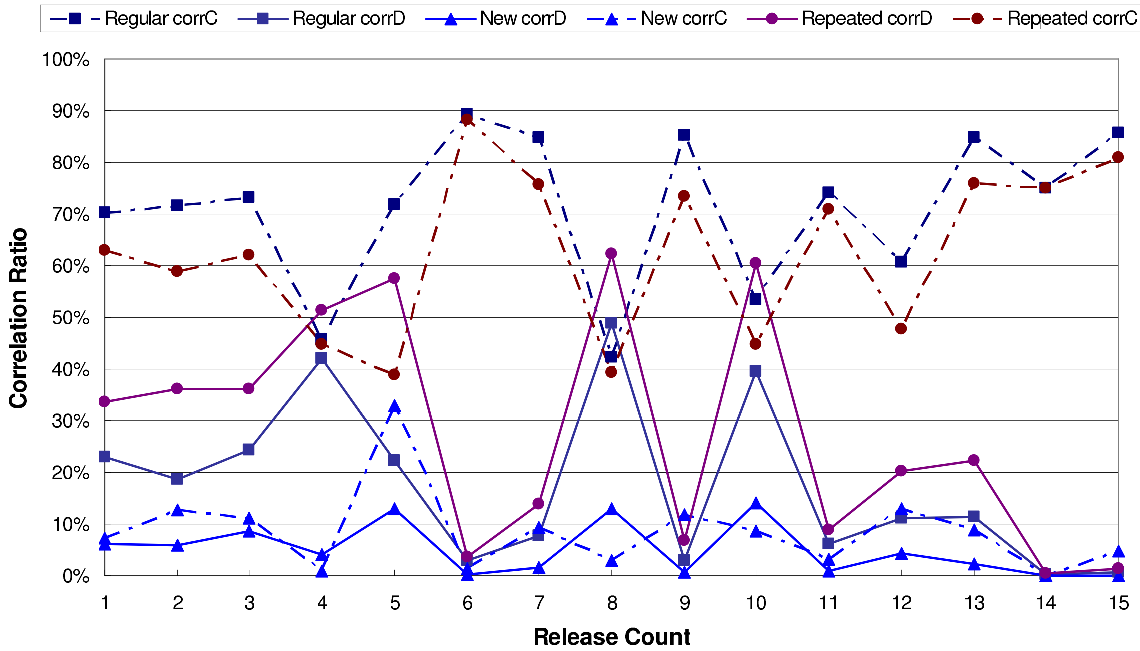

Figure 4.13: Results for Ant

Next, we noticed similar behavior of the curves Regular $corr_C$ and Repeated $corr_C$. These two dashed lines follow similar change patterns, except for the correlation value in release 1.5, in which the size of repeated discussion is much smaller compared to the one of new topic discussion, therefore it reduces the number of common concepts and the correlation value.

Analyzing the results, we noticed that some correlation values are not as high as we would expect, even in the case when a release has large discussion and changed code vocabularies. We examined how the discussion of each release relates to the source changes in the releases that come next. Therefore, next we concentrated on discovering latent correlation. The word “latent” means potential but not evident or active, it also means hidden [10].

When revealing latent correlation between the discussion of the current release with

	1.5	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**		**1.5**	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**
1.5	22%	1%	4%	22%	2%	25%	**1.5**	72%	94%	89%	72%	88%	69%
1.5.1		3%	9%	38%	3%	41%	1.5.1		89%	79%	53%	81%	50%
1.5.2			8%	34%	3%	37%	1.5.2			85%	61%	84%	57%
1.5.3				49%	5%	53%	1.5.3				42%	74%	39%
1.5.4					3%	39%	1.5.4					85%	55%
1.6.0						40%	**1.6.0**						53%

Table 4.4: Correlation matrices represent $corr_D$(left) and $corr_C$(right) for regular vocabulary.

the code modifications of the releases following next, we observed an interesting behavior of the values in the correlation matrices of several Ant releases. Tables 4.4, 4.5 and 4.6 show these correlation matrices. Each table shows the correlation between discussion vocabulary of a given release with the changed code vocabularies of that release and all the other releases that come after it.

