Correlating Social Interactions to Release History During Software Evolution

Olga Baysal
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
obaysal@uwaterloo.ca

Andrew J. Malton
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
ajmalton@uwaterloo.ca

Abstract

In this paper, we propose a method to reason about the nature of software changes by mining and correlating discussion archives. We employ an information retrieval approach to find correlation between source code change history and history of social interactions surrounding these changes. 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.

1. Introduction

Imagine that there were a tool that could store a record of all social 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 paper 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 [9]. 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 discussion archives that can be used to understand the nature of the changes. 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.

Social interactions are studied as the collection of written discussions between stakeholders, especially developers and users, in the form of mailing lists and e-mail archives.

In our approach we used the techniques of the Natural Language Processing(NLP). Despite their success in many areas, NLP methods are little used in software engineering.

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

Similar to Biggerstaff [3] 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 discussion archives in order to find common concepts—words that are common to the vocabularies of the discussion archives and source code changes. 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, discussion 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.

The rest of the paper is organized as follows. Section 2 presents a brief overview of related and background work. Section 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. Section 4 provides case studies and the results that were obtained after applying our approach on two open source software systems. Section 6 discusses possible directions of future work in the area of identifying and predicting software changes. Finally, Section 5 outlines the contributions of our work.

2. Related Work

A number of data mining studies have drawn on release history information to reveal the nature of software change [13, 17, 8], to find change patterns [16, 18, 15] and even to predict future changes [10, 5].

Godfrey and Tu [13] investigated a way to detect and model structural changes such as moving and renaming, by performing origin analysis [12]. Origin analysis is used to reason about where, how and why the design changes have occurred in the system. Wu investigated the punctuated evolution [17]. He observed that software architecture mainly changes during the punctuation periods that are the periods of sudden and discontinuous change.

Zimmermann *et al.* [16] presented a data mining approach over Concurrent Versions System(CVS) repositories to recommend source code that is relevant to a given source code fragment. Ying *et al.* [18] 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. Mockus *et al*. [8] 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. Hipikat [15] is a tool that gives recommendation about the project information a developer should consider during a modification task.

Shirabad *et al.* [10] used machine learning techniques to extract models from the past experience that can be used in future predictions. Hassan and Holt [5] used historical source control systems to predict change propagation. They presented some heuristics for change propagation, as well as the approach to study various change propagation models [5].

Several researchers have investigated relationships between software artifacts. 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 two IR models, probabilistic and vector space model (VSM), to rank the documents against the query consisting of the source code identifiers. Marcus and Maletic [7] used Latent Semantic Indexing (LSI), an extension of the VSM, that searches for concepts rather than searching for terms. De Lucia *et al*. [6] 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, discussion archives and source code. However, we did not apply some predefined IR models but rather we first applied data mining techniques on the release history of a software system and discussion archives to recover useful information about these resources and then we used NLP methods in text analysis to identify correlation between code and social interactions among developers.

3. Correlating Discussion Archives with Source Code Changes

The conceptual similarity method [2] uses the terms extracted from the discussions and identifiers extracted from the code to find correlation between natural language discussion archives and source code changes. 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.

In our work, we aim at mapping discussion archives 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 discussion 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.

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*. Each delta contains a line by line difference, such as added, deleted and changed lines, between two source code documents, thus Δ_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.

3.1. Building the Discussion Vocabulary

We now define the regular, new, and repeated vocabularies which arise from the discussion history. The idea is that the regular vocabulary represents the common coin throughout the history of the system; that the new vocabulary is whatever is being specially discussed for the next release; and the repeated vocabulary is whatever is still being specially discussed since the last release.

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

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 [14] is applied to eliminate most common English words that are articles, prepositions, etc.

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

0-7695-2950-X/07 $20.00

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:

$$N_i = D_i - D_{i-1}. \quad (1)$$

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

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:

$$P_i = D_i \bigcap D_{i-1}. \quad (2)$$

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

3.2. Building the Changed Code Vocabulary

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 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. 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.

