Proceedings

International Workshop on
Empirical Software Engineering in Practice 2011
(IWESEP 2011)

Nara, Japan, November 1st, 2011

Sponsored by
StagE Project, MEXT Japan
Osaka University
Nara Institute of Science and Technology (NAIST)

In cooperation with
SIG Software Science, Information and Systems Society, IEICE
SIG Software Engineering, IPSJ

Table of Contents

Open your Mind

Poster

Preface

It is our great pleasure to welcome everyone to the 2011 International Workshop on Empirical Software Engineering in Practice (IWESEP 2011). Our workshop aims to foster the development of the area by providing a forum where researchers and practitioners can report on and discuss new research results and applications in the area of empirical software engineering. The workshop encourages the exchange of ideas within the international community so as to be able to understand, from an empirical viewpoint, the strengths and weaknesses of technology in use and new technologies, with the expectation of furthering the field of software engineering in general.

IWESEP has received 10 submissions, including 6 regular papers and 4 tool demonstration proposals. After careful evaluation by the program committee, 5 regular papers and 3 tool demonstrations have been accepted to be presented at the workshop. The other papers were invited for a poster presentation, to still gather feedback from the workshop participants. The papers cover a variety of topics, including software quality, refactoring and analysis of open source development processes.

Finally, on behalf of the program committee and the organizing committee, we thank the attendants for making IWESEP 2011 such an interactive venue. We would also like to take this opportunity to thank the program committee members, who spent considerable time reviewing publications.

We hope you will have a great time and an unforgettable experience at IWESEP 2011.

Yasutaka Kamei, Kyushu University, Japan
IWESEP 2011 General Chair
Bram Adams, Queen's University, Canada
IWESEP 2011 Program Chair

Organization

General Chair

Yasutaka Kamei (Kyushu University, Japan)

Program Chair

Bram Adams (Queen's University, Canada)

Publication Chair

Ryosuke Nakashiro (Kyushu University, Japan)

Publicity Co-Chair

Kentaro Yoshimura (Hitachi, Ltd., Japan)

Emad Shihab (Queen’s University, Canada)

Registration Chair

Koji Toda (Nara Institute of Science and Technology, Japan)

Local Arrangements Chair

Kyohei Fushida (Nara Institute of Science and Technology, Japan)

Web Chair

Thanh H.D. Nguyen (Queen's University, Canada)

Advise Chair

Akinori Ihara (Nara Institute of Science and Technology, Japan)

Program Committee

Sousuke Amasaki (Okayama Prefectural University, Japan)
Christian Bird (Microsoft Research, USA)
Ahmed E. Hassan (Queen's University, Canada)
Hideaki Hata (Osaka University, Japan)
Yasuhiro Hayase (University of Tsukuba, Japan)
Shinpei Hayashi (Tokyo Institute of Technology, Japan)
Israel Herraiz (Complutense University of Madrid, Spain)
Abram Hindle (University of Alberta, Canada)
Takashi Ishio (Osaka University, Japan)
Yuichiro Kanzaki (Kumamoto National College of Technology, Japan)
Shinji Kawaguchi (Japan Manned Space Systems Corporation, Japan)
Hua Jie Lee (University of Melbourne, Australia)
Yin Liu (Rensselaer Polytechnic Institute, USA)
Yuki Manabe (Osaka University, Japan)
Meiyappan Nagappan (Queen's University, Canada)
Peter Rigby (University of Victoria, Canada)
Rodrigo Vivanco (University of Manitoba, Canada)
Thomas Zimmermann (Microsoft Research, USA)

Moneyball

Analysis of Software Maintenance Efficiency Focused on Process Standardization

Masateru Tsunoda
Nara Institute of Science and Technology
Kansai Science City, 630-0192
Japan
masate-t@is.naist.jp

Akito Monden
Nara Institute of Science and Technology
Kansai Science City, 630-0192
Japan
akito-m@is.naist.jp

Ken-ichi Matsumoto
Nara Institute of Science and Technology
Kansai Science City, 630-0192
Japan
matumoto@is.naist.jp

Tomoki Oshino
Economic Research Institute,
Economic Research Association
Higashi-Ginza Mitsui Bldg. 5-13-16,
Ginza, Chuo-ku, Tokyo, 104-0061 Japan
er352@zai-keicho.or.jp

ABSTRACT
In this research, to establish a benchmark for software maintenance efficiency, we analyzed factors affecting the work efficiency and showed reference value stratified the factors. We analyzed dataset of software maintenance collected from 83 organizations. Attributes recorded in the dataset are standardization status of organization, system architecture, the number of engineers, the number of base modules, the number of modified modules, and so on. In the analysis, we regarded modified modules per engineer as a work efficiency index, and clarified relationships to other attributes. As a result, we identified that standardization status of organization is important for work efficiency.

Categories and Subject Descriptors
K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*, **Management of Computing and Information Systems**]: general – *Economics*.

General Terms
Management, Measurement, Economics.

Keywords
Module modification, productivity, cross-company dataset, benchmarking.

1. INTRODUCTION
Recently, a number of software users (companies or organizations) contract with software developers (companies) for maintenance of enterprise software, and therefore the agreement of software maintenance becomes more important. Software maintenance does not mean only removing faults found after software release. Software needs extensions or modifications of its functions due to changes in a business environment, and software maintenance also indicates them. ISO/IEC 14764 classifies software maintenance into followings:

1. Corrective maintenance: modifications of faults found after software release.
2. Preventive maintenance: corrective modifications before potential faults become actual faults, after software release.
3. Adaptive maintenance: modifications to keep software availability against environmental changing after software release.
4. Perfective maintenance: modifications for conservation or improvement of software performance or maintainability after software release.

In this research, we try to establish a benchmark (reference values to compare an organization's work efficiency with others [5]) of work efficiency for software maintenance contract. To establish the benchmark, factors affecting work efficiency (e.g. system architecture) are clarified first, and then the dataset is stratified by the factors, using dataset collected from various organizations (cross-company dataset). When using the benchmark, compare work efficiency with a reference value whose factor (e.g. system architecture) is correspond to the target. We focus on the standardization of software maintenance process in the analysis. When standardization status (process is standardized or not) has strong relationships to work efficiency, reference values stratified standardization status can be used to confirm difference of work efficiency between standardized organizations and not standardized ones. That is useful to decide whether software maintenance process should be standardized or not. Major contribution of our research is to clarify relationships between standardization status and maintenance efficiency using cross-company dataset.

The dataset used in the analysis was collected from 83 organizations in 2007 by Economic Research Association [2]. We analyzed relationships between work efficiency and other attributes such as system architecture, in addition to standardization status. The dataset does not have enough cases to use analysis results as rigorous benchmark. So, values shown in the research should be

Table 1. Description of dataset

Attribute	Description
Process standardization	Status of standardization of software maintenance process (process is standardized, standardization is work in progress, or maintenance process is not standardized)
System architecture	System architecture on which the software runs (client–server system, Web based system, mainframe system)
Number of base modules	Total number of modules included in the software
Number of modified modules	The number of modified modules in the maintenance activity in a year
Number of engineers	The number of contractor's resident engineers
Modified modules per engineer	The number of modified modules / number of engineers
Modified modules per engineer and base	The number of modified modules / the number of engineers / the number of base modules
Human factor	Degree of difficulties about size of project (or organization) and level of skill
Problem factor	Degree of difficulties about type, importance, relationships, restriction, and ramification of problems
Process factor	Degree of difficulties about programming language and software development methodology
Product factor	Degree of difficulties about reliability, size, control structures, and complexity of the software
Resource factor	Degree of difficulties about hardware, duration, and budget
Tool factor	Degree of difficulties about library, complier, test tool, maintenance tool, and reverse engineering tool

used casually. Nevertheless, we think our result is effective because there are very few researches or reports which analyzed software maintenance efficiency using cross-company dataset.

In the analysis, we regarded modified modules per engineer as a work efficiency index. That is, the number of engineers was considered as inputs, and the number of modified modules in a year was considered as outputs (High modified modules per engineer indicates high work efficiency). It is more precise that a work efficiency index is defied using software maintenance effort and modified lines of code (or function point). However, modified lines of code (and function point) in the dataset used includes many missing values, and software maintenance effort is not recorded. In addition, it is easier for software uses (companies or organizations) to measure the number of modified modules than modified lines of code or function point (Actually, the dataset is almost collected from software uses). Similarly, it is easier to measure the number of engineers than software maintenance effort. So using the number of modified modules and the number of engineers makes it easy for software users to refer the benchmark.

In what follows, Section 2 explains dataset used in the analysis. Section 3 shows analysis results. Section 4 introduces related works, and Section 5 concludes the paper with a summary.

2. DATASET

The dataset used in the analysis includes 83 cases of software maintenance agreement which were collected from 83 organizations in 2007 by Economic Research Association [2]. 78 cases are business software, and rest cases are factory automation software and other software. 46 cases are fixed price contract (Software maintenance is performed during certain period by fixed price [6]). The cases were collected mainly from software uses (companies or organizations). Each case is representative software maintenance agreement in each organization (Each organization provided one case), and number of modified module was collected in a year. The number of analyzed cases was different for attributes, because each attribute includes missing values. Attributes analyzed in this research are described in Table 1, and the followings are detailed explanations for some attributes.

- Process standardization status does not represent the status of a case included in the dataset, but the status of the entire organization which offered the case (Values of other attributes were collected in each software maintenance agreement).
- When process is standardized, standard process of software maintenance (sequence of activities such as analyzing, reviewing, documenting, and approval) is explicitly defined.
- Some cases have multiple system architectures (for example, a case of system architecture includes both mainframe and web based system).
- Modified modules per engineer and base is explained in section 3.2.
- Attributes from human factor to tool factor (We call them productivity factors) are defined based on [6], and they were evaluated on a three-point scale (Low value indicates severe condition, i.e., productivity may be decreased).

3. ANALYSIS RESULTS

3.1 Analysis Procedure

In the analysis, we analyzed influences of system architecture, productivity factor, the number of base modules, and the number of engineers to work efficiency (modified modules per engineer), in addition to influence of process standardization. When an attributes has considerable influence, the dataset was stratified by the attribute, to eliminate the influence of the attribute. For example, if system architecture had influence to work efficiency, the dataset was stratified by it, and a relationship between process standardization and work efficiency was analyzed. The dataset has missing values, so the number of analyzed cases is different in each analysis.

Table 2. Relationships between size attributes

Attribute		Number of base modules	Number of modified modules	Number of engineers	Modified modules per engineer
Number of base modules	ρ	1.00	0.70	0.23	0.60
	p-value		0.00	0.30	0.02
	Number of cases	35	25	22	15
Number of modified modules	ρ	0.70	1.00	0.56	0.83
	p-value	0.00		0.03	0.00
	Number of cases	25	25	15	15
Number of engineers	ρ	0.23	0.56	1.00	0.03
	p-value	0.30	0.03		0.92
	Number of cases	22	15	35	15
Modified modules per engineer	ρ	0.60	0.83	0.03	1.00
	p-value	0.02	0.00	0.92	
	Number of cases	15	15	15	15

When analyzing a relationship of ratio (or ordinal) scale attributes, we used Spearman's rank correlation coefficient to avoid influence of outliers. In what follows, "correlation" and ρ indicate the Spearman correlation. We did not apply a multivariate regression model because there are many missing values in the dataset. Instead of applying it, we stratified the dataset as mentioned above.

We used a box plot to analyze relationships between nominal scale attribute and ratio scale attribute. In a box plot, the bold line in each box indicates the median value. Small circles indicate outliers, that is, values that are more than 1.5 times larger than the 25%-75% range from the top of the box edge. Stars indicate extreme outliers, whose values are more than 3.0 times larger than this range. Some outliers do not include figures to improve readability of them. Mann–Whitney U test was applied to confirm median is statistically different or not between two attributes. We set significance level as 5%.

3.2 Relationships between Size Attributes

We analyzed relationships between modified modules per engineer, the number of base modules, and the number of engineers, to confirm whether modified modules per engineer can be used as work efficiency index or not. It may be helpful in comprehending the analysis to regard the number of base modules as function point, the number of engineers as development effort, and modified modules per engineer as productivity.

Table 2 shows correlations of the relationships, and the major relationships are illustrated in Figure 1. Observations are as follows:

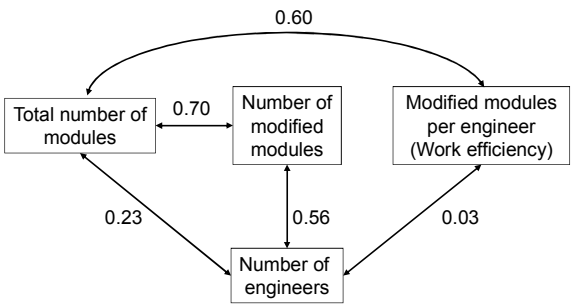


Figure 1. Major correlations between the number of base modules, number of engineers, and other attributes

- There is a positive correlation between the number of modified modules and the number of engineers (ρ = 0.56). So, when the number of engineers is large, the number of modified modules is also large.
- The correlation between the number of engineers and the number of base modules is not large (ρ = 0.23). Therefore, size of entire software does not greatly affect the number of engineers.
- The correlation between the number of engineers and modified modules per engineer is very small (ρ = 0.03). This does not mean fewer engineer increases modified modules per engineer.

The number of base modules has significant correlation to modified modules per engineer (ρ = 0.60) and the number of modified modules (ρ = 0.70). It may be likely that when the number of base modules is large, the number of modified modules is also large, and as a result, modified modules per engineer becomes large. So we defined modified modules per engineer and base (the number of modified modules / the number of engineers / the number of base modules), to eliminate influence of the number of base module, and used it to strengthen analysis results.

3.3 Relationship to System Architecture

Figure 2 and Figure 3 show boxplots of modified modules per engineer and modified modules per engineer and base for three different system architectures. In modified modules per engineer (Figure 2), mainframe system is the highest, and Web based system is the lowest. The median value of Web based system is 2.2 times smaller than client-server system, and 6.9 times smaller than mainframe system, as shown in Table 3. However, they are not statistically different. There are not enough cases in the dataset (see Table 3), and it may affect statistical test results.

Also, in modified modules per engineer and base, mainframe system is the highest while the distribution is wide, as shown in Figure 3. The distribution (position and size of the box in the figure) of Web based system is almost same as client-server system. The median value of Web based system is same as client-server system,

Table 3. Median values of modified modules per engineer (and base) stratified by system architecture

	System architecture	Number of cases	Median	p-value (difference from Web based)	p-value (difference from client-server)
Modified modules per engineer	Client-server	9	40.0	88%	-
	Web Based	11	18.0	-	88%
	Mainframe	8	125.0	21%	37%
	Total	28	55.0	-	-
Modified modules per engineer and base	Client-server	9	0.007	82%	-
	Web Based	11	0.007	-	82%
	Mainframe	8	0.011	27%	37%
	Total	28	0.008	-	-

Table 4. The number of base modules, the number of modified modules, and the number of engineers stratified by system architecture

System architecture		Number of base modules	Number of modified modules	Number of engineers
Client-server	Median	3000	125	10
	Number of cases	25	16	26
Web Based	Median	3000	180	10
	Number of cases	23	17	21
Mainframe	Median	4628	400	10
	Number of cases	21	14	20
Total	Median	3000	250	10
	Number of cases	69	47	67

and 1.6 times smaller than mainframe system (Table 3). Note that there is no statistically difference among them.

In the results, modified modules per engineer (and base) is larger when system architecture is mainframe system. Although we should carefully understand it because there is no statistically difference, it suggests system architecture should be considered when comparing work efficiency (modified modules per engineer).