For example, in in Table 4.4, the first correlation value 22% represents the correlation ratio of the discussion vocabulary of the release 1.5 with the changed code vocabulary for the corresponding release 1.5. The next value in the same row, 1%, stands for the correlation ratio between discussion vocabulary of the release 1.5 and changed code vocabulary of the release 1.5.1.

In this table, the values of $corr_D$ in major releases, that are shown in bold, are much higher than those in the minor releases, meaning that discussions of any minor releases are more likely to affect the changes in major releases than in minor ones. The discussions of major releases relate to the changes in those releases with the average correlation ratio of 36%. Contrary to $corr_D$, the values of $corr_C$ are higher in minor releases than in major ones, meaning that on average 85% of actual code changes in minor releases correspond to

	1.5	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**		**1.5**	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**
1.5	13%	0%	1%	13%	0%	15%	**1.5**	33%	12%	22%	33%	17%	33%
1.5.1		0%	0%	5%	0%	7%	1.5.1		1%	1%	2%	1%	2%
1.5.2			2%	14%	0%	17%	1.5.2			9%	13%	6%	13%
1.5.3				13%	0%	16%	1.5.3				3%	1%	3%
1.5.4					1%	22%	1.5.4					12%	19%
1.6.0						14%	**1.6.0**						9%

Table 4.5: Correlation matrices represent $corr_D$(left) and $corr_C$(right) for new vocabulary.

	1.5	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**		**1.5**	1.5.1	1.5.2	1.5.3	1.5.4	**1.6.0**
1.5	57%	5%	16%	57%	6%	61%	**1.5**	39%	81%	67%	39%	71%	36%
1.5.1		4%	12%	47%	4%	52%	1.5.1		88%	78%	52%	81%	48%
1.5.2			14%	55%	5%	59%	1.5.2			76%	48%	79%	44%
1.5.3				62%	7%	67%	1.5.3				39%	72%	36%
1.5.4					7%	68%	1.5.4					73%	36%
1.6.0						60%	**1.6.0**						45%

Table 4.6: Correlation matrices represent $corr_D$(left) and $corr_C$(right) for repeated vocabulary.

the ones discussed in the electronic communication.

Comparing the results in Tables 4.5 and 4.6, we draw the following conclusions:

- A repeated topic is more likely than a new topic of a discussion vocabulary to affect code changes. It is not surprising as Ant's discussion vocabularies mostly contain repeated topics.
- A repeated topic is more likely than a new topic to be implemented in minor releases.

- A new topic, in general, is weakly related to the changes as the correlation ratio is less than 15% . In rare cases of correlation, it associates with the changes of Ant's major releases.
- Code modifications tend to implement repeated topic rather than new topic vocabulary.

We also noticed that release 1.5.3 doesn't behave as a minor release. All the results in tables show that the correlation values for this release follow the same correlation patterns as major releases do.

Maintenance vocabulary of Ant consists of following 34 concepts: add ant apache attribute class code constructor date default defaults directory element exception execute file flag method names new optional output path project property run set sets software source string task use used version. These words are the concepts mentioned in the email interactions happened prior to every single release except for the last two releases. We decided not to consider releases 1.6.4 and 1.6.5 for the reason that their vocabularies share only 1-2 terms. If we did include the last two releases in our calculation, the maintenance vocabulary would contain these 1-2 terms in the best case. In fact, the maintenance vocabulary for the complete release history for Apache Ant is empty.

We noticed that some concepts in the maintenance vocabulary like constructor and exception are related to the program items, for example maintaining Ant system, developers are interested in the robust exception mechanisms. Others, like task, project, attribute and property are mainly domain concepts used as the keywords in a build file.

4.3 Discussion

This section we begin by comparing the results of our experiments on two software systems. We then present the correlation patterns discovered by analyzing the results of the case studies. At the end, we discuss some weak points of our approach.

4.3.1 Comparison of Case Studies

Comparing the two case studies, the obvious difference is in the data used to validate our approach: the release history of Ant consists of only 16 versions, while the size of mailing list archives is very significant, shown in Table 4.1. On the contrary, LSEdit has a very big release history containing 91 released versions and a poor collection of emails.