The process of building changed code vocabulary consists of the following steps:

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.

The process of building the changed code vocabulary slightly differs from the process of building discussion vocabulary. 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 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.

3.3. Comparison of the Vocabularies

The final stage of the approach deals with comparing two generated vocabularies. We define a *correlation* between discussion vocabulary and changed code vocabulary as a set of terms that two vocabularies have in common 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.

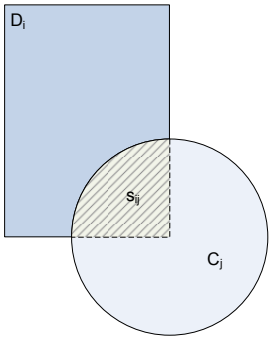

Figure 1. Finding correlation between two vocabularies

Figure 1 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 propose two more correlation measures as:

$$corr_D(D,C) = \frac{\mid D \bigcap C \mid}{\mid D \mid}, 0 \leq corr_D(D,C) \leq 1 \quad (3)$$

and

$$corr_C(D,C) = \frac{\mid D \bigcap C \mid}{\mid C \mid}, 0 \leq corr_C(D,C) \leq 1 \quad (4)$$

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

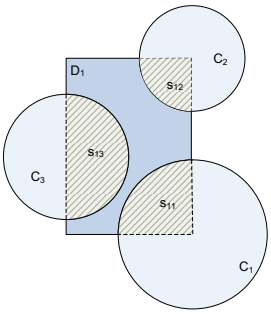

Figure 2. How discussion vocabulary affects future changes

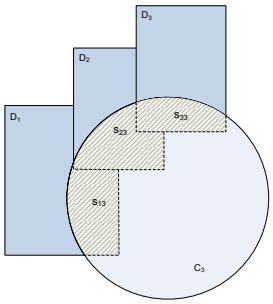

Figure 3. How many changes were discussed earlier

The first correlation $corr_D$ corresponds to how much of the source code changes are discussed by stakeholders, shown in Figure 2. While the second one $corr_C$ determines how much of the discussed issues are actually found in changes, demonstrated in Figure 3. The larger area of the intersection between the box and the circle, the stronger correlation between the vocabularies.

Next, we examine how new topic relates to the source code changes. In Figure 4, correlation s_{22} shows that a new topic affects more than one third of all the modifications in the source code.

To determine a correlation between the repeated topic vocabulary and the changed code vocabulary, shown in Figure 5, 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.

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 is computed to indicate the strength of the relationships between discussion vocabularies and changed code vocabularies for the complete release history of a system.

A *correlation matrix* is a k × 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 5, where entries below or above, for lower triangle, the main diagonal are zeros s_{ij} = 0 if i > j :

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & . & . & s_{1k} \\ 0 & s_{22} & s_{23} & & & s_{2k} \\ 0 & 0 & s_{33} & & & . \\ . & & & . & & . \\ . & & & & . & . \\ 0 & . & . & . & 0 & s_{kk} \end{pmatrix} \quad (5)$$

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.

Several correlation matrices are generated. Each matrix is characterized by the correlation measure and the type of a discussion vocabulary used at the comparison stage.

4. Two Case Studies

We apply the proposed correlation method in two case studies. The goal is to assess how well our approach of correlating software changes with the social interactions among stakeholders performs on systems with different characteristics. As shown in Table 1, the studied systems have different sizes of both the release history and email interactions.