We examined programming language used in each system. In the result, the most used programming language is Visual Basic in client-server systems, it is SQL in Web based systems, and it is COBOL in mainframe systems (Note that multiple programming languages are used in each system). Difference of work efficiency between system architectures may be affected by characteristics of programming languages.

We counted the number of base modules, the number of modified modules, and the number of engineers by system architecture (Table 4). Observations are as follows:

- Median values of the number of engineers are same among system architectures.
- In median values of the number of modified modules, mainframe system is 2.2 times larger than Web based system, and 3.2 times larger than client-server system.
- In median values of the number of base modules, mainframe system is 1.8 times larger than others.

In mainframe systems, it may be probable that the number of base

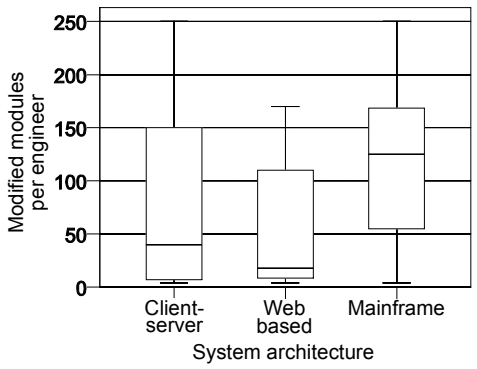


Figure 2. Modified modules per engineer stratified by system architecture

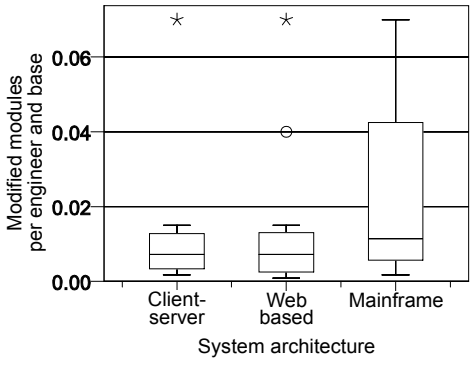


Figure 3. Modified modules per engineer and base stratified by system architecture

Table 5. Median values of modified modules per engineer (and base) stratified by process standardization

	Process standardization	Number of cases	Median	p-value (difference from standardized)	p-value (difference from not standardized)
Modified modules per engineer	Standardized	6	118.3	-	29%
	Work in progress	8	14.0	11%	89%
	Not standardized	1	7.1	29%	-
	Total	15	40.0	-	-
Modified modules per engineer and base	Standardized	6	0.026	-	57%
	Work in progress	8	0.005	2%	100%
	Not standardized	1	0.007	57%	-
	Total	15	0.008	-	-

Table 6. The number of base modules, the number of modified modules, and the number of engineers stratified by process standardization

Process standardization		Number of base modules	Number of modified modules	Number of engineers
Standardized	Median	3000	400	6
	Number of cases	13	10	11
Work in progress	Median	4000	100	10
	Number of cases	15	12	17
Not standardized	Median	1000	5	6
	Number of cases	7	3	7
Total	Median	3000	180	10
	Number of cases	35	25	35

modules increased the number of modified modules (see section 3.2). However, relative difference of the number of base modules between mainframe system and others is smaller than the number of modified modules. This reinforces the analysis result that work efficiency is different for system architecture.

3.4 Relationship to Process Standardization

Figure 4 illustrates a relationship between process standardization and modified modules per engineer, and Table 5 shows median values of them and statistical test results for their differences. In Figure 4, when process is standardized, modified modules per engineer is the highest, although the distribution is wide. The median value of "standardized" is 8.5 times larger than "work in progress," as shown in Table 5. But they are not statistically different.

Same tendency was observed in modified modules per engineer and base. Figure 5 shows a relationship between process standardization and modified modules per engineer and base, and Table 5 presents median values of them and statistical test results. In Figure 5, the values of "standardized" are larger than "work in progress." In Table 5, the median value of "standardized" is 5.2 times larger than "work in progress," and there is statistically difference between them. The results indicate process standardization is related to work efficiency.

We checked the number of base modules, the number of modified modules, and the number of modules, stratifying the dataset according to process standardization. In Table 6, between "standardized" and "work in progress," there is 0.6 times difference in the

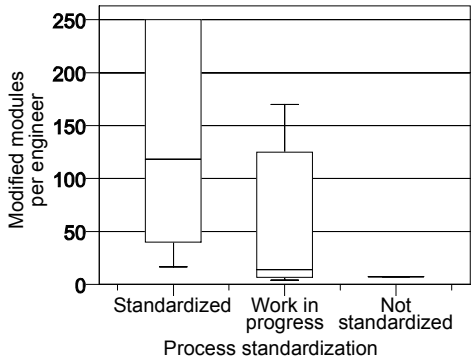


Figure 4. Modified modules per engineer stratified by process standardization

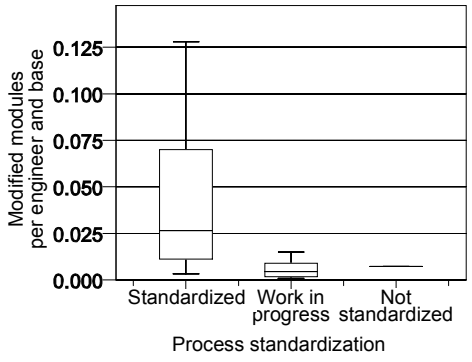


Figure 5. Modified modules per engineer and base stratified by process standardization

Table 7. Relationships to productivity factors

		Human factor	Problem factor	Process factor	Product factor	Resource factor	Tool factor
Modified modules per engineer	ρ	-0.09	0.38	0.20	0.32	-0.02	0.07
	p-value	76%	20%	51%	29%	96%	81%
	Number of cases	13	13	13	13	13	13
Modified modules per engineer and base	ρ	-0.26	-0.10	-0.23	-0.01	0.06	0.00
	p-value	39%	74%	46%	97%	85%	100%
	Number of cases	13	13	13	13	13	13

number of engineers, and 0.75 times difference in the number of base modules. Nevertheless, there is 4.0 times difference in the number of modified modules. So in this case, we do not have to care the positive correlation between the number of base modules and the number of modified modules (see section 3.2). This also supports the result that process standardization is related to work efficiency.

When stratifying the dataset by system architecture, both modified modules per engineer and modified modules per engineer and base are higher in process standardized organizations (except for modified modules per engineer in mainframe systems). Similarly, when stratifying the dataset by process standardization, they are higher in mainframe systems. Although we should be care that there are a few cases in each group after stratifying, both system architecture and process standardization are considered to affect work efficiency.

3.5 Relationship to Productivity Factors

Table 7 shows correlations between productivity factors and modified modules per engineer (and base). When a value of productivity factor is smaller, condition is more severe. That is, positive correlation indicates work efficiency decreases when the condition of the factor is severe.

Although the correlation between problem factor and modified modules per engineer is relatively higher (ρ = 0.38; not significant), the correlation to modified modules per engineer and base is low (ρ = -0.10). So, we cannot conclude problem factor affects work efficiency. Other factors have low correlation, and they are not statistically significant. In addition, their correlations are inconsistent between modified modules per engineer and modified modules per engineer and base. Namely, we did not observe explicit influences of productivity factors on work efficiency.

4. RELATED WORKS

Except for process standardization, some researches analyzed work efficiency factors on software maintenance. Jørgensen [4] analyzed software company dataset, and showed that work efficiency is not affected by the number of base modules and programming language. Ahn et al. [1] used variables which are similar to the productivity factors in a software maintenance effort estimation model. However, past researches did not clarified effect of process standardization. This is because they used dataset collected from a few companies, and it made the analysis of process standardization effect difficult.

There are very few reports or researches which analyzed cross-company software maintenance dataset. Japan Users Association of Information Systems (JUAS) and Ministry of Economy, Trade and Industry used the cross-company dataset, and showed work efficiency (maintenance cases per engineer) stratified by business sector [3]. However they did not clarified the relationship between process standardization and work efficiency. Yokota [7] showed standardizing maintenance process is effective for work improvement, based on questionnaire data. But quantitative data analysis was not performed.

5. CONCLUSIONS

In this research, to establish a benchmark for software maintenance efficiency, we analyzed software maintenance data collected from 83 organizations. Modified modules per engineer is regarded as work efficiency index. We compared work efficiency by stratifying the dataset, and clarified process standardization status is related to work efficiency (modified modules per engineer). Our future work is collecting more data and analyzing it to enhance reliability of the results.

6. ACKNOWLEDGMENTS

This work is being conducted as a part of the StagE project, The Development of Next-Generation IT Infrastructure, and Grant-in-aid for Young Scientists (B), 22700034, 2010, supported by the Ministry of Education, Culture, Sports, Science and Technology.

7. REFERENCES

[1] Ahn, Y., Suh, J., Kim, S., and Kim, H. 2003. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance: Research and Practice*, 15, 2, 71-85.

[2] Economic Research Association. http://www.zai-keicho.or.jp/about/english.php

[3] Japan Users Association of Information Systems 2008. Software Metric Survey 2008. Japan Users Association of Information Systems.

[4] Jørgensen, M. 1995. Experience With the Accuracy of Software Maintenance Task Effort Prediction Models. *IEEE Transactions on Software Engineering*, 21, 8, 674-681.

[5] Lokan, C., Wright, T., Hill, P, and Stringer, M. 2001. Organizational Benchmarking Using the ISBSG Data Repository. *IEEE Software*, 18, 5, 26-32.

[6] Software Evolution Research Consortium 1995. Software Evolution Research Consortium Report, Software Evolution Research Consortium.

[7] Yokota, T. 2003. An Assessment for Software Maintenance Environment and its Improvement Examples. *Journal of the Society of Project Management*, 5, 2, 40-44.

A Case Study of Committers' Activities on the Bug Fixing Process in the Eclipse Project

Anakorn Jongyindee[†] Masao Ohira[‡] Akinori Ihara[‡] Ken-ichi Matsumoto[‡]
[†]Kasetsart University
50 Ngam Wong Wan Rd, Chatuchak
Bangkok, Thailand
b5105896@ku.ac.th

Nara Institute of Science and Technology
8916-5, Takayama, Ikoma
Nara, Japan
{masao, akinori-i, matumoto} @ is.naist.jp

ABSTRACT

There are many roles to play in the bug fixing process in an open source software development. A developer called "Committer" who has permission to submit the patch into the software repository plays a major role in this process and holds a key to the success of the project. In this work, we have observed committers' activities from the Eclipse-Platform bug tracking system and version archives. Despite the importance of committer's activities, we suspected that sometimes committers can make mistake, which have a negative consequence to the bug fixing process. Therefore, our research focused on studying the consequences of committers' activities to this process. We collected committers' history data and evaluated each of them by comparing the more cautious to less cautious committers. From our results, we would like to create a clear understanding on committers' activities and their consequences on the bug fixing process in order to find a better way to improve the bug fixing process in OSS projects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Performance measures*; D.2.9 [**Software Engineering**]: Management —*Productivity, Programming teams, Software quality assurance (SQA)*

General Terms

Human Factors

Keywords

open source software (OSS), committer, bug fixing process

1. INTRODUCTION

Open Source Software (OSS) has been attracting a great deal of attention from a variety of areas as an alternative way to use and develop software. Currently, OSS products has a large impact on not only the end-users, but the manufacturers of mobile devices as well, since they need to exploit OSS to produce their end products. For instance, Linux, the OSS that sets the strong roots for other operating systems, and Google's Android's operating system that, based on this OSS, has become the best-selling smart phone platform. As OSS become more common and popular among us, however, its projects are faced with a big challenge to their quality assurance activities. Due to their growing user base, especially large OSS projects such as the Mozilla and Eclipse projects receive considerable amount of bug reports from the users on a daily basis [13] (e.g., several hundred bug reports are posted to the Bugzilla database of the Mozilla project every day). Therefore, OSS projects require an effective way of dealing with the large number of bug reports.

In an OSS project, a bug is fixed through the bug fixing process [19] which is a process of fixing the bug from the time a bug was reported in the project until the patches for fixing the bug have been submitted into a software repository such as Bugzilla. Each bug report in this process is passed through one or more developers who play different roles before the process is closed.

In this study, we focused on developers who have the privilege to submit patches into the software repository, called *Committers*. This group of developers play major roles in the bug fixing process [7]. Their main task is to review (and sometimes edit) patches posted by other developers and submit them into the software repository. Some of them also perform other tasks including bug resolution and bug reports management. Using Concurrent Versions System (CVS) and Bug Tracking System (BTS), they resolve bugs, join discussions about bugs, verify fixed bugs by developers, close bug reports, and so forth. As just described, committers' activities are vital for sustaining and improving the quality of OSS products.

However, committers are not always perfect. They sometimes make mistakes. For instance, they can uncautiously verify a bug report already resolved by a developer and close the bug report, creating another bug report for the same bug again (i.e., reopen bug). In this paper we are interested in creating a better understanding of committers' activities and their consequences on the bug fixing process in order to find a better way to improve the bug fixing process in OSS projects.

Selecting the Eclipse-Platform's Version archives (CVS) and Bug Tracking System (BTS) as the information source of our case study, we ask the following research question.

RQ: What are the consequences of the committer's activities to the bug fixing process? In this question, we studied the committer's activities and their effects to the bug fixing process. Then we compared the consequence of more cautious committer's activities to the lesser one.
Through answering the research question, we provide contributions in this paper as follows.

- To create a better understanding on a committersÕ activities and their consequences on the bug fixing process in order to find a better way to improve the bug fixing process in OSS projects.

In what follows, we introduce our related works in Section 2 and extraction method in Section 3. Section 4 shows the results and process on how we answered our research question. Additional interesting results that we are able to identify during this work are discussed further in Section 5. Section 6 describes our limitations and future work, and we summarize our study in Section 7.

2. RELATED WORK AND MOTIVATION

Most existing studies are focused on how to reduce the time to fix bugs since it has been gradually increasing, especially in large OSS projects. There are currently three promising approaches to improving the bug fixing process. In what follows, we describe the existing approaches and our motivation of this study.

2.1 How to make a good bug report?

A good bug report contributes by reducing the time to fix bugs because it can help developers to quickly find, replicate, understand the bugs at hand. However, developers' information needs in the bug reports are often unsatisfied, since users do not know what type of information are required to fix a problem and so rarely articulate the problem on the software use as developers can fix it. For instance, users do not correctly report procedures to reproduce an error (e.g., sometimes they just say "This option does not work in my computer!"). Therefore, developers have to ask users to give more information again and again to identify and fix the error. If things go wrong, developers cannot confirm the error and then leave it unresolved reluctantly.

In order to improve cooperation on a bug report between developers and users, many studies [3, 4, 6, 10] have interviewed OSS developers and users to understand the information needed to fix the bug. For example, through interviews with over 150 developers and 300 reporters of the Apache, Eclipse and Mozilla projects, Bettenburg et al. [BettenburgFSE2008] have found that steps to reproduce and stack traces are most useful in bug reports.

2.2 Duplicate bug detection

Users often report the same problem that was reported by another user in the past or that has already been fixed by developers. Developers also sometimes try to resolve the same problem which had been resolved in other times. This can happen because there are a large number of bug reports in the bug tracking system. Both the users and developers cannot be aware of all the reported bugs though the searching function that was provided to find bugs reported in the past. In this manner, the same bugs are duplicated in BTS which result in wasting developers' time and efforts.

To avoid duplicate bugs in BTS, several studies [18, 15, 20] have tried to detect duplicate bug reports automatically. For example, Wang et al. [20] present an approach to detect duplicate bugs based on a natural language processing techniques.