The results are not as promising as we hoped. Table 4.7 summarizes the correlation results for both case studies, LSEdit and Apache Ant. The best results are achieved on correlating regular and repeated topic discussion vocabularies in Ant case study. The average values of $corr_C$ for these vocabularies are 71% and 63% respectively, while maximum values hit the 88-89% level. Even their bottom level exceeds 39%. This tells us that there are a lot of changes in the source code that were actually discussed in the emails. And that the repeated topic, not new one, of the discussion vocabulary is implemented in the changes. These findings can be used in the case when there are a lot of modifications in the code but the discussion surrounding these changes is not large. Then we can justify that these changes were actually discussed earlier, in previous discussions.

For both studies, the correlation values $corr_D$ are very low for any type of the discussion vocabulary. This shows it would be very difficult, almost impossible, to predict code modifications from the content of the emails.

The conclusion of the results is that issues which are repeated the most, are the ones

	LSEdit						Ant					
	Regular		New		Repeated		Regular		New		Repeated	
	$corr_D$	$corr_C$	$corr_D$	$corr_C$	$corr_D$	$corr_C$	$corr_D$	$corr_C$	$corr_D$	$corr_C$	$corr_D$	$corr_C$
avg	11%	17%	9%	11%	15%	6%	17%	71%	5%	9%	28%	63%
max	50%	60%	43%	40%	65%	47%	49%	89%	14%	33%	62%	88%
min	0%	0%	0%	0%	0%	0%	0%	42%	0%	0%	0%	39%

Table 4.7: Comparison of the results for LSEdit and Ant

that will be implemented in the code. Our understanding of repeated topic originally was the following: we considered these words to be trivial as they were about everyday maintenance tasks like bug fixes. A new interpretation of the topic vocabulary goes other way around. These are the words that carry importance of the issues discussed. If the matter is talked about again and again, it might be of big concern.

We were not able to compare maintenance vocabularies of LSEdit and Ant, because for LSEdit case study we were simply unable to determine one. The reason for this is the quality of the discussion vocabularies, which contain mostly new topics. In order to identify maintenance vocabulary, discussions should have some words in common, which is not the case for LSEdit case study.

4.3.2 Correlation Patterns

Table 4.8 presents a list of correlation patterns identified from the case studies on LSEdit and Ant. We observed these patterns from the correlation matrices computed for various types of discussion vocabularies, thus we grouped the patterns according to the vocabulary type. The list of correlation patterns includes five patterns for the regular vocabulary, three patterns for the new topic vocabulary and four patterns for the repeated topic vocabulary.

Discussion Vocabulary Type	Correlation Pattern	Apache Ant	LSEdit
Regular	Correlation between discussion and changes is higher in major releases than in minor ones	✓	✓
	Discussions of minor releases affect changes of major releases	✓	
	Longer discussions predict more changes	✓	hard to tell
	Discussions contain more new topic than repeated one		✓
	Discussions contain more repeated topic than new one	✓	
New	New topic is implemented in changes of major release	✓	✓
	Most changes are related to new topic of the longest discussion	✓	hard to tell
	Big changes are discussed in longer interactions prior to the current release		✓
Repeated	Repeated topic is higher correlated with small changes and thus more found in minor release	✓	✓
	Code modifications implement new topic vocabulary		✓
	Code modifications implement repeated topic vocabulary	✓	
	Big discussions contain less repeated vocabulary than smaller ones		✓

Table 4.8: Comparison Table

Applying our correlation patterns on both systems, we observed that some releases considered as minor in fact, are similar to the characteristics of major releases for the following reasons:

- they all have correlation values similar to those of most of the major releases in the system,
- they do not conform to the same correlation patterns as the rest of the minor releases.

We believe that these releases should be analyzed in details and later be treated as major ones.

The Table 4.9 summarize the releases for both LSEdit and Ant, that we believe should be labelled as major ones.

System	Release
LSEdit	6.0.13, 7.0.12, 7.0.28, 7.1.6, 7.1.13, 7.1.15
Ant	1.6.3

Table 4.9: “Major” minor releases for LSEdit and Ant

4.3.3 Weaknesses of Our Approach

The main idea of our correlation method is based on the assumption that developers use application-domain knowledge when writing programs and particularly when assigning names to program identifiers. Under this assumption, the changed code vocabulary shares a large amount of terms with the discussion vocabulary. If the number of common words decreases, our method will not achieve the same results as we obtained in our case studies. And of course, our method can not be applied to the cases when email communications

are carried in a language different from the one used in assigning program identifiers and writing comments.