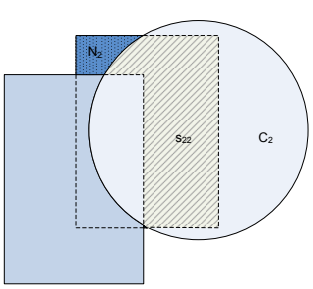

Figure 4. Correlation of new topic vocabulary with changed code vocabulary

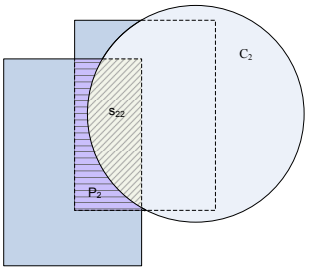

Figure 5. Correlation between repeated topic and changed code vocabularies

0-7695-2950-X/07 $20.00 © 2007 IEEE

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

Table 1. Size of release history and discussion archives for LSEdit and Apache Ant

The first case study was a freely available graph visualization tool, called LSEdit [11], 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. Over the three and half years, 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. We examined only 91 sequential released versions starting from release 6.0.1 to release 7.1.25.

Due to poor email communication during the development process of LSEdit, the size of its discussion vocabulary for each release is quite small. 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. 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, discussion archives 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. 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.

The software system used for the second case study is Apache Ant [4]. 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. We investigated the complete release history of Ant consisting of 16 versions. Discussion archives accumulated during the development process of Ant tool, are of significant size, 67377 emails. We considered electronic communications among developers only.

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.

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

Table 2. Comparison of the results for LSEdit and Apache Ant

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, the size of a discussion document is about 3K emails. The largest discussion vocabulary contains 16508 terms and belongs to the release 1.5. On average, a discussion vocabulary has 7571 terms. Comparing to LSEdit, having 252 terms in an average discussion vocabulary, it is extremely large. We observed that for most releases discussion vocabularies contain repeated topic vocabulary rather than new topic. It is expected from extensive discussions with large vocabularies to involve repetition of words. Changed code vocabularies range 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.

4.1. Comparison of Case Studies

We are unable to present the complete results [2] due to space limitations. Rather, we summarize the results for both 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 discussion archives is very significant, shown in Table 1. On the contrary, LSEdit has a very big release history containing 91 released versions and a poor collection of emails.

Table 2 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 discussion archives.

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

4.2. Correlation Patterns

Table 3 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.

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 summarize the releases for both LSEdit and Ant, that we believe should be labeled as major ones.

Correlation Pattern	Ant	LSEdit
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	✓	
Discussions contain more new topic than repeated one		✓
Discussions contain more repeated topic than new one	✓	
New topic is implemented in changes of major release	✓	✓
Most changes are related to new topic of the longest discussion	✓	
Big changes are discussed in longer interactions prior to the current release		✓
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 3. Correlation Patterns

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. "Major" minor releases for LSEdit and Ant

5. Conclusions

This paper 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 func-

tion 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 discussion archives. These patterns can be used to predict software changes by monitoring the interactions among developers.

6. 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. 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 discussion archives.

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 currently assumes no semantic information when extracting keywords from the source code or email messages. This informally avoids bias in the relationship between technical discourse and language use in source, but ignores elementary facts about natural language morphology. We will experiment with applying stemming and/or synonym correlation to increase (but also render less precise) the amount of correlation.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[2] O. Baysal. Attaching Social Interactions Surrounding Software Changes to the Release History of an Evolving Software System. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 2006.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. 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.

[4] T. A. S. Foundation. Apache ant. `http://ant.apache.org/`.

[5] A. E. Hassan and R. 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.

[6] A. D. Lucia, F. Fasano, R. Oliveto, and G. 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.

[7] A. Marcus and J. 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.

[8] A. Mockus and L. 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.

[9] E. S. Raymond. O'Reilly & Associates, 1999. Originally appeared online in 1999.

[10] J. Sayyad-Shirabad, T. Lethbridge, and S. Matwin. Mining the maintenance history of a legacy software system. In *ICSM*, pages 95–104, 2003.

[11] U. o. W. Software Architecture Group (SWAG). Lsedit. `http://www.swag.uwaterloo.ca/lsedit/index.html`.

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

[13] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *ICSM*, pages 398–407, 2001.

[14] C. J. 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].

[15] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.

[16] P. Weissgerber and S. 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.

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

[18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.