2.3 Re-opening and reassigned bugs

Even if a bug fixing task is assigned to a developer, it may not be completed by the developer which is then reassigned (*tossed* [13]) to other developers. This often happens because a trigger assigned a bug fixing task to an inappropriate developer who does not have sufficient knowledge and skill to complete the task. In the Eclipse and Mozilla projects, 37% to 44% of bugs are reassigned to another developer [13]. Preventing the bug tossing (assigning a bug fixing task to appropriate developers) is very effective in reducing the time to fix bugs.

Several approaches [1, 14, 13, 9, 8] exist in this topic. For instance, Anvik et al. [1] proposed an approach to assign a bug to an appropriate developer based on past bug reports with natural language processing. Jeong et al. [13] also tried to establish a method for the bug assignment based on a social graph which reflects on social relationships among developers in the bug assignment. Other approaches involved in achieving better understanding on why reassignment occurs many times [9] and in creating a method to predict which bugs will be reopened or get fixed without being reopened [8].

2.4 Cautious and uncautious committers

In general, a dedicated developer is nominated or elected to become a committer in an OSS project [12]. The OSS project carefully selects a developer as a committer candidate since committers play the important roles as described earlier. It takes one or two years to be a committer. A developer who wish to be a committer has to keep showing devoted activities to the project for a considerable period of time. Due to this promotion process, many of developers leave the project within a year and the OSS projects are always faced with the difficulty in having to find more committers who can greatly contribute to the bug fixing process.

In order to find a way to increase committers in OSS projects, Fujita et al. [7] have examined activities of developers and committers in the PostgreSQL project and tried to identify promising developers who were making a significant contribution equivalent to committers and potentially should be nominated to be committers in the future. Although the study found interesting aspects of committer candidates, it still adhered the current practice of existing committers and their promotion process in OSS project. In fact, one of their conclusions was that long-term participation in a project was the most important aspect to become a committer.

Different from [7], in this paper we have studied the committers' activities and their consequences. Our basic assumption on committers is that committers are not always perfect and sometimes make mistakes because they are also human-beings. Some of them might uncautiously verified a fixed bug by developers, creating reopened bug reports in the future. They also might uncautiously reviewed and accepted a patch posted by developers to fix a bug and then committed it into the repository such as CVS that would bring reopen and another bug reports. In this study we are

interested in having a clear understanding of committers' activities and the consequence to the bug fixing process.

3. EXTRACTION METHOD

In this section, we describe how we extracted information of committers' activities from the Eclipse-Platform project for our case study. Firstly, we describe how records of committer's activities are preserved in OSS the development. Then we introduce our method of extracting the information to observe committer's support activities and main activities respectively.

3.1 Committers' Activities

When developers are involved in the bug fixing process, their action are recorded in many formats. In common with many other OSS projects, Eclipse-Platform had chosen Bugzilla as their BTS where records each committer's support activities such as patch reviews and status changes (e.g., sometimes developers check the resolution of a bug and mark the bug's status to "VERIFIED" or "CLOSED"[19]) are stored. Developers in the project can see the information of the database in the HTML form through a web browser.

The project also used CVS to keep track of their commit history which is recorded in the plain text format called "commit log". By combining the information from both CVS and BTS, we are able to observe when/why/how each developer had contributed in the bug fixing process.

Using both the commit log and the Bugzilla database as our data sets, we collected 85,387 bug report data on BTS and over 30,833 commit log data on CVS. As a result, we were able to captur activities of 2,584 different developers from October 2001 until January 2010.

In order to archive our results, first, we need to identify who the committers are, excluding them from thousand of regular developers in the projects. To our knowledge, there is no specific activity that can decide whether one developer is a committer or not. [7] suggests only a rough description of how they extract their committer list. Based from their work, they had defined a developer as a committer, who has a privilege to submit a patch to the software repository. By using this definition, we managed to extract our list of committer's names. When a committer makes a patch commitment, their action is captured and their name is recorded in commit log's author field. By using regular expressions to scan every CVS line, we are able to identified 74 privilege developer names to create our list of committers' names.

From the committer list, we need to collect each committer's behavior data. We describe the procedures in the following separate subsections. In the first subsection we described how we collect each committer's support jobs in BTS, that is, how we studied the footprint of their support activities left on Bugzilla[1]. The second section describes how we observe committer's main jobs in CVS, that is, how we collects their patch commitment footprint left on CVS data.

3.2 Observing Support Activities From BTS

[1]Only some committers use the same account name in CVS and BTS. In order to map between their two accounts, we used automated method to find an exact-name-match for the committer who use same account name in both records. For those who did not, we have no choice but to map each committer's name manually.

In the bug tracking system, each reported bug is identified by a number called bug-id, attached with other data such as bug priority, bug status history, developer's comment, and so on. Each bug has its own current status varying from NEW, ASSIGNED, VERIFIED or CLOSED. Some bug status have its own resolution to indicate what happened to the bug such as FIXED, INVALID, and DUPLICATED[19].

Bug status history are used in many researches as a very useful source of information. Researchers can test a hypothesis[7], create prediction models[16], or performs statistical analysis[11]. In this paper, when we describe the bug status that have changed from one to the others in the bug history, for better clarifications, we present the bug's history in the form of bug status patterns. We use "⇒" for separation between bug status. Time dimension flow from left to right of the patterns. "..." Symbols represent any or many bug status changed and we use "()" to show the resolution of the bug status if it is exist. These bug status pattern can start from as simple as OPENED ⇒ NEW ⇒ ASSIGNED ⇒ RESOLVED (FIXED) to the more complex pattern such as OPENED ⇒ NEW ⇒ ASSIGNED⇒ RESOLVED (INVALID) ⇒ REOPENED ⇒ ASSIGNED ⇒ RESOLVED (WORKSFORME). For the first pattern, we can observe that the bug has been assigned only once before its resolved. This type of the pattern usually leads to short or normal bug life cycle while the more complex one often leads to longer bug life cycle.

We are able to observe and collect each committer's support activities based on this bug status patterns. We could identified 52,013 of 85,387 bug reports that were involved by our committers with 4,941 of 30,833 difference bug status patterns.

3.3 Observing Main Activities From CVS

For the Eclipse-Platform's commit log (from CVS), we wish we could have observed the committer's main behaviors solely from the commit log as we did in the Bugzilla bug history. Unfortunately, from this commit log we can only have a narrow vision of committer's activities; It captures only revision numbers, date of commit and some information about source code changes. The description about each change is solely depend on committer's opinion. This description field has no centralized format and often recorded in an ill-organized pattern (some of them are empty sometimes).

Due to these inconsistencies, the CVS description is not adequate to judge each committer's behavior. In order to overcome this problem, we decided to adapt T. Zimmerman's[17] approach to our study. Their approach has suggested that, despite its inconsistencies, sometimes committer has mentioned bug-id in the CVS description. By using bug-id as a trails, we identified it as a *links* from the CVS repository to the bug database. From these *links* we can look further into the BTS database where we have wider behavior information to study. The technique on finding these *links* has been used in many works in this field (e.g., [2],[17] and [5] has described these links and illustrated it clearly.). By adapting the Zimmerman's approach to our study, we managed to identify 1,193 links from our commit log. Thus, we could collect each committer's main activities.

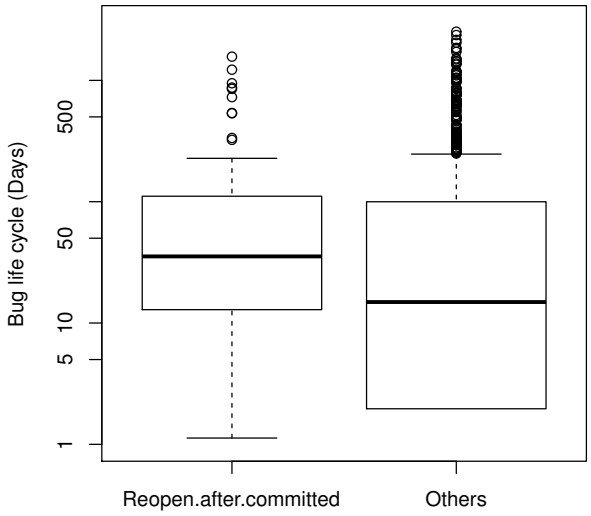
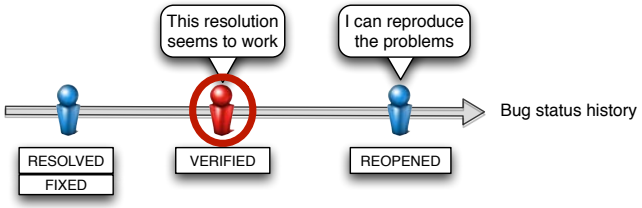
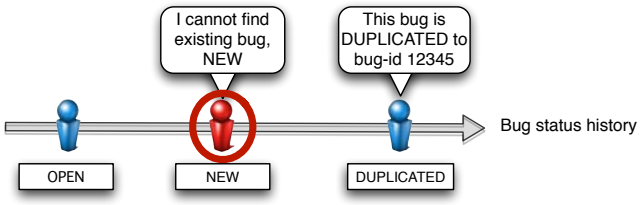


Figure 1: Box plot comparing the life cycle of *Reopen-after-committed* bug and normal bug



(a) *Reopen-after-verified/closed* pattern



(b) *Duplicate-after-new* pattern

Figure 2: *Bad-status-patterns* observed in our study

4. RESULTS

RQ: What are the consequences of the committer's activities to the bug fixing process?

By focusing on the consequences of committer's activities, we separated the results for this question into two parts: the consequences of a committer's main activities and the consequences of a committer's support activities.

4.1 Consequences of Main Activities

APPROACH.

As mentioned above, we suspect that when a committer un-cautiously committed the patch that fixed the bug, this bug might be reopened later to be resolved again. After compare the bug life cycle of these *Reopen-after-committed* bugs with the other bugs that did not reopened from 1,193 links found from CVS and BTS, we identified 140 bugs that had been reopened after committers committed the patches.

FINDING.

The result are shown in Fig 1, we can see that when a bug is reopened after patches were committed, the bug tends to have longer life cycles.

4.2 Consequences of Support Activities

APPROACH.

As explained earlier, we collected committer's support activities by observing the bug status history and rewrote these status history in the form of patterns. In order to study the consequence of these patterns to the bug fixing process, we had randomly chosen these patterns to inspect manually. By focusing only on patterns related to committer's jobs, we can identify two status patterns that potentially have negative effects on the bug fixing process.

The first pattern shown in Fig 2 (a) we call *Reopen-after-verified/closed* pattern represents that bugs have been REOPENED after they had been marked as VERIFIED or CLOSED (e.g., ... ⇒ VERIFIED (FIXED) ⇒ REOPENED or ... ⇒ CLOSED (FIXED) ⇒ REOPENED). We suspect that this pattern occurs when committers do not cautiously check a patch before they changed status to VERIFIED. So this bug has to be reopened later (in worse case, the bug has been left over and no one reopen it).

The second pattern shown in Fig 2 (b) called *Invalid/Duplicated-after-new* indicates that bugs have been detected as INVALID or DUPLICATE after they had been marked as NEW (e.g., NEW ⇒ ASSIGNED ⇒ RESOLVED (INVALID) or DUPLICATE). In this case, we suspects that a developer who marked NEW made a mistake. This bug is not *new* but was actually duplicated or was invalid (sometimes invalid mean it is not even a bug.). In this paper, we will use *Bad-status-pattern* to represent the two bug status patterns above. We suspect that the bug life cycle followed *Bad-status-pattern* might be longer compared to the bugs without such the bad pattern. Thus, these bugs waste more developers' time and efforts.

FINDING.

After extracting above patterns from every bugs in Bugzilla, we were able to identify 405 bugs that followed *Reopen-after-verified/closed* patterns with 289 bugs that has been marked as *VERIFIED* by committers. The other bugs were verified or closed by other developers that had verified permission but does not have commit permission. Thus, they are not committers. And for 696 bugs that followed Duplicate/Invalid-after-new patterns, there were 470 patterns that had been marked as new by our committers.

By using only bug reports that were involved with the committers (total of 52,013 bug reports), we used a box plot to compare the bug life cycle between the bugs that followed

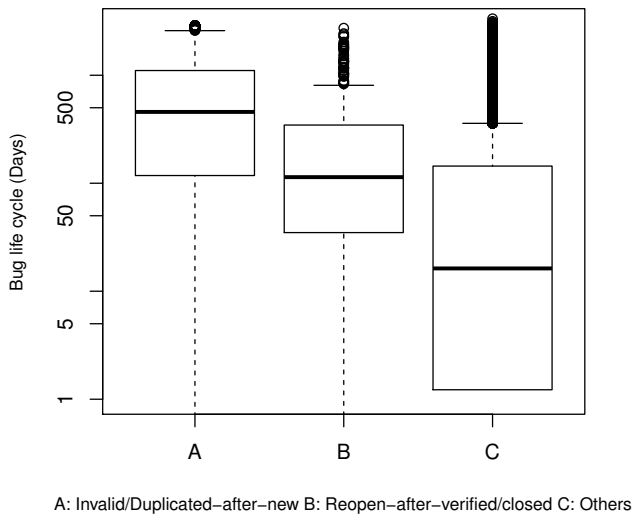


Figure 3: Box plot comparing the bug's life cycle that followed *Invalid/Duplicated-after-new* patterns and *Reopen-after-verified/closed* patterns with normal bugs

Reopen-after-verified/closed patterns, *Invalid/Duplicated-after-new* patterns , to other bugs that our committer has been involved. There were 51,254 bug reports that did not follow *Bad-status-pattern*. The results showed significant different number of days between these bugs. Unsurprisingly, the bugs where a committer made a mistake has longer bug life cycle.

5. DISCUSSION

In this section, we discuss the results from our research question and additional results we can find during our research.

5.1 Committer's Uncautious Activities: Negative Effects On The Bug Fixing Process

From our research question's results, we suspect some activities that potentially has a negative consequence and we compared it with normal activities. We can identify that the patch that has been *Reopen-after-committed* or the bugs that followed *Bad-status-pattern*s have longer life cycles compare to the other bugs. From this result, we wish to make a humbly suggestion to an OSS's committer to be aware of their importance to the bug fixing process. When they are not cautiously doing their jobs, they might extend the bug life cycles.

5.2 Additional Results: Not all Reopend Bug Are Bad

When we observed the bug status patterns in BTS, against popular beliefs, we have found that not all reopens have negative effects to the project. In the Eclipse-Platform, we can identified that there are 6 types of reopens. Each type has different impact to the bug life cycle. Some patterns showed that reopens can have positive effect such as ... ⇒ RESOLVED(WONTFIX) ⇒ REOPENED ⇒ ASSIGNED ⇒ RESOLVED(FIXED). We manually observed this patterns, found that some developers simply change the bug resolution to WONTFIX because they did not have enough knowledge to fix it, which later has been REOPENED and fixed by other developer. Another example is the reopened-after-later pattern (... ⇒ RESOLVED(LATER) ⇒ REOPENED), this reopen is actually intended, LATER resolutions usually mean "this bug must wait for the new patch to be fixed", "this is not the target milestones", or "need some minor tweaks later". We want to make suggestion to other researchers who use bug status pattern in their work to be aware of these types of reopen and their different impacts.

6. LIMITATION AND FUTURE WORK

In our extraction process, we have collected each committer's patch commitment activities by the observed CVS description that had a link to the bug database. Unfortunately, this collected links number are considered to be small in portions compared to all of the activities in the CVS. From 30,833 commits in the commit log, we can identify only 1,193 links with a unique bug-id. To reduce the bias resulted from a sample size, our goal was to capture the largest representation of population as we can. As we explained earlier, we (hopefully) archived this goal by adapting [17] approach in order to overcome this limitations.