When building changed code vocabulary, we extracted identifiers from the deltas computed using Unix *diff* command. These deltas store lines of code that have been added, deleted or modified. These approach of measuring changes in a system is very simple to implement since it is easy to compute such deltas. However, *diff*-like tools determine lexical difference and ignore the high-level structural changes of the software system. For example, if a class has been renamed or a method has been moved to another class, these will be counted as two change events: deleted line for the class renaming and added lines for removing a method.

Source code metrics [11, 55], origin analysis [16, 42] and clone detection [37, 43] can be used to detect structural changes like renames, moves, merges and splits. Therefore, employing such techniques to identify source code changes can greatly improve the results of our correlation approach.

We aimed at designing a lightweight approach to correlate email interactions with the code modifications. Our method does not need a lot of computations or data preprocessing when building vocabularies. The correlation is also computed by using exact string matching algorithm. Clearly, the simplicity of the approach, does not let us to detect the best possible correlation between two artifacts. To overcome this limitation, we propose to use LSI model, details are discussed in Chapter 5.

4.4 Summary

Chapter 3 describes two approaches for finding similarity between emails interactions and source code changes. This chapter presents the results of applying those two approaches

on two case studies with different characteristics of data.

Change event–topic correlation method demonstrated the existence of positive association between the quantity of discussion topics and the quantity of the changes. However, we could not detect a strong positive correlation between the two data sets, indeed we found two linear relationships among them. Thus, this method can be used as a diagnostic step in determining the presence of correlation.

The second method, based on the conceptual similarity between the vocabularies constructed from the mailing communication and modifications of code, demonstrated a better performance. We could identify several correlation patterns. Some patterns are common for both case studies, others differ from one system to another. The correlation values between email discussions and code changes vary depending on the size and type of the discussion vocabulary.

Chapter 5

Future Work

There are several future directions that can be followed to improve the results of our work.

The first immediate extension would be to implement our approach as a tool. We eventually hope to build a tool to assist developers and architects to monitor, plan and predict software changes. Right now our implementation is a set of scripts.

Although the results are promising to support future research in correlating social interactions and code changes, the correlation model needs to be further validated in different types of software systems to assess its performance. We should apply our approach on various case studies analyzing systems written in different programming languages, with different quality and quantity of mailing communication.

We could also include other types of electronic media, such as forums, online bug report systems and so on, to correlate with source code changes in order to help developers predict future ones.

In the process of building changed code vocabulary we extract identifiers of only program entities like class declarations, method names and comments. In future case studies, we should add variable names to the list of identifiers to enrich vocabulary of code changes.

Enlarging change code vocabulary might improve the results of our correlation approach.

Our correlation method is lightweight, it does not employ any semantic information when extracting keywords from the source code or email messages. To improve the results, we should apply a morphological analysis such as stemming [35], on the extracted terms that build discussion and changed code vocabularies. During the stemming process all plural forms of words are converted into singular ones, for example, files to file, and various forms of verbs are transformed into infinitives, for example, extracting and extracted will be changed to extract. Such morphological analysis can improve the results [1], however it will require additional computation. An alternative approach to achieve better correlation is to use a technique suggested by Goldin and Berry [17]. They developed a tool that finds commonalities between requirements by using a sliding window technique that compares sentences character-by-character, with the space not treated differently. Their tool supports automatic matching of subwords that share a common root, avoiding need for stemming.

Another possible future extension will be to perform term frequency analysis [38] - counting the number of times each term occurs in a document. Assigning weights to terms is a technique often used in information retrieval or text mining [54]. Weights are useful to measure the importance of a word in a document. In longer documents, term frequency is usually normalized to measure the actual importance of a term with a hight frequency count. Obviously, the terms of the highest weight are most commonly used words, for example, the, of, and, to, a, in, that [12], and in many cases they do not carry useful information. But since our text normalization step includes stop list removal process, all meaningless terms will be eliminated prior to term frequency computation.