Please be noted that the results from this research is focus only on the Eclipse-Platform's development community. This community structure are well-organized and have full-time workers (like commercial development community). Different OSS communities can have different structure which will reflects in different results from the same approach.

In order to observe variation of the results reflected from the different communities, our future works will apply the approaches used in this research to another OSS projects and we would like to look deeper into existing committers themselves to find the committer candidates who will become "**good**" committers. We also would like to observe another developer's role in the OSS's bug fixing process, and hopefully received a useful results that can benefit all OSS communities.

7. CONCLUSION

In this paper, we have focused on developers who played a major role in bug fixing process called *Committers*. We suspected that, when the bugs are taken care by more cautious developers (and verified by cautious committer), their life cycles might be shorter. We identified committer's activities that have different consequences to the bug fixing process. Our findings can be summarized as follows:

- We ware able to determined the patches that have been reopened after it was committed and showed that, when the committer committed the patch and that patch had to be reopened later, it tends to have longer life cycle.
- We categorized severals bug status patterns, showed that when the bugs have its status followed *Bad-status-pattern*, they have longer bug life cycle than the bugs with other patterns.

8. ACKNOWLEDGMENT

The first author is grateful to the internship program cooperated and supported between Kasetsart University, Thailand, and Nara Institute of Science and Technology, JAPAN. It bestows a grant as well as an opportunity for undergraduate student to achieve a wealth experience in abroad graduated school research.

This work is also conducted as part of StagE Project (the Development of Next Generation IT Infrastructure), Grant-in-Aid for Scientific Research (B), 23300009, 2011, and Grant-in-aid for Young Scientists (B), 22700033, 2011 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

9. REFERENCES

[1] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering (ICSE'06)*, pages 361–370, 2006.

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *SIGSOFT'10/FSE-18: Proceedings of the 16th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 2010.

[3] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT'08/FSE-16)*, pages 308–318, 2008.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, 2008.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.

[6] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work (CSCW'10)*, pages 301–310, 2010.

[7] S. Fujita, M. Ohira, A. Ihara, and K. ichi Matsumoto. An analysis of committers toward improving the patch review process in oss development. In *Supplementary Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering (ISSRE2010)*, pages 369–374, November 2010.

[8] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10) - Volume 1*, pages 495–504, 2010.

[9] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work (CSCW'11)*, pages 395–404, 2011.

[10] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07)*, pages 34–43, 2007.

[11] A. Ihara, M. Ohira, and K. ichi Matsumoto. An analysis method for improvinga bug modification processin open source development. In *In 10th international workshop on principles of software evolution (IWPSE'09)*, pages 135 – 143. ACM, August 2009. Amsterdam,The Netherland.

[12] C. Jensen and W. Scacchi. Role migration and advancement processes in ossd projects: A comparative case study. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 364–374, 2007.

[13] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE'09)*, pages 111–120, 2009.

[14] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR'09)*, pages 131–140, 2009.

[15] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 499–510, 2007.

[16] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. ichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *the 17th Working Conference on Reverse Engineering (WCRE 2010)*, pages 249–258. IEEE, IEEE Computer Society, October 2010.

[17] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the Second International Workshop on Mining Software Repositories*, pages 24–28, May 2005.

[18] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10) - Volume 1*, pages 45–54, 2010.

[19] The Bugzilla Team. The Bugzilla Guide: 5.4. Life Cycle of a Bug. http://www.bugzilla.org/docs/3.4/en/html/Bugzilla-Guide.html.

[20] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 461–470, 2008.

Refactoring the Refactoring

Inferring Restructuring Operations on Logical Structure of Java Source Code

Hideaki Hata
Osaka University
Osaka, Japan
h-hata@ist.osaka-u.ac.jp

Osamu Mizuno
Kyoto Institute of Technology
Kyoto, Japan
o-mizuno@kit.ac.jp

Tohru Kikuno
Osaka University
Osaka, Japan
kikuno@ist.osaka-u.ac.jp

ABSTRACT

Restructuring source code structure, such as moving and renaming classes or methods, are inevitable activities in software development, and are recommended for the improvements of maintainability. However, it has been not easy to understand properly what logical structural changes occur. This is because we can obtain only file-level and line-level information from source code management systems about changes. This paper presents a technique of such inferring restructuring operations on logical structure of Java source code. For inferring structural change operations, the core part is mapping elements between two revisions. Previous related studies tackle this problem based on the analysis of subgraph similarity, which takes lots of time. We find match candidates based on the similarity of element contents and identify matches with Bayesian inference based on empirical data. We report the result of empirical evaluation of our technique with open source software projects from Android and Eclipse. We see that our technique identify most element matches correctly and provide appropriate operations, and it took only a few seconds to analyze entire history of each project.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Management

Keywords

change analysis, refactoring, software evolution

1. INTRODUCTION

Software evolves dynamically. While developing and maintaining source code, restructuring source code structure including moving and renaming program elements is a common practice. From an empirical study, Murphy-Hill et al. reported that though pure refactorings (root-canal refactorings) rarely occurred, floss refactorings, which are refactorings with other types of programming activities, occurred frequently [18].

Though refactorings are recommended and restructuring source code occurs frequently, there is a problem on understanding those changes. Mailloux reported an experience of industrial software development from the initial implementation to several business phases [17]. In this report, it is described how configurations, bugs, changes, and so on were managed as the project grew. At the initial implementation, there was no change management. From the first release, informal one-to-one coaching and formal documentation began. As the project grew, an initial training was provided to developer team. However, it is reported that changes were so fast, then the initial training became obsolete. As seen in this experience report, it has been not easy to understand changes.

This paper presents a technique of inferring change operations, especially restructuring change operations on logical structure of Java source code. We target moving and renaming of program elements, such as packages, classes, fields, constructors, and methods. There are several studies providing change operations between two revisions. One approach is a record-and-replay technique [3, 12]. Though these approach are able to provide accurate change operations, it is not always possible to record change operations because developers do not always use refactoring tools [18]. The other studies based on matching techniques. Change operations are inferred based on identified program element matches. The objective of most previous studies were identifying what changes occur between two releases. Most approaches based on the subgraph isomorphism problem, which requires large time to analyze. Consequently, analyzing entire histories (every changes) is not practical because of large time consumption.

We propose a light-weight technique to overcome this limitation. Our technique do not directly infer restructuring based on finding subgraph isomorphism, but there are mainly two phases to infer restructuring. First, we identify matches between individual element using simple heuristics and then restructuring is inferred using Bayesian inference based on empirical data. Our technique is empirically evaluated with open source software projects in Android and Eclipse. From this evaluation, we see that our technique infers most change operations properly and takes only a few seconds to analyze entire history of each project.

The rest of this paper is organized as follows. Section 2 describes restructuring operations we target and clarifies that program element matching problem is the core of structure change operation inference. Section 3 discusses program element matching problem

Previous revision

```
package pck;

public class Cls_a {
 void mth_x () {
 }
}

public class Cls_b {
 void mth_y () {
 }
}
```

Changed revision

```
package pck;

public class Cls_a {
 void mth_p (int a) {
 }
}

public class Cls_c {
 void mth_q () {
 }
}
```

Figure 1: A change example between two revisions.

with related work. In Section 4, we explain our inference algorithm and evaluate our technique in Section 5. Finally, we conclude in Section 6.

2. RESTRUCTURING OPERATIONS

Motivation. Changes of software structure is inevitable and important, but it is difficult to understand such changes. In some paper, it is reported that the lack of change management is big risk of software quality and team management [17, 21]. Tools that help developers to understand restructuring changes should be required.

Definition. The problem addressed in this paper is proposing a technique that suggests restructuring operations applied to change software structure between two revisions of software. In source code, there are some structures, such as physical structures (directories and files), logical structures (packages, classes, methods, and so on), dependencies (define-use and overriding) [14]. This paper targets changes on logical structures. We treat *packages*, *classes*, *fields*, *constructors*, and *methods* as program elements in logical structures of Java source code. Targeted restructuring operations include *rename*, *move*, *hide*, and *unhide* following the terms in [22]. Rename means the identifier changes in this paper. So method renames is the changes of method signatures, that is changes on method name, parameters.

Overview. Figure 1 is an example of source code change. If a method mth_p is identified as a modified version of a method mth_x, that is, a match is found between the two methods, it is easy to interpret that the method is renamed from mth_x to mth_p and its parameter is changed. The method is not moved since the method exists in a same class Cls_a. If a match is found between a method mth_y and a method mth_q, it is easy to suggest that the method is renamed. However, this case is different from the previous case. The method mth_y exists in a class Cls_b and the method mth_q exists in a class Cls_c. If the class Cls_b and the class Cls_c are different one, we can recognize that the method is moved from the class Cls_b to Cls_c. If the class Cls_c is a renamed version of Cls_b, the method is not moved but just renamed. Changes on element names or parameters for *constructors* and *methods* are identified based on corresponding matches. To identify whether elements are moved or not, it is needed to investigate their parent element matches. In summary, we have to identify every program element matches.

3. PROGRAM ELEMENT MATCHING

The problem of identifying program element matches can be seen as a *link prediction problem* [6, 13], which is a problem of predicting the existence of a link between two nodes in a network as shown in Figure 2. For program elements, networks can be seen in logical structures, call dependency graphs, and so on. Existence of the link can be regarded as a match between two program elements. The link prediction problem can fall into two categories in accordance with the information used for prediction [13]:

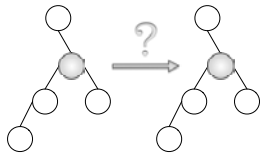


Figure 2: Link prediction problem.

Table 1: Information for program element matching

Studies	Topological info.	Node attributes
S. Kim et al. [16]	calls	name, text, metrics
Godfrey and Zou [7]	calls	name, metrics
Wu et al. [23]	calls	name
Fluri et al. [5]	structure	name
Dig et al. [2]	calls, structure	tokens
Weißgerber and Diehl [22]	structure	name, text
Prete et al. [20]	calls, structure	text
Xing and Stroulia [24]	structure	name
Dagenais and Robillard [1]	calls, structure	name

Topological-information-based methods: nearby nodes are similar or not.

Node-information-based methods: attributes of nodes are similar or not.

There are many studies inferring change operations. Table 1 summarizes previous studies based on the two information

Origin identification. S. Kim et al. applied several method matching techniques for *origin analysis* identifying renaming and moving to open source software projects, and evaluated the effectiveness of the techniques [16]. They reported that though *clone detection* yields an accuracy value 67.4, *function body diff* achieved 90.2. Splitting and merging of software entities are targeted by *origin analysis*. Godfrey and Zou proposed a technique of inferring such events based on matching procedures using multiple criteria including names, signatures, metric values, and call dependencies [7]. Splitting and merging correspondence analysis is also known as one-to-many and many-to-one matching. Wu et al. combined text similarity analysis on names and call dependency analysis for those method matching [23]. Fluri et al. proposed *change distilling*, a tree differencing algorithm [5]. *Change distilling* target not only method-level changes but also more fine-grained element changes. Name string similarities and tree similarities are calculated for matching.

Refactoring identification. Dig et al. proposed a technique for detecting refactorings based on identifying renaming packages, classes, methods, and moving methods [2]. Those changes are identified by using structural data, call-graph and tokens from entities. Weißgerber and Diehl presented a technique to detect changes that are likely to be refactorings [22]. Their matching technique is based on structure similarity and code clone analysis. M. Kim and Notkin proposed an approach, *LSdiff* to discover and represent systematic code

changes [14]. They intended to infer what changes are occurred based on analyzed structure differences. The matches are analyzed based on a set of predicates that describe program elements, their containment relationships, and their structural dependencies. *REFFINDER* proposed by Prete et al. extends predicate sets of *LSdiff* for identifying refactorings [20]. It supports sixty-three refactoring types. Though original *LSdiff* does not identify matches, *REFFINDER* does.

Framework usage changes. Xing and Stroulia proposed an approach for API-evolution support, called Diff-CatchUP [25]. On the step of change identification, UML-diff, which is based on name similarity and code dependency similarity of program elements [24], is used. After identifying changes, plausible API replacements are proposed. Dagenais and Robillard presented a technique to recommend adaptive changes for clients of framework code based on structure change analysis [1]. Their matching technique is based on structure similarity and out going call dependency similarity.

Discussion. As shown in Table 1, every study uses both methods for program element matching. As *topological-information–based methods* and *node-information-based methods* have different advantages and limitations, the combination of both methods is expected to achieve better results. Most studies mainly adopt *topological-information-based methods* and use *node-information-based methods* for program element matching.

Topological-information-based methods require unchanged or easily understandable neighborhood. Therefore, it is difficult to identify matching elements if there is no enough nearby elements or there are major changes. Wu et al. reported the limitations and insist that topological-information-based methods cannot be overcome them [23]. Fluri et al. reported following two limitations [5]:

- Mismatching can propagate. Not only mismatching for each targeting entity, correlate entities can be mismatched.
- The worst-case complexity increase. To decrease mismatching, complex algorithm is needed and this increase the worst-case complexity.

Because of these problems, previous techniques are not light-weight for analyzing entire histories. In addition, some studies report the difficulties of identifying moving operations [5, 24].

4. INFERENCE ALGORITHM

Our algorithm infers restructuring operations between two revisions of Java source code. Our algorithm consists of three parts: (1) finding candidates of program element matches, (2) identifying program element matches, (3) interpreting restructuring operations.

For program element matching in the part (1) and (2) , we use only node information because there are problems in using topological information as seen before.

4.1 Finding match candidates

We have proposed a system *Historage*[1] that can track program elements beyond renaming and moving [10, 11]. With this system, match candidates between program elements can be found based on the similarity of their text. We have found that it is possible to find most of match candidates in *methods*, *constructors*, and *fields* if contents are similar enough [11]. It is also possible to find match candidates between *classes* with the same technique.

The percentages of the same content in the size of smaller content (original or new) are calculated as text similarity values. In the previous study [11], we immediately identify matches based only on the similarity value. If the value is larger than or equal to 30%, elements can be regarded as matches, and if the value is less than 30%, elements are regarded as independent elements.

Though this procedure works relatively well, we miss some matches if the similarity values are low, which is caused by major modification. The next part of our algorithm is introduced for decreasing such missing.

4.2 Identifying matches

Though the high similarity value is a good evidence for finding element matches, we can use other node information as additional evidence. These additional evidence should be valuable especially when the similarity value is low. In this part, we identify matches based on Bayesian inference. If we can obtain additional node information X, we can calculate the posterior probability of matches as follows:

$$P(match|X) = \frac{P(match)P(X|match)}{P(X)}$$

We identify matches if the posterior probability $P(match|X)$ is greater than or equal to 50%, where $P(X) = P(match)P(X|match) + P(match)P(X|\texttt{not}match)$. To build not a project-specific model but a general model, we will determine parameters based on empirical investigation of several open source projects.

Prior probability. From the empirical study [11], if the similarity value is greater than or equal to 30%, more than 95% of matches are correct. There are not many matches if the similarity value is less than 30%. Based on this observation, we determine the prior probability as follows:

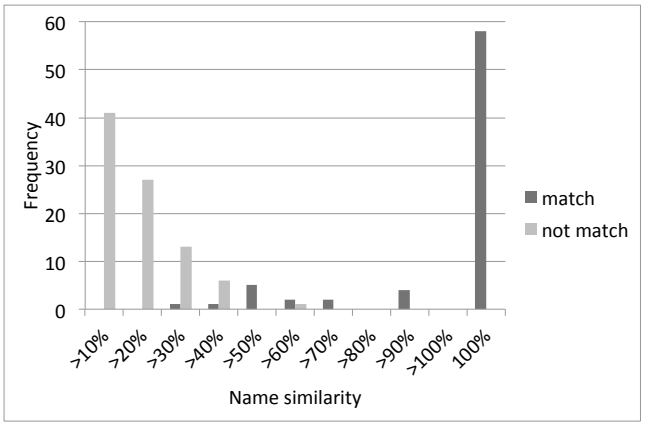
High text similarity: $P(match) = 95\%,\ P(\text{not } match) = 5\%$

Low text similarity: $P(match) = 20\%,\ P(\text{not } match) = 80\%$

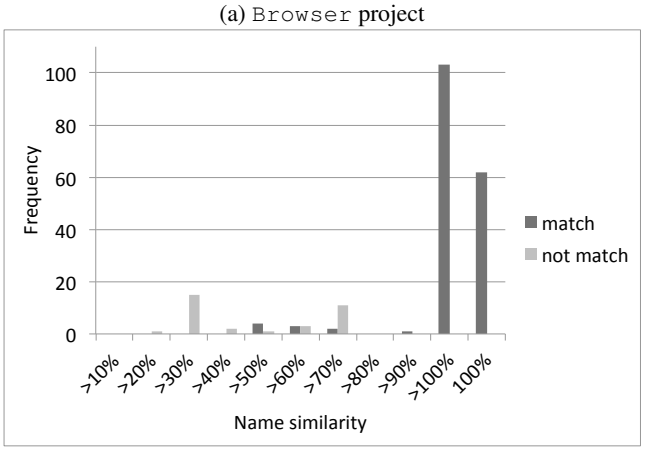
Evidence. For additional evidence, we collect following two node information: (i) names of program elements, (ii) existence of corresponding child elements. The similarity of names between two elements are widely used for matching [1,5,7,16,22–24] as seen in Table 1. The existence of corresponding child elements is an additional evidence from our observation. If there is a match between classes, which means the matched class is equal, there should be elements that exists in the previous and the new class. Though additional information (i) can be used for every program element types, (ii) can be used for only *class* and *package*.

(i) names of program elements. To compute the similarity of program element names (s_1 and s_2), we calculate their *longest com-*

[1]A tool to build Historage is available from `https://github.com/hdrky/git2historage`.



(a) `Browser` project



(b) `Xpand` project

Figure 3: Name similarity and matches.

mon subsequence (LCS). We adopt the following expression proposed in [7] for the name similarity:

$$\frac{length(LCS(s_1, s_2) * 2)}{length(s_1) + length(s_2)}$$

Based on the name similarity, we want to determine the parameter of $P(\text{name sim.}|match)$ and $P(\text{name sim.}|\text{not } match)$. We investigate four open source software projects (Browser, Phone, EMF Compare, Xpand) to see the relation of the name similarity and the existence of program element matches. Figure 3 shows the distribution of program element matches based on the name similarity in two projects, Browser and Xpand. Though most matches have higher name similarities (more than or equal to 70%), there are a few matches that have middle name similarities (40% to70%). Not match candidates have low name similarities (less than 40%) and middle name similarities. We found similar distribution on every project. Based on these observation, we determine the parameter as follows:

$$\begin{aligned} P(\text{name sim.}high|match) &= 0.85 \\ P(\text{name sim.}middle|match) &= 0.1 \\ P(\text{name sim.}low|match) &= 0.05 \\ P(\text{name sim.}high|\text{not } match) &= 0.05 \\ P(\text{name sim.}middle|\text{not } match) &= 0.15 \\ P(\text{name sim.}low|\text{not } match) &= 0.8 \end{aligned}$$

(ii) existence of corresponding child elements. We investigate the existence of corresponding child elements for program element match candidates. From empirical investigation, we observed that there are a few matches without corresponding child elements, and there are few cases for not matches with child elements. We determine the parameters as follows:

$$\begin{aligned} P(\text{exists child}|match) &= 0.9 \\ P(\text{not exists child}|match) &= 0.1 \\ P(\text{exists child}|\text{not } match) &= 0.05 \\ P(\text{not exists child}|\text{not } match) &= 0.95 \end{aligned}$$

Built model. The name similarity and the existence of corresponding child elements can be seen as independent features. Therefore we a built naive Bayes classifier as follows:

$$P(\text{name sim., child}|match) = P(\text{name sim.}|match)P(\text{child}|match)$$

Using determined parameters, we build a classifier model. Instead of the detail posterior probability values, we show that when our model identify match candidates as matches. For *methods*, *constructors*, and *fields*, which do not have child elements, matches are identified if any one of the following conditions is satisfied:

- Text similarity value is greater than or equal to 30%.
- Name similarity value is greater than or equal to 70%.

For *classes* and *packages*, matches are identified if any one of the following conditions is satisfied:

- Text similarity value is greater than or equal to 30% and name similarity value is greater than or equal to 70%.
- Text similarity value is greater than or equal to 30% and there are corresponding child elements.
- Name similarity value is greater than or equal to 40% and there are corresponding child elements.

Program element match identification begins for *methods*, *constructors* and *fields*. After identifying these matches, it is easy to know there are corresponding child elements for *classes*. Then we identify matches for *classes*. Finally, we identify matches of *packages*. As seen in our classify model, we use only node attribute information, which should fit our intuition.

4.3 Interpreting restructuring operations

After identifying every program element matches, we interpret restructuring operations. Renaming is easily known between matches. As seen in Section 2, moving can be identified after clarifying whether parent elements are same (matched) or different (not matched). Though some paper describes the limitations of identifying moving operations [5, 24], there is no such limitation in our technique. Now our technique support the following restructuring operations: move, rename, parameter change, access modifier change (hide or unhide).

Figure 4 presents an example of inferred restructuring operations. In a package `com.android.phone`, there are changes of two methods as follows:

```
Package com.android.phone
  PhoneApp.java
    Class PhoneApp
      Method displayCallScreen() -> private displayCallScreen(): 1. hide
  PhoneUtils.java -> CallNotifier.java
    Class PhoneUtils -> CallNotifier
      Method showIncomingCallUi() -> private showIncomingCall(): 2. move & rename & hide
```

Figure 4: An example of inferred restructuring operations.

Table 2: Target project data.

Project		Initial	Last	# Changes
Android	Browser	2008-10-21	2011-05-03	1,517
	Contacts	2008-10-21	2011-04-04	2,082
	Phone	2008-10-21	2011-05-31	2,253
Eclipse	ECF	2004-12-03	2011-05-17	5,251
	EMF Compare	2007-04-03	2011-05-24	860
	Xpand	2007-11-10	2011-05-31	637

1. A method `displayCallScreen()` is **hidden** by being attached a `private` access modifier.

2. A method `showIncomingCallUi()` that existed in a class `PhoneUtils` is **moved** to a class `CallNotifier`, and is **renamed** `showIncomingCall`, and **hidden** by being attached a `private` access modifier.

These information should be useful for further research on fine-grained level, such as software evolution analysis, historical information based fault-prone/failure-prone module prediction, code clone management, and so on. Text-based output like Figure 4 may not be human readable. Appropriate visualization is one of required future work.

5. EVALUATION

In this section, we evaluate the accuracy of our technique and the performance of analysis time. We investigate the accuracy of identifying program element matches because inference of restructuring operations depends on this identification. As shown in Table 2, we select six open source software projects from Android and Eclipse to empirically evaluate our technique. Each project is developed more than two years and is committed (changed) about 500 to 5,000 times. These projects are written in Java and Git repositories are available.

5.1 Program element matching

We manually investigate every match candidates. To evaluate with *Recall* measure, we need to prepare reference set that should be collected from every potential matches, which is very hard task. Hence we measure *CRecall*, which is the number of identified correct matches divided by the number of correct matches in match candidates. From our large inspection, there are few cases that there are correct matches that are not identified as match candidates. *Precision* is the number of identified correct matches divided by the number of all identified matches. Since we manually identify correct matches, we may introduce some bias.

Table 3 summarizes the result of each project. The result is divided in two tables based on the text similarity values, that is, (a) for text similarity value is greater than or equal to 30% and (b) for text similarity value is less than 30%, because match identification with low text similarity is more difficult than with high text similarity. From Table 3 (a), which is a summary of matching with high text similarities, we can see that all matches are identified (every CRecall value is 1.00) and there are a few false positives (precision values range from 0.94 to 1.00). Table 3 (b) is a summary of matching with low text similarities. This case is relatively difficult because there are not many correct matches in entire match candidates as seen in the second and the fifth row of Table 3. As seen in Table 3 (b), there are no false positives (high precision except for matches of classes and packages in Contacts project). CRecall values ranges from 0 to 1.00. Matches with low text similarities can be identified based on the new evidence introduced in Section 4.2. We can see that naive Bayes inference framework works relatively well.

Table 3: Matching evaluation.

(a) text similarity $\geq 30\%$

	Method et al.†			Else‡		
Project	Num.*	CRec.	Prec.	Num.*	CRec.	Prec.
Browser	66/66	1.00	1.00	2/2	1.00	1.00
Contacts	162/162	1.00	1.00	10/10	1.00	1.00
Phone	102/102	1.00	1.00	6/6	1.00	1.00
ECF	894/897	1.00	1.00	125/125	1.00	1.00
EMF Compare	84/85	1.00	0.99	7/7	1.00	1.00
Xpand	178/189	1.00	0.94	2/2	1.00	1.00

(b) text similarity $< 30\%$

	Method et al.†			Else‡		
Project	Num.*	CRec.	Prec.	Num.*	CRec.	Prec.
Browser	8/79	0.88	1.00	0/13	–	–
Contacts	9/33	0.89	1.00	2/3	0	0
Phone	13/31	1.00	1.00	0/0	–	–
ECF	98/287	0.95	1.00	9/22	0.89	1.00
EMF Compare	20/38	0.70	1.00	0/2	–	–
Xpand	7/20	0.29	1.00	0/0	–	–

†: methods, constructors, and fields.
‡: classes and packages.
*: number of matches / number of match candidates.

Most of program element matches are identified well. To decrease false positives and false negatives for more improvement, there are some possible plans as follows:

- Readjust parameters of naive Bayes models.
- Find new evidences for naive Bayes models.
- Build different models for different program element types.

5.2 Performance

We show a rough comparison of performance based on reported papers. At this time, we do not compare our technique with pre-

Table 4: A rough comparison of performance

Studies	Time for one change analysis
Wu et al. [23]	a few minutes
Dig et al. [2]	several minutes
Prete et al. [20]	several seconds to a hour
Xing and Stroulia [24]	several seconds to a hour
Dagenais and Robillard [1]	several hours
This paper	**less than a second**

vious techniques with same hardware platforms and same target projects. Table 4 summarize the reported time for analyzing one change between two revisions. Note that though previous studies analyze medium-size or large-seize projects, our target projects are relatively small-size. Previous techniques require from a few minutes to several hours to analyze one change, which may be difficult to analyze entire histories (hundreds to thousands of changes). Our technique took less than a second for one change and only two seconds for entire changes of each project in Table 2. Major reason of this difference is that our technique consists of simple methods using only the information of elements (nodes), though other studies mainly use topological-information methods, which require high cost.

6. CONCLUSION

This paper presents a technique for inferring restructuring change operations on Java source code. Though previous techniques used topological information for program element matching, which have some limitations, our technique uses only node information. From empirical evaluation with six open source software projects from Android and Eclipse, it is revealed that our technique identifies program element matches with high accuracy. In addition, our technique require only a few seconds for analyzing entire histories.

With our technique, it is possible to analyze fine-grained and detail project histories. There are some possible application of this technique, such as fault-prone/failure-prone module prediction based on histories [8, 9, 19] and code clone history analysis [4, 15]. Improvements of program element matching and comparison with other techniques on same environment and same target projects are future work of our research.

7. ACKNOWLEDGMENTS

This research is supported by Grant-in-Aid for JSPS Fellows (No.23-4335).

8. REFERENCES

[1] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. ICSE '08, pages 481–490,2008.

[2] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. ECOOP '06, pages 404–428, 2006.

[3] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. ICSE '07, pages 427–436, 2007.

[4] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. ICSE '07, pages 158–167, 2007.

[5] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33:725–743, November 2007.

[6] L. Getoor and C. P. Diehl. Link mining: a survey. *SIGKDD Explor. Newsl.*, 7:3–12, December 2005.

[7] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31:166–181, February 2005.

[8] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. ICSM '05, pages 263–272, 2005.

[9] H. Hata, O. Mizuno, and T. Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Softw. Eng.*, 15:147–165, April 2010.

[10] H. Hata, O. Mizuno, and T. Kikuno. Reconstructing fine-grained versioning repositories with git for method-level bug prediction. IWESEP '10, pages 27–32, 2010.

[11] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. IWPSE-EVOL '11, pages 96–100, 2011.

[12] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support api evolution. ICSE '05, pages 274–283, 2005.

[13] H. Kashima, T. Kato, Y. Yamanishi, M. Sugiyama, and K. Tsuda. Link propagation: A fast semi-supervised learning algorithm for link prediction. SDM '09, pages 1099–1110, 2009.

[14] M. Kim and D. Notkin. Discovering and representing systematic code changes. ICSE '09, pages 309–319, 2009.

[15] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. ESEC/FSE-13, pages 187–196, 2005.

[16] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. WCRE '05, pages 143–152, 2005.

[17] M. Mailloux. Application frameworks: how they become your enemy. SPLASH '10, pages 115–122, 2010.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. ICSE '09, pages 287–297, 2009.

[19] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. ICSE '05, pages 284–292, 2005.

[20] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. ICSM '10, pages 1–10, 2010.

[21] J. Streit and M. Pizka. Why software quality improvement fails: (and how to succeed nevertheless). ICSE '11, pages 726–735, 2011.

[22] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. ASE '06, pages 231–240, 2006.

[23] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: a hybrid approach to identify framework evolution. ICSE '10, pages 325–334, 2010.

[24] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. ASE '05, pages 54–65, 2005.

[25] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Trans. Softw. Eng.*, 33:818–836, December 2007.

A Tool Support to Merge Similar Methods with a Cohesion Metric COB

Masakazu Ioka[1], Norihiro Yoshida[2], Tomoo Masai[1], Yoshiki Higo[1], Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University, Japan
{m-ioka, t-masai, higo, inoue}@ist.osaka-u.ac.jp
[2]Graduate School of Information Science, Nara Institute of Science and Technology, Japan
yoshida@is.naist.jp

ABSTRACT
"Form Template Method" is a refactoring pattern to merge similar Java methods with syntax differences. In this refactoring, developers divide target similar methods into a template method and primitive methods corresponding to the common part and the differences, respectively. In this proposal, we present a tool to show candidates of appropriate divisions between the common part and the differences based on a cohesion metric COB when developers select a pair of similar methods.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms
Experimentation

Keywords
Code Clone, Refactoring, Template Method Pattern

1. INTRODUCTION
Code clone is a code fragment that has identical or similar fragments to it in the source code[3]. It is regarded as one of factors that makes software maintenance more difficult. When developers modify a code fragment, they have to find code clones corresponding to modified code fragment.

Clone refactoring (i.e., merging code clones) is a disciplined technique to reduce code clones[1]. Syntactically identical code clones can be merged by straightforward technique (e.g., Pull-Up Method refactoring, Extract Method refactoring). On the other hand, when code clones have syntactic differences, it is necessary to extract those differences as new functions (e.g., Java method, C++ function) before merging.

"Form Template Method"[1] is a common refactoring to merge a similar pair of Java methods with syntactic differences. In the refactoring, developers divide similar methods into a template method and primitive methods corresponding to the common part and the differences, respectively.

However, it is difficult for developers to identify the common part and the difference from similar methods, and extract primitive methods so that each of them has a functionality that can be given suitable method name. Therefore, tool support is needed for desirable evolution of a pair of similar Java methods with syntactic differences. Juillerat et al. proposed an approach of automatic "Form Template Method"[2]. This approach detects different subtrees by comparing sequences of AST nodes which are generated by using post-order traversal, and shows only a candidate of "Form Template Method" for each pair of similar methods regardless of satisfying developers.