Unlike WordNet [9] that accounts for semantic relationships between words, our correlation model does not support synonym problem. Synonyms, words having similar or close meanings, are treated as different words in our method. For example, words like change

and modification are carrying the same meaning, but yet we consider them as two distinct concepts. To overcome this limitation, we can use LSI model in order to find similarity between discussion document and changed code document. LSI model has shown good results in recovering trace links between source code and documentation [28]. LSI finds relevant documents by identifying similar concepts rather than single terms. Therefore, LSI is able to solve synonym problem by producing a positive similarity between related documents sharing no terms. LSI model also uses a term weight, the number of occurrences of a term in a document, to solve the problem of rare words, providing a possible solution for term frequency problem that our method lacks.

Chapter 6

Conclusions

This thesis describes our approach of attaching electronic communication history to the change history of a software system to help developers identify architectural changes based on the similarity of these two artifacts. We have validated our research question that conceptual correlation can provide useful recommendations about source code modifications by applying the approach to two open-source systems, LSEdit and Apache Ant. Although the correlation ratio between public interactions and change history is not very high, we can yet reveal valuable findings that human interactions can be very useful to propagate future changes in the source code.

We compare and analyze the results of two case studies to determine correlation patterns between two artifacts. These patterns support our hypothesis that discussions, in particular those that include a newly introduced topic, are more likely to affect major revisions of a system than minor ones, while a repeated topic, issues that are constantly discussed, is implemented in minor releases, indicating that bugs are likely to be fixed as soon as possible by issuing a minor revision.

We observed that a typical source code change is a function of the type of the discussion

vocabulary. A new topic has a higher correlation with the code modifications for small discussion corpus than a repeated topic has, while a repeated topic is more related to the changes of a system with a large amount of discussion documents than to the changes of a system with poor discussion corpus.

Identified correlation patterns demonstrated the evidence of similarity between code modifications and email discussions. These patterns can help developers manage subsequential changes.

We wanted to promote the use of social knowledge captured in electronic media during the development of open source projects in understanding and managing software changes. Our main premise, as described in Chapter 4.3.3, is that developers use their application domain knowledge in writing code and particularly in naming source code identifiers. So the names of source code items are likely to be related to the natural language words appeared in email messages to propose, state, and discuss upcoming changes. Thus, under this assumption, we are able to correlate source code changes with email messages. However, the performance of our correlation method decreases when the number of common concepts between the source code vocabulary and the discussion vocabulary reduces.

Recovered correlation patterns can be used to predict software changes by monitoring the interactions among developers.

The thesis discusses the weaknesses of our approach, as well as possible future extensions to improve this work. One of the main future directions is combining structural and semantic information extracted from the changed code and discussion documents.

The contributions of this work are:

- We proposed a method to correlate email discussions with the source code changes by finding common concepts between discussion and changed code vocabularies.
- We empirically evaluated our correlation models on two software systems.

- We identified correlation patterns that can help developers manage future modifications in the source code.

Bibliography

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

[3] Felix Bachmann and Len Bass. Managing variability in software architectures. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 126–132, New York, NY, USA, 2001. ACM Press.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[6] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Gunter Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.

[7] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[8] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[9] Princeton University Cognitive Science Laboratory. Wordnet, a lexical database for the english language).
`http://wordnet.princeton.edu/`. [Online; accessed 25-November-2006].

[10] Answers Corporation. Latent – Answers.com, world's greatest encyclodictionalmanacapedia.
`http://www.answers.com/latent`. [Online; accessed 02-November-2006].

[11] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM Press.

[12] Net Dictionary. List of 2000 most frequntly used words (Brown corpus).
`http://www.edict.com.hk/lexiconindex/`. [Online; accessed 21-November-2006].

[13] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society.

[14] The Apache Software Foundation. Apache ant. `http://ant.apache.org/`.

[15] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[16] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[17] Leah Goldin and Daniel M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engg.*, 4(4):375–412, 1997.

[18] David J. Hand. Data mining: Statistics and more? *The American Statistician*, 52(2):112–119, 1998.

[19] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[20] Richard C. Holt. Software architecture as a shared mental model. In *Proceedings of the ASERC Workhop on Software Architecture*, University of Alberta, aug 2002.

[21] Capers Jones. Software return on investment preliminary analysis. *Software Productivity Research, Inc*, 1993.

[22] Kostas Kontogiannis and Peter G. Selfridge. Workshop report: The two-day workshop on research issues in the intersection between software engineering and artificial intelligence (held in conjunction with ICSE-16). *Autom. Softw. Eng.*, 2(1):87–97, 1995.

[23] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):45–50, 1995.