In this proposal, we present a tool to show candidates of appropriate divisions between the common part and the differences based on a cohesion metric Cohesion of Blocks (COB) [5] when developers select a pair of similar methods.

2. PROPOSED TOOL
First, we explain COB. COB is a cohesion metric between block statements in source code. It is proposed by Miyake et al. for identification of a set of block statements suitable for Extract Method refactoring. Proposed tool uses COB to see whether or not expanded fragments should be extracted as primitive methods, and then suggests pairs of code fragments with high COB as excellent candidates of pairs of primitive methods. The definition of metric COB is as follow:

$$COB = \frac{1}{b}\frac{1}{v}\sum_{j=1}^{v}\mu(V_j) \quad (0 \leq COB \leq 1)$$

where:

- b is the number of code blocks,
- v is the number of used variables in the method,
- V_j is j-th variable used in the method,
- $\mu(V_j)$ is the number of code blocks using variable V_j.

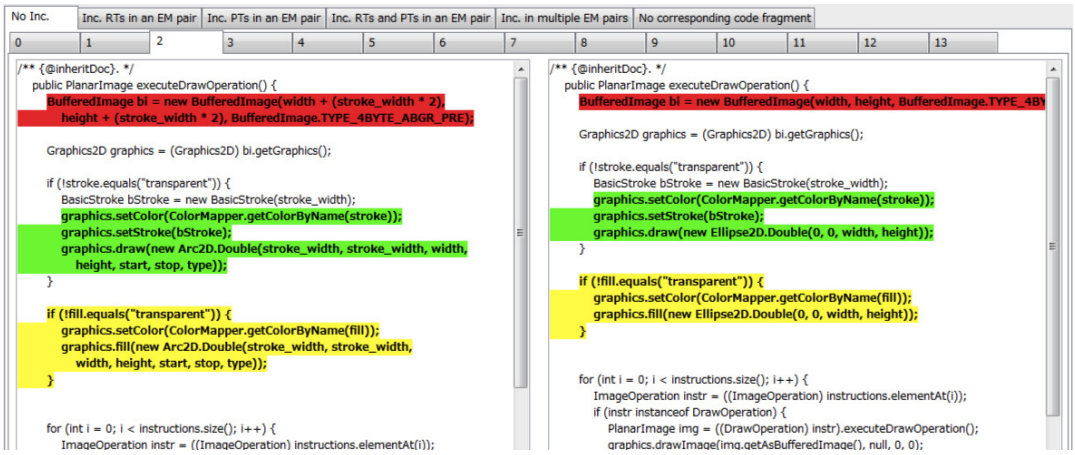


Figure 1: A screenshot of proposed tool

Next, we explain proposed tool. Proposed tool shows candidates of appropriate divisions between the common part and the differences when developers select a pair of similar methods.

Figure 1 shows a screenshot of proposed tool. Highlighted code fragments represent candidates of primitive methods. A pair of regions painted the same color means a pair of corresponding differences (i.e., candidates for primitive methods having the same name). Non-highlighted regions mean code fragments have an identical fragment to it in the corresponding method (i.e., candidate for a template method).

The steps to derive candidates of primitive are as follows.

1. Detect code fragments corresponding differences between Abstract Syntax Trees (ASTs) of given similar methods
2. Expand detected code fragments into code fragments including above, below or parent statements until all of expanded fragments are identified as extractable by the refactoring features of Eclipse JDT
3. Rank expanded fragments based on COB metric.

3. DEMONSTRATION

We applied proposed tool to the method pair in Figure 1.

This method pair is *genErrorHandler* method in CppCodeGenerator class and *genErrorHandler* method in JavaCodeGenerator class in ANTLR 2.7.4 [1]. Those methods are very similar to each other.

Figure 1 shows one of candidates for "Form Template Method" refactoring suggested by proposed tool. This candidate is highly ranked by COB metric with 0.86 because each difference shares values between blocks (COB of a method including only a block always indicates 1.0). Each highlighted code fragment has a single functionality that can be given suitable method name. Therefore, this can be considered as an excellent candidate for "Form Template Method" refactoring. Using proposed tool with COB based ranking, developers are possible to find appropriate candidates of "Form Template Method" refactoring easily.

[1]http://antlr.org/

4. SUMMARY AND FUTURE WORK

We proposed a tool to show candidates of template primitive methods for "Form Template Method" refactoring, and demonstrate it. As future work, we are planning to use cohesion metrics based on program slicing [4] instead of metric COB, because we expect ranking is better using program slicing. Also, we will implement the code transformation for "Form Template Method" refactoring using the Language Toolkit of Eclipse Project [2].

Acknowledgments

This work is partially supported by JSPS, Grant-in-Aid for Scientific Research (A) (21240002) and Grant-in-Aid for Research Activity start-up(22800040).

5. REFERENCES

[1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[2] N. Juillerat and B. Hirsbrunner. Toward an Implementation of the "Form Template Methodï£¡h Refactoring. In *Proc. of SCAM 2007*, pages 81–90, Paris, France, 2007.

[3] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[4] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Trans. Softw. Eng. Methodol.*, 17:2:1–2:27, December 2007.

[5] T. Miyake, Y. Higo, and K. Inoue. A software metric for identifying extract method candidates. *IEICE Trans. Inf.& Syst.(Japanese Edition)*, J92-D(7):1071–1073, 2009.

[2]http://www.eclipse.org/articles/Article-LTK/ltk.html

What did you do to our Data?!

An Improvement of Accuracy in Product Quality Prediction Using Imbalanced Project Data in Japan

Junya Debari
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
j-debari@ist.osaka-u.ac.jp

Tohru Kikuno
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
kikuno@ist.osaka-u.ac.jp

Nahomi Kikuchi
Oki Electric Industry Co., Ltd.
1-16-8, Chuou Warabi, Saitama, Japan
kikuchi386@oki.com

Masayuki Hirayama
Department of Electronics and Computer Science
College of Science and Technology, Nihon University
hirayama.masayuki@nihon-u.ac.jp

ABSTRACT

We constructed a prediction model with the data set provided by Software Engineering Center, Information-technology Promotion Agency, Japan(IPA/SEC) by applying the naive Bayesian classifier. The result showed that accuracy of predicting successful projects was 0.86. However, accuracy of predicting unsuccessful projects was 0.53, which was very low.

To find the reason for low accuracy, we analyzed the characteristics of the IPA/SEC dataset and revealed the following two factors that appeared to affect accuracy. (1) Incompleteness: 44.6% of the values in the data were missing. (2) Imbalance: The number of successful projects was three times that of unsuccessful projects.

We attempted to reduce the degree of incompleteness by mechanically minimizing the data that contained many missing values. The result of the preliminary experiment showed that the degree of incompleteness was reduced by our method. Moreover, thus the imbalance was simultaneously reduced.

On the basis of these preliminary experiments, from a given dataset, we deleted the data in which the number of missing values was larger than γ, where γ was a certain integer value that indicates the upper limit of the number of missing values. The result of the experimental evaluation showed that when γ was 6, accuracy of predicting unsuccessful projects was 0.88, which indicated a major improvement.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management

Keywords

Project Success, Prediction, Bayesian Classifier

1. INTRODUCTION

In empirical software engineering, many researchers have tried to predict the quality and cost of a product using project data sets. In previous research [1–3], project data sets were obtained from well-designed projects. Therefore, the project data sets were complete, i.e. all metric data were filled out. Even when the data was not complete, there were only a few missing values [4, 5].

However, we worked with a public project data set obtained from an ordinary software development company, which contained many incomplete data. For example, the International Software Benchmarking Standards Group (ISBSG) collects software development project data with 65.4 % of the data missing [6]. In Japan, Software Engineering Center, Information-Technology Promotion Agency Japan (IPA/SEC) collects data with 83.8% of the data missing [7].

In this paper, we report an attempt to improve accuracy of prediction for product quality by using a subset of project data collected by IPA/SEC [7]. The total number of project data in the dataset is 237.

First, we constructed a prediction model with the IPA/SEC data set by applying the naive Bayesian classifier. The result showed that accuracy of predicting successful projects was 0.86, which was very high. However, accuracy of predicting unsuccessful projects was 0.53, which was very low.

To find the reason for low accuracy, we analyzed the characteristics of the IPA/SEC data set and revealed the following two factors that appeared to affect accuracy:

(1) Incompleteness: 44.6% of the values in the data were missing.

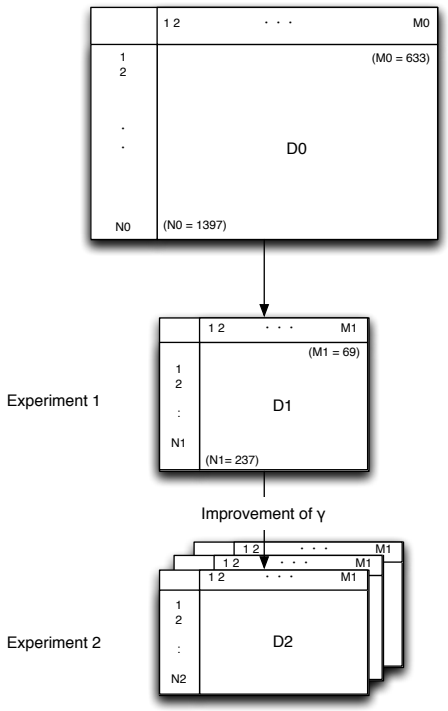


Figure 1: Modification of the dataset

(2) Imbalance: The number of successful projects was three times that of unsuccessful projects.

We first attempted to reduce the degree of incompleteness by mechanically minimizing the data that contained many missing values. The results of the preliminary experiment on the IPA/SEC data set showed that the rate of missing values was reduced by our method. Moreover, the ratio between the number of successful and unsuccessful projects also gradually reduced, which simultaneously reduced the imbalance.

On the basis of the preliminary experiments, from a given dataset, we deleted the data in which the number of missing values was larger than γ, where γ was a certain integer value, and calculated prediction accuracy by constructing a prediction model using the resultant dataset. The result of the experimental evaluation showed that when γ was 6 and the total number of data was 56, accuracy of predicting successful projects was 0.92. In addition, accuracy of predicting unsuccessful projects was 0.88, which indicated a major improvement.

Figure 1 shows how the data set is successively modified in this study. D0 is the original data collected by IPA/SEC. The characteristics of D0 are explained in Section 3. D1 is the dataset for the prediction (Experiment 1), which is explained in Section 4. On the basis of Experiment 1, we improved the data set and obtained the resultant dataset D2. We explain the details of the improvement in Section 5. Finally, we performed the prediction with D2, which is explained in Section 6.

2. RELATED WORKS

In this section, we refer to related works that have considered the two characteristics of incompleteness and imbalance of the data sets.

Tsunoda et al. studied the influence of imbalanced data on the prediction accuracy [8]. They evaluated some prediction methods by decreasing the number of unsuccessful projects and then showed that prediction accuracy reduces when imbalanced data is used.

Similarly, many studied have been conducted for "incompleteness" [4, 5, 9–12].

Three methods have been proposed to deal with missing values: (a) use the incomplete data without any change, (b) delete the data with missing values, and (c) substitute the missing value with any value.

Research efforts [4] and [5] used incomplete project data similar to method (a). Abe et al. applied the Bayesian classifier technique to software development project data to predict whether a project will be completed successfully [4]. Amasaki et al. applied association rule mining to software project data to identify risk factors [5].

Research efforts [10] and [11] used method (b). Twala et al. evaluated some imputation methods and concluded that a multiple imputation method attains the highest level of prediction accuracy [10]. Cartwright et al. evaluated the k-Nearest Neighbor imputation and the median imputation methods and found that the k-Nearest Neighbor method provided the best results [11].

Research efforts [9] and [12] compared methods (b) and (c). Strike et al. evaluated some imputation and deletion methods [9]. They concluded that a listwise deletion method is the most reasonable approach. Kromrey et al. found that when the project data contains many missing values, prediction accuracy reduces significantly with method (c) [12].

Because there are many missing values in the IPA/SEC dataset, we did not use imputation methods and used incomplete data after deleting data with many missing values.

3. CHARACTERISTICS OF THE DATASET

3.1 Project Data D0

Using IPA/SEC data, released as "IPA/SEC White Paper 2008 on Software Development Projects in Japan" [7], we attempt to predict whether a project will be completed successfully at the end of the design phase.

In Figure 1, the data is described as D0. $M0$ is the number of metrics and $N0$ is the number of projects of D0. The data consists of 1,397 custom enterprise software projects from 20 Japanese companies[1]. The number of metrics, which describe the projects, is 633. Thus, $N0$ is 1397, and $M0$ is 633.

Among the 633 metrics, in this paper, we use the metric "evaluation of results (quality)," which indicates whether the project will be completed successful or not.

Note that the data are characterized by a large number of metrics that are commonly used in these companies. However, the data also include many missing values, which is discussed in Section 3.3.

[1]The data were collected from 2,056 projects. However, we deleted the data that IPA/SEC or the company evaluated as unreliable. A detailed explanation is found in Appendix A

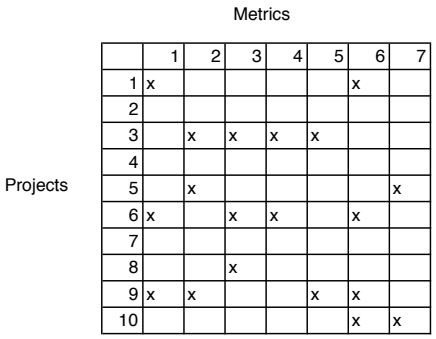


Figure 2: Explanation of missing data

3.2 Successful/Unsuccessful Projects

In this study, we distinguish successful projects from unsuccessful projects by using the metric "evaluation of results (quality)." The value of this metrics indicates the level of the quality delivered. This metric can be assigned a value from "a" to "e."

Here, "a" indicates that the number of defects after the system cutover is less than the planned value by 20% or more, and "b" indicates that the number of defects after the system cutover is less than the planned value.

A value of "c" indicates that the number of defects after the system cutover exceeds the planned value by 50% or less. A value of "d" indicates that the number of defects after the system cutover exceeds the planned value by 100% or less, and a value of "e" specifies that the number of defects after the system cutover exceeds the planned value by more than 100%.

In this study, if a project has the metric values "a" and "b," it is considered to be successfully completed, and if a project has the values "c," "d," and "e," it is considered to be an unsuccessful projects. In the data set D0, 186 projects were successful and 51 projects were unsuccessful. The values were missing in 1,160 projects.

3.3 Missing Values

Figure 2 shows the data consisting of seven metrics and 10 projects. In data set D0, the number of metrics is 633 and the number of projects is 1,397. The mark "x" in Figure 2 denotes the missing values. The fourth metric has two missing values and the third project has four missing values.

The rate of missing is calculated as $100 \times X/(M \times N)\%$ where the number of metrics is M, the number of projects is N, and the number of missing is X. In Figure 2, X is 19, M is 7, and N is 10. Thus, the rate of missing of Figure 2 is $100 \times 19/(7 \times 10) \fallingdotseq 27.1\%$. In case of dataset D0, the rate of missing is 83.8%.

4. EXPERIMENT 1

4.1 Outline of Experiment 1

In this paper, we predicted whether a project will be completed successfully. For this purpose, we selected only those projects that had data for the metric "Evaluation of result (quality)." As mentioned in Section 3.2, the number of project data that have values was 237 and of the 237 projects, 186 projects were successful and 51 projects were unsuccessful.

Table 1: An Example of Confusion Matrix

		Correct Result	
		Successful	Unsuccessful
Prediction Result	Successful	s	t
	Unsuccessful	u	v

Furthermore, the prediction is performed at the end of the design phase in the software lifecycle. Thus, the metrics that were unavailable at that time were deleted from D0. As a result, the number of metrics was reduced from 633 to 69. As shown in Figure 1, the obtained resultant data set D1 has values N1 = 237 and M1 = 69.

In this study, projects are predicted by applying the naive Bayesian classifier method, commonly known as the Bayesian classifier, which is one of the most common approaches for classifying categorical data into several classes based on a probabilistic model. This is an optimal method for supervised learning if the values of the attributes of an example are independent, given the class of the example. Although this assumption is almost always violated in practice, research effort [13] has shown that naive Bayesian supervised learning is also effective.

A project is predicted to be successful when the success probability calculated by the 10 fold cross validation is greater the criteria (in this paper, we set the criteria as 0.5). When success probability is less than 0.5, the project is predicted to be unsuccessful.

In this paper, we used the F-measure to evaluate the prediction accuracy of successful and unsuccessful projects. Table 1 is an example of a confusion matrix. In this example, the F-measure for predicting successful and unsuccessful projects are calculated as $2s/(2s+t+u)$, and $2v/(2v+t+u)$, respectively.

To ensure statistical validation, we repeated the 10-fold cross validation 100 times and used the average.

The averages of the F-measure for predicting successful and unsuccessful projects were 0.87 and 0.52, respectively. The F-measure for predicting unsuccessful projects was lower than that for predicting successful projects.

4.2 Analysis of Experiment 1

To find the reason for low accuracy in predicting unsuccessful projects, we analyzed the characteristics of dataset D1. The result of the analysis showed that the following two factors appeared to affect the prediction accuracy:

- Incompleteness: The rate of missing in data set D1was approximately 44.6%, which was high.
- Imbalance: Successful projects constituted 78.5% of project data, whereas unsuccessful projects comprised 21.5% of the project data.

Regarding incompleteness, Kromrey et al. found that in case of many missing values in the project data, prediction accuracy reduces significantly [12].

Regarding imbalance, Tsunoda et al. found that the imbalanced data affects prediction accuracy [8]. Therefore, we conjecture that reducing the imbalance may improve accuracy.

Hence, we tried to improve accuracy of predicting unsuccessful projects by reducing incompleteness and imbalance.

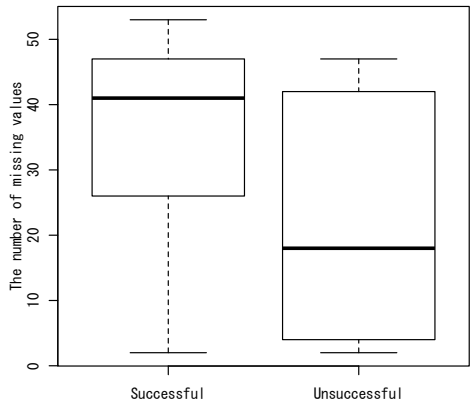


Figure 3: Distribution of the number of missing values

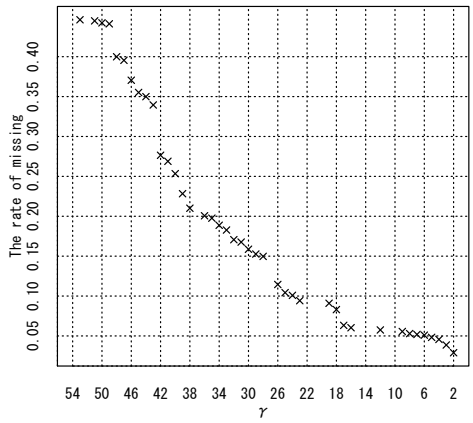


Figure 4: Improvement of incompleteness

5. IMPROVEMENT OF DATASET

5.1 Distribution of Missing Values

To reduce incompleteness, we first analyzed distribution of the number of missing values for dataset D1. As mentioned earlier, the project data included 186 successful and 51 unsuccessful projects. In addition, the rate of missing of D1 was 44.6%.

The result of the analysis of distribution of missing values is shown in Figure 3, which is a box plot of the number of missing values. From the figure, it can be observed that in successful projects, the maximum number of missing value is 53, the minimum is two, the median is 41, and the average is 33.5. In unsuccessful projects, the maximum number of missing values is 47, the minimum is two, the median is 18, and the average is 20.9.

The result shows that the number of missing values in successful projects is larger than that in unsuccessful projects.

For example, for 43 missing values, the number of successful and unsuccessful projects is 26 and four, respectively. Therefore, if projects are deleted in descending order of the number of missing values, incompleteness and imbalance may be reduced simultaneously.

5.2 Removing Missing Values

To remove projects in descending order of the number of missing values, we introduced γ, an integer that indicates the upper limit of the number of missing values. By the definition, γ must be the same as the number of missing values in the dataset D1.

The algorithm for eliminating the projects that have a large number of missing values is as follows: For γ_i: where $i = 1, 2, \cdots, m$ and $\gamma_1 > \gamma_2 > \cdots > \gamma_m$,

> Remove the data in which the number of missing values is larger than γ_i from the given data.

Here, we explain how the method is applied using the example in Figure 2. In this example, γ assumes the values 4, 3, and 1. From the 10 projects, the projects in which the number of missing values is four are removed. Thus, the third, fifth, and ninth projects are removed.

When we apply the method to dataset D1, γ assumes the values 53, 51, $\cdots$, 3, and 2. First, the projects in which the number of missing values is 53 are deleted. Thus, one project is removed. Next, the projects in which the number of missing value is 51 are removed. Thus, two projects are deleted. The remaining projects are removed in a similar manner.

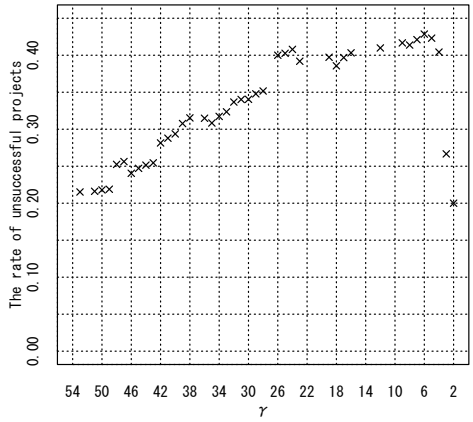


Figure 5: Improvement of imbalance

5.3 Improvement of Incompleteness

In this section, we evaluate incompleteness of the resultant dataset. According to the research [12], when the rate of missing is greater than 30%, prediction accuracy reduces significantly. Thus, in this experiment, we set a goal of the rate of missing as 30%.

Figure 4 shows the relationship between γ and the rate of missing. The horizontal axis indicates γ and the vertical axis indicates the rate of missing.

From Figure 4, it is observed that as γ reduces, the rate of missing reduces. Furthermore, Figure 4 shows that when γ is 42, the rate of missing is 27.6%, and when γ is 43, the rate of missing is 33.9%. Thus, we can infer that, from the viewpoint of incompleteness, it is desirable that γ is less than 42.

5.4 Improvement of Imbalance

As mentioned before in Section 5.1, it is expected that the imbalance can be reduced by removing the projects with missing values. In this section, we evaluate imbalance of the

resultant dataset.

Figure 5 shows the relationship between γ and the rate of unsuccessful projects. The horizontal axis indicates γ and the vertical axis indicates the rate of unsuccessful projects.

When the rate of unsuccessful projects is 0.5 (in Figure 5), the data is balanced. From Figure 5, it is observed that when γ is 6, the rate of unsuccessful projects is the closest to 0.5, i.e. 0.43. In this case, the total number of projects is 56, with 32 successful and 24 unsuccessful projects.

6. EXPERIMENT 2

6.1 Overview of Experiment 2

On the basis of the proposed method in Section 5.2, to predict product quality by using an improved dataset, we use the procedure that consists of improvement of a certain γ value and evaluation of accuracy of the γ.

For the first improvement, we applied the method used in Section 5.2, and for the second evaluation, we used the 10-fold cross validation to calculate the F-Measure. To ensure statistical validation, we repeated the 10-fold cross validation 100 times, and used the average value.

The procedure is defined as follows:

Phase 1 (Improvement for γ)

> Remove the data in which the number of missing value is larger than γ_i from the given data.

Phase 2 (Evaluation of accuracy)

> Calculate prediction accuracy by constructing the prediction model by applying the naive Bayesian classifier to the resultant dataset.

6.2 Analysis of Experiment 2

Figure 6 shows the result of the experiment. The horizontal axis indicates γ and the vertical axis indicates F-Measure. The "+" mark indicates the F-Measure for predicting successful projects, and the "x" mark indicates the F-Measure for predicting unsuccessful projects.

From Figure 6, it is observed that when γ is 6, the F-Measure for predicting unsuccessful projects is 0.88, which is the highest in this experiment. Furthermore, the F-Measure for predicting successful projects is 0.92, which is higher than that calculated from the given data. When γ is 6, the rate of unsuccessful projects is 0.43, the number of projects is 56, and the rate of missing is 5.1%.

Regarding the F-Measure for predicting successful projects, the highest value is 0.95, which is obtained when γ is 3. In contrast, the F-Measure for predicting unsuccessful projects is 0.83. When γ is 3, the rate of unsuccessful projects is 0.27, the number of projects is 30, and the rate of missing is 3.9%.

As mentioned in Section 4.2, our goal was to improve accuracy of predicting unsuccessful projects. Thus, we consider the case when γ is 6 as the result of improvement.

7. CONCLUSION

In this paper, we attempted to improve the prediction accuracy (F-Measure) using the IPA/SEC dataset [7]. Initially, accuracy of predicting successful projects was 0.86 and accuracy of predicting unsuccessful projects was 0.53, when the number of projects was 237.

We then analyzed the characteristics of the IPA/SEC dataset. The results of the analysis showed that incompleteness and imbalance may affect accuracy. In addition, we found that when incompleteness is reduced, imbalance is also reduced.

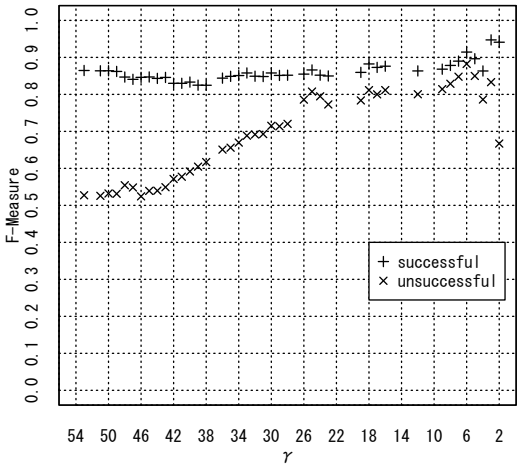


Figure 6: Relationship between γ and F-Measure

Therefore, we deleted the data in which the number of missing values was larger than γ, where γ was a certain integer value, and then calculated prediction accuracy by constructing the prediction model using the resultant dataset. The result of the experimental evaluation showed that when γ was 6 and the total number of data was 56, the resultant data was the most balanced and accuracy of predicting successful projects was 0.92. Furthermore, accuracy of predicting unsuccessful projects was 0.88, which indicated a major improvement.

8. ACKNOWLEDGMENTS

This work was partially supported by Grant-in-Aid for Scientific Research(C) (21500035) Japan and Grant-in-Aid for JSPS Fellows(21 3963) Japan.

9. REFERENCES

[1] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, vol. 23, pp. 736–743, 1997.

[2] Z. Chen, B. Boehm, T. Menzies, and Daniel Port. Finding the right data for software cost modeling. *IEEE Software*, Vol.23, pp. 38–46, 2005.

[3] M. Kläs, H. Nakao, F. Elberzhager, and J. Münch. Predicting defect content and quality assurance effectiveness by combining expert judgment and defect data - a case study. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pp. 17–26, Washington, DC, USA, 2008. IEEE Computer Society.

[4] S. Abe, O. Mizuno, T. Kikuno, N. Kikuchi, and M. Hirayama. Estimation of project success using Bayesian classifier. In *Proceedings of 28th International Conference on Software Engineering (ICSE2006)*, pp. 600–603, 5 2006. Shanghai, China.

[5] S. Amasaki, Y. Hamano, O. Mizuno, and T. Kikuno. Characterization of runaway software projects using association rule mining. In *Proceedings of 7th International Conference on Product Focused Software*

Process Improvement (PROFES2006), vol. LNCS 4034, pp. 402–407, 6 2006. Amsterdam, The Netherlands.

[6] International Software Benchmarking Standards Group. ISBSG estimating, benchmarking and research suite release 11. http://www.isbsg.org/, 2009.

[7] Information-technology Promotion Agency Software Engineering Center. *IPA/SEC White Paper 2008 on Software Development Projects in Japan*. Nikkei Business Publications, Tokyo, Japan, `http://www.ipa.go.jp/english/sec/reports/20100507a\_2/20100507a\_2\_WP2008E.pdf` 2008.

[8] M. Tsunoda, A. Monden, J. Shibata, and K. Matsumoto. Empirical evaluation of cost overrun prediction with imbalance data. In *Proceedings of International Conference on Computer and Information Science (ICIS 2010)*, August 2010. Yamagata, Japan.

[9] K. Strike, K. E. Emam, and N. Madhavji. Software cost estimation with incomplete data. *IEEE Transactions on Software Engineering*, vol. 27, pp. 890–908, 2001.

[10] B. Twala, M. Cartwright, and M. Shepperd. Comparison of various methods for handling incomplete data in software engineering databases. *Empirical Software Engineering, International Symposium on*, pp. 105–114, 2005.

[11] M. H. Cartwright, M. J. Shepperd, and Q. Song. Dealing with missing software project data. *Software Metrics, IEEE International Symposium on*, pp. 154, 2003.

[12] J. Kromrey and C. Hines. Nonrandomly missing data in multiple regression : An empirical comparison of common missing-data treatments. *Educational and Psychological Measurement*, Vol. 54, No. 3, pp. 573–593, 1994.

[13] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, vol. 29, pp. 103–130, November 1997.

APPENDIX

A. DATA RELIABILITY

To delete unreliable projects, we referred to the metrics "Data reliability (IPA/SEC)" and "Data reliability (company)" in the white paper [7]. These metrics evaluate the reliability of the project data and have four values: "a: The project data is confirmed as reasonable and completely consistent," "b: The project data looks reasonable, but it has several factors that affect consistency of the data," "c: Consistency of the project data cannot be evaluated because critical data items are missing," and "d: The project data has one or more factors indicating that the data is unreliable." In this study, we deleted the projects in which the value of these metrics was "c" or "d."

B. THE BAYESIAN CLASSIFIER METHOD

B.1 Bayes' theorem

Bayes' theorem relates the conditional probabilities of events A and B, provided that the probability of B does not equal zero. In Bayes' theorem, $P(A|B)$, the conditional probability of A given B is represented as follows:

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

In this expression, $P(A)$, $P(B)$, and $P(B|A)$ are defined as below:

- $P(A)$ is the prior probability of A.
- $P(B)$ is the prior probability of B.
- $P(B|A)$ is the conditional probability of B given A.