[24] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[25] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Enhancing an artefact management system with traceability recovery features. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 306–315, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *International Conference on Software Engineering*, pages 103–112, 2001.

[27] Jonathan I. Maletic and Naveen Valluri. Automatic software clustering via latent semantic analysis. In *ASE*, pages 251–254, 1999.

[28] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Nenad Medvidović, Alexander Egyed, and Paul Grunbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *STRAW'03: Second International SofTware Requirements to Architectures Workshop at ICSE 2003*, Portland, Oregon, USA, 2003.

[30] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference*

on Software Maintenance (ICSM'00), pages 120–130, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.

[32] Josef Nedstam, Even-Andre Karlsson, and Martin Host. The architectural change process. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Ciaran O'Reilly, Philip Morrow, and David Bustard. Lightweight prevention of architectural erosion. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 59, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[35] Martin F. Porter. An algorithm for suffix stripping. pages 313–316, 1997.

[36] Eric Steven Raymond. O'Reilly & Associates, 1999. Originally appeared online in 1999.

[37] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. Technical report, Ithaca, NY, USA, 1987.

[39] Jelber Sayyad-Shirabad, Timothy Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *ICSM*, pages 95–104, 2003.

[40] University of Waterloo Software Architecture Group (SWAG). Lsedit. `http://www.swag.uwaterloo.ca/lsedit/index.html`.

[41] Qiang Tu. On navigation and analysis of software architecture evolution. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1992.

[42] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *ICSM*, pages 398–407, 2001.

[43] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.

[44] Cornelis Joost van Rijsbergen. List of english stop words. `http://www.dcs.gla.ac.uk/idom/ir_resources/linguistic_utils/stop_words`. [Online; accessed 25-August-2006].

[45] Cornelis Joost Van Rijsbergen. *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979.

[46] Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.

[47] Peter Weissgerber and Stephan Diehl. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005. Student Member-Thomas Zimmermann and Member-Andreas Zeller.

[48] Wikipedia. Comments — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Comments`. [Online; accessed 26-October-2006].

[49] Wikipedia. Correlation matrices — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Correlation`. [Online; accessed 05-October-2006].

[50] Wikipedia. Data mining — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Data_mining`. [Online; accessed 15-October-2006].

[51] Wikipedia. Identifier — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Identifier`. [Online; accessed 26-October-2006].

[52] Wikipedia. Open source — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Open_source`. [Online; accessed 17-October-2006].

[53] Wikipedia. Scatterplot — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Scatterplot`. [Online; accessed 28-October-2006].

[54] Wikipedia. Tf-idf — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Tf-idf`. [Online; accessed 21-November-2006].

[55] Jingwei Wu. *Open Source Software Evolution and Its Dynamics*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2006.

[56] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

Appendix A

LSEdit Results

Table A.1: Size of discussion vocabularies for LSEdit

Release	Regular	New	Repeated
6.0.2	0	0	0
6.0.3	92	0	92
6.0.4	85	77	8
6.0.5	139	124	15
6.0.6	146	127	19
6.0.7	0	0	0
6.0.8	232	232	0
6.0.9	64	0	64
6.0.10	0	0	0
6.0.11	24	24	0
6.0.12	0	0	0
6.0.13	123	123	0

Continued on Next Page...

Table A.1 – Continued

Release	**Regular**	**New**	**Repeated**
6.0.14	16	11	5
6.0.15	0	0	0
6.0.16	196	196	0
6.0.17	770	659	111
6.0.18	126	37	89
6.0.19	508	472	36
6.0.20	0	0	0
7.0.1	800	800	0
7.0.4	969	653	316
7.0.5	105	57	48
7.0.6	94	78	16
7.0.7	48	47	1
7.0.8	106	94	12
7.0.9	98	90	8
7.0.10	155	131	24
7.0.12	412	360	52
7.0.13	267	176	91
7.0.14	35	28	7
7.0.15	92	86	6
7.0.16	183	168	15
7.0.17	25	13	12
7.0.18	133	124	9
7.0.19	94	79	15

Continued on Next Page...

Table A.1 – Continued

Release	**Regular**	**New**	**Repeated**
7.0.20	359	335	24
7.0.21	31	19	12
7.0.22	396	374	22
7.0.23	344	205	139
7.0.24	152	112	40
7.0.25	565	490	75
7.0.26	444	293	151
7.0.28	245	156	89
7.0.29	306	230	76
7.0.30	136	92	44
7.0.31	42	28	14
7.0.33	386	372	14
7.0.34	466	343	123
7.0.35	53	26	27
7.0.36	183	166	17
7.0.37	531	453	78
7.0.38	307	186	121
7.0.39	352	283	69
7.0.40	40	26	14
7.0.41	0	0	0
7.0.42	169	169	0
7.0.43	93	73	20
7.0.44	40	33	7

Continued on Next Page...

Table A.1 – Continued

Release	**Regular**	**New**	**Repeated**
7.0.45	328	310	18
7.0.46	639	509	130
7.0.47	357	208	149
7.0.48	156	85	71
7.0.49	628	547	81
7.0.50	320	144	176
7.0.51	185	123	62
7.0.52	344	288	56
7.0.53	455	312	143
7.0.54	0	0	0
7.0.55	489	489	0
7.0.56	454	311	143
7.0.57	160	111	49
7.0.58	341	279	62
7.0.59	439	346	93
7.0.60	290	182	108
7.1.4	130	84	46
7.1.6	591	546	45
7.1.7	860	628	232
7.1.13	713	482	231
7.1.14	497	312	185
7.1.15	317	212	105
7.1.16	79	63	16

Continued on Next Page…

Table A.1 – Continued

Release	Regular	New	Repeated
7.1.17	219	204	15
7.1.18	698	612	86
7.1.19	339	198	141
7.1.20	433	284	149
7.1.21	170	98	72
7.1.22	108	67	41
7.1.23	0	0	0
7.1.24	111	111	0
7.1.25	339	305	34

Table A.2: Size of changed code vocabularies for LSEdit

Release	Size
6.0.2	1207
6.0.3	98
6.0.4	91
6.0.5	58
6.0.6	94
6.0.7	24
6.0.8	27
6.0.9	9

Continued on Next Page…

Table A.2 – Continued

Release	**Size**
6.0.10	22
6.0.11	139
6.0.12	31
6.0.13	1347
6.0.14	83
6.0.15	188
6.0.16	151
6.0.17	300
6.0.18	41
6.0.19	57
6.0.20	227
7.0.1	1612
7.0.4	142
7.0.5	90
7.0.6	27
7.0.7	209
7.0.8	23
7.0.9	300
7.0.10	258
7.0.12	1353
7.0.13	113

Continued on Next Page...

Table A.2 – Continued

Release	**Size**
7.0.14	61
7.0.15	183
7.0.16	50
7.0.17	76
7.0.18	142
7.0.19	22
7.0.20	115
7.0.21	62
7.0.22	178
7.0.23	143
7.0.24	106
7.0.25	552
7.0.26	153
7.0.28	1288
7.0.29	121
7.0.30	155
7.0.31	193
7.0.33	158
7.0.34	158
7.0.35	154
7.0.36	26

Continued on Next Page…

Table A.2 – Continued

Release	**Size**
7.0.37	199
7.0.38	5
7.0.39	133
7.0.40	158
7.0.41	73
7.0.42	138
7.0.43	61
7.0.44	17
7.0.45	89
7.0.46	132
7.0.47	93
7.0.48	72
7.0.49	165
7.0.50	151
7.0.51	50
7.0.52	26
7.0.53	149
7.0.54	33
7.0.55	56
7.0.56	208
7.0.57	0

Continued on Next Page…

Table A.2 – Continued

Release	**Size**
7.0.58	55
7.0.59	130
7.0.60	19
7.1.4	129
7.1.6	618
7.1.7	243
7.1.13	1797
7.1.14	959
7.1.15	1018
7.1.16	82
7.1.17	109
7.1.18	357
7.1.19	146
7.1.20	256
7.1.21	212
7.1.22	85
7.1.23	46
7.1.24	20
7.1.25	158

Index

corpus
  code changes, 27
  discussion, 26
correlation, 32
  $corr_C$, 33
  $corr_D$, 32
  matrix, 36

delta, 26
document
  discussion, 26
  source code, 26

release history, 25

scatter plot, 22
similarity, 32

vocabulary
  changed code, 29
  discussion, 27
    maintenance, 37
    new topic, 28
    repeated topic, 28