B.2 Bayesian Classifier

Let $M_1, M_2, \cdots, M_n$ be the variables denoting the observed attribute values to predict a discrete class C. Furthermore, let c represent a particular class. Given the values m_1, m_2, $\cdots$, m_n, we can use Bayes' theorem to calculate the probability $P(C = c|M_1 = m_1 \wedge \cdots M_n = m_n)$ and then predict the most probable class. This probability is expressed as follows:

$$\frac{\Pi_{i=1}^{n} P(M_i = m_i|C = c)}{P(M_1 = m_1 \wedge \cdots M_n = m_n)} \times P(C = c)$$

A System for Information Integration between Development Support Systems

Soichiro Tani[1] Akinori Ihara[1] Masao Ohira[1] Hidetake Uwano[1,2] Ken-ichi Matsumoto[1]
[1]Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, JAPAN
+81-743-72-5318
[2]Nara National College of Technology
22 Yata,Yamatokoriyama,Nara,JAPAN
+81-743-55-6000
{ soichiro-t, akinori-i, masao, matumoto } @ is.naist.jp, uwano @ info.nara-k.ac.jp

ABSTRACT
Many software projects use dezvelopment support systems such as bug tracking system (BTS) or version control system (VCS) to manage development information. Such the support systems preserve information according to a type of information (e.g., bug information in BTS and change information of source code in VCS). Since the systems do not provide developers with a feature to integrate several types of information required to complete development tasks, the developers need to collect the information by themselves that would result in inefficient development. In this paper, we demonstrate a system called SUSHI that helps developer integrate the information between multiple development support systems. Our system collects information which belongs to the same development context and provides developers with hyper links to related information in the support systems. Since our system also runs as a proxy server, developers can continue to use existing systems and stored information without any conversion.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *software configuration management, software quality assurance, programming teams.*

General Terms
Management

Keywords
information sharing, development context, bug tracking system, version control system

1. INTRODUCTION
With increasing outsourcing of software development or open source software projects, developers need to collaborate with geographically distributed co-workers and share information between them. In such the environment, they communicate each other to share information with asynchronous communication tools/systems such as mailing list (ML), bug tracking system (BTS) or version control system (VCS) that we call development support systems (DSS) in this paper.

In a large-scale software development organization, several DSSs are often integrated into a single system in order to optimize the efficiency of collaboration and information sharing among developers. The optimized integration of DSSs helps developers find and understand the past and current progress of development is dispersed in each DSS.

Meanwhile, a small-scale organization and open source software project does not have the integrated DSSs because they do not afford to build it by themselves or buy it. They often use a rental hosting service such as sourceforge.net that can be used at no fee. In case of using a rental hosting service, it is difficult to change/modify each DSS and composition of DSS to make the performance of information sharing better. Each developer needs to search several kinds of information in DSSs to understand the development context. For instance, when a developer receives a new patch via ML, s/he has to find a bug report which asks developers to make a patch to fix a certain bug, understand discussions about the bug on ML and BTS, and identify a file or module in VCS to apply the patch, and commit it into VCS. As just described, using DSSs as a combination of independent DSS sometimes consumes developers' time and effort [1, 2].

In order to resolve the issue on the use of DSSs in a small organization, in this paper we propose a system called SUSHI that supports information integration between multiple DSSs without changing in-use independent DSS in the organization. Our system serves as a proxy server which detects user's access to DSSs, collects the same information that the user refers to, and make an association with several kinds of the referred information in multiple DSSs. Using the association of development information, the next developer can easily find information relevant to his tasks.

2. SUSHI: INTEGRATING INFORMATION BETWEEN MULTIPLE DSSs

2.1 Overview
Figure 1 shows the information flow in using SUSHI. SUSHI works as a proxy server, collects information from BTS, and provides users with related information while the users look for information in each DSS through a web browser. Due to this configuration, SUSHI can deal with several kinds of information

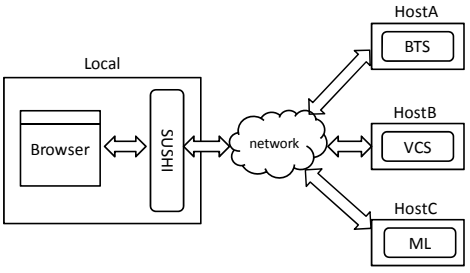


Figure 1. Information flow in SUSHI.

in multiple DSSs. At the same time, the users of SUSHI are not required to intentionally operate SUSHI to have information relevant to the development context.

2.2 Architecture of SUSHI

Figure 2 shows the architecture of SUSHI. SUSHI has four components: (1) **Access Detector**, (2) **Data Collector**, (3) **Estimator**, and (4) **Formatter**.

Access Detector automatically detects user's access to DSSs and notifies Data Collector of what information are browsed by the user. At this moment, Access Detector eliminates information irrelevant to DSSs such as web searching results.

Data Collector collects information in DDSs that the user referred to and sends them to the internal database. More importantly, it also extracts information which is used in *Estimator* to estimate which information are related each other. In the current implementation, *Data Collector* of SUSHI gathers bug ID, reporter's name, reported time and descriptions of a reported bug from BTS, revision number, committer's name, commit message and name of committed file from VCS, and poster's name, posted time, message ID and etc. from ML.

Estimator estimates the development context which consists of several events around the same time. *Estimator* makes an association with several kinds of information using such the events recorded in DSSs. The next subsection describes *Estimator* in detail.

Formatter provides a web page (html) which has the original information that the user is about to browse and relevant information from other DSSs. The user can browse several kinds of relevant information at a time to understand the development context.

2.3 Procedure of Estimation

In *Estimator*, making an association with relevant information, that is, estimating the development context is conducted as follows.

1. Developers specify characteristic words which are often used in their project.
2. *Estimator* counts the number of the specified key/characteristic words used in both the information that a user is browsing and the information that SUSHI's database has already stored.
3. *Estimator* calculates the used rate of a specified word by dividing the number of the specified word used in each information by the total number of all the specified words used in each information.
4. *Estimator* calculates the estimated score by squaring the difference of the used rate between the information that a user is browsing and the information in SUSHI's database.
5. *Estimator* considers combinations of the information with lower estimated score as relative information each other.

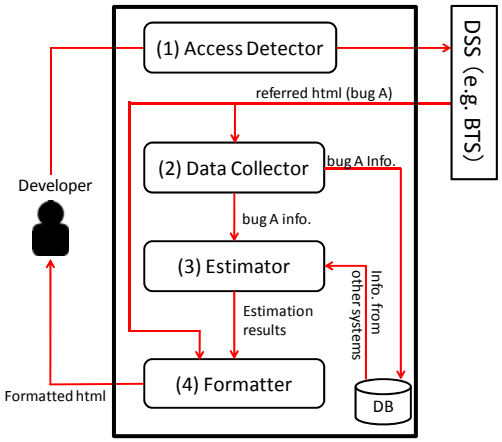


Figure 2. System architecture.

3. PRELIMINARY EVALUATION

So far, we have applied SUSHI to two small-scale open source projects to confirm if the system works properly as we intended. Due to the space limit, we introduce a summary of the preliminary evaluation.

We asked open source developers to make links between information recorded in several DSSs to define which information is relevant each other. We also applied SUSHI to DSSs used in their projects and extracted links which were created by SUSHI. Comparing links manually defined by the developers with the links created by SUSHI, we found that the links (BTS→BTS, VCS→BTS, and VCS→VCS) created by SUSHI covered over 50% of the links defined by the developers. However, we also found that the links (BTS→CVS) less matched the developers' links (17%). This result showed that finding and recovering the missing links [3] are still difficult in our system.

4. FUTURE WORK

In the current, SUSHI only shows a simple output with estimated relevant information to uses. We need to consider more user-friendly user interface and visualization as uses can easily understand and remember the development context. We also improve the estimation algorism for relevant information. The current algorism is too simple to estimate relevant information correctly. Although the preliminary evaluation showed good results basically, we believe we can enhance the algorism (e.g., using TF-IDF) to make developers' information retrieval much more efficient.

5. ACKNOWLEDGMENTS

This work was conducted as part of StagE Project (the Development of Next Generation IT Infrastructure), Grant-in-Aid for Scientific Research (B), 23300009, 2011, and Grant-in-aid for Young Scientists (B), 22700033, 2011 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

6. REFERENCES

[1] Ohira, M,. Yokomori R., Sakai M., Matsumoto K., Inoue K., and Torii K. 2004. Empirical Project Monitor: A Tool for Mining Multiple Project Data. In *Proceedings of International Workshop on Mining Software Repositories (MSR'04)*. pp.42-46.

[2] Johnson, P.M. 2007. Requirement and Design Trade-offs in Hackystat: An In-Process Software Engineering Measurement and Analysis System. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*. pp.81-90.

[3] Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE'10)*. pp.97-106.

Open your Mind

Understanding OSS Openness through Relationship between Patch Acceptance and Evolution Pattern

Passakorn Phannachitta[†] Pijak Jirapiwong[‡] Akinori Ihara[†]
Masao Ohira[†] Ken-ichi Matsumoto[†]

[†]Nara Institute of Science and Technology
8916-5, Takayama, Ikoma
Nara, Japan
{phannachitta-p, akinori-i, masao, matumoto}
@is.naist.jp

[‡] Kasetsart University
50 Ngam Wong Wan Rd, Chatuchak
Bangkok, Thailand
b5005135@ku.ac.th

ABSTRACT

Openness is referred how much does the OSS core committer team share and compile with their non-committers. Because the openness can be varied from time to time, a study for explaining the vary of openness would be a good approach to support the OSS. It will not only encourage the non-committers to exert more contribution when the openness is high, but it will also make the them still have an optimistic outlook on the project when the openness is low. Unfortunately, there is only a few studies aiming to understand this principle. This work, we seek out for the key factors that identify the transitory changed of the openness. Unlike most previous studies, we focus on the clear relevant evidences that are more concrete. Our temporal-based analysis on patch acceptance in two major OSS projects: Apache HTTP Server and Eclipse Platform conclude that special event occurrences have a decisive influence over the openness either in transitory or lasting. Moreover, our investigation on the relationship between the temporal changes of openness and a plausible OSS evolution pattern broadens the mutual accord in both openness and the evolution of OSS.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.9 [**Software Engineering**]: Management

General Terms

Management, Measurement, Experimentation

Keywords

Open Source Software, OSS Evolution Pattern, Temporal Factor, Openness, Patch Submission, Patch Acceptance

1. INTRODUCTION

An active collaboration between every role in Open Source Software (OSS) project is one of the most important key for the sustainable development [6]. Many OSS supportive activities will be ardently achieved whenever each role is well-balanced in the number of participant and everyone exerts the best effort. For an example on patch submitting activity, which committers and non-committers are the two main different roles. Non-committers whose authority makes them unable to apply patches for the project directly will be the patch submitter, and the committers will be the judge who accept the patch. An OSS project needs a high number of active non-committers to increase and widen the perspective of requirements and patches the covers the larger area of defects. The OSS committer team should also have enough members that is capable for verifying all the incoming requirements and patches thoroughly. Ideally the committer should always be open and shared to the non-committer so as to encourage them to increase their contribution.

Nowadays, a study that aims for comprehension between committers and non-committers is quite overlooked. Most studies give more attention to the individual supportive either for committers or the non-committers. Studying the OSS committer's openness can be one that support both committers and non-committers at the same time. Although it has been concluded that temporal changes of the openness are existence [14], more deepening study for the varieties of the temporal factors is noteworthy. Analyzing without concerning the temporal factors, the broader insightful study of the OSS committer's openness such as the open season prediction or answering why the openness level is always changed will be inconceivable.

This research aims to explain the transitory changed of the committer's openness. First, we have analyzed the patch acceptance in temporal on the two well-known OSS projects, Apache HTTP Server and Eclipse Platform. Temporal factor is important since it's unfair to conclude how open of an OSS committer team from the summarized statistic. From our preliminary results, we have found that the special event occurrences were the key factor influenced to the transitory openness level. Further, we have noticed that there are some remarkable treads exposed to the vary of the openness. Elucidating those trends we have found the relationship between the temporal changed of openness with a plausible OSS evolution pattern proposed by Nakakoji et.al [7]. The relationship we found is not only a sounded explanation for understanding the changes of the openness, but also a proof of the validity of our chosen evolution pattern.

The remainder of the paper is organized as follows: Section II explains the backgrounds and briefs the existing related works. Section III introduces our research questions. Section IV explains our methodology. Section V describes the case study setup and discusses on our finding. Section VI contains an observation from the study. And finally, section VII provides the conclusion and outlines some future works.

2. BACKGROUND KNOWLEDGE

2.1 Related Works

To date, it has been quite lack of studies aimed to improve the comprehension between committers and non-committer. Most researches provide useful guidances for each individual role. Bettenburg et al. suggested non-committers what will make a good bug report [1]. Weißgerber et al. also suggested them the proper size of the submitting patch [13]. On the committer side, Wang et al. and Runeson et al. help the committer ascertain if the reported bug is duplicated [9, 12], and Rigby et al analyze the personality of the top committer [8].

Nonetheless, a few studies proposed to make the comprehension between both roles, which will support both the same time. Shibuya et al. have studied the process of participating in OSS [10]. Their finding encourages the non-committer to gain more reputations; however, their work is still lacking in enthusiasm for making committer be more open to the non-committer.

2.2 Patch Submission and Patch Acceptance

In OSS, patches contain different information between two versions of the same file. Exchanging patches instead is very convenience since the receiver can definitely know the changed information without any further attempt. In practical, when a non-committer needs some changes with a target file (i.e. A file that locates in the project repository) , that non-committer will check out the target file and edit it. After that, the edited target file will be converted into a patch and sent to the committers. Patches are usually sent through a common channel provided by the OSS community. The currently most popular channel is bug-tracking system such as Bugzilla. Along the way to the committers, the bug-tracking will let the submitted patch be discussed. It helps the committer to evaluate how good and valuable of the submitted patch. At last, if that patch reach a consensus and the committers concur, the committers will apply the submitted patch with the target file and commit it. Whether the whole patch or just its portion is committed, it means the committers accept the patch.

There is an interesting phenomenon that the committers can gradually accept a patch. Because a large patch would contain more components which probably have high dependencies. It seems impossible that all the components would be accepted and committed at once. Unless we include the gradual patch-acceptance case, we will be unable to report the reflected result from the patch acceptance analysis. Unfortunately, the recently existing researches ignored this phenomenon [2, 13] that may lead the authors to an inaccurate conclusion.

2.3 Openness and Open Season

Openness is a term referred how much do the committers share and comply with their non-committers. To date, it is still inconclusive that a high openness committer is always a good committer [8]. However it's a rational thought that as long as the non-committers know the committers are open, they will give their earnest effort to the project, because the non-committers would feel they are important know that they are not being isolated. Moreover, avoiding the decreasing of openness will induce the committers to give more attentions to the non-committers.

Openness can be analyzed in many aspects, such as counting the committers' reply messages or approximately accumulating their spending time with the bug report. However, an absolutely clear evidence such as the number of patch acceptance would rather conclude the openness. In this work we can know the actual committed number of a patch (from the partially committed case). We define an openness in our works by counting the time that committer's committed any part of patches in a defined period. It would give us more accurate insight than counting the number of accepted patches directly. There is an interesting question about analyzing the openness about concerning the temporal factor Definitely, an OSS project has been continually and unceasing developed. It's hard to believe that the committers team is composed with the same people throughout the time. The team can be reorganized so that the openness is possible to be changed. Since the openness is changeable, patches can also be more or less accepted in some periods. We will denote a period contains a relative high number of patch acceptance as in opening season.

2.4 Evolution Pattern of OSS

The temporal-based analysis of the OSS committers' openness will inherently reveal the characteristics of OSS project distinctively between each development period. A further analysis from these informative characteristic will broaden the accord about an evolution of OSS. Currently, there are several existing studies of OSS evolution patterns [3, 4, 11], we selected one plausible pattern proposed by Nakakoji et al [7]. They categorized OSS project into three types, and also outline the characteristics of each type in many dimensions Three types of OSS project are Exploration Oriented, Utility Oriented, and Service oriented. We summarize the dimensions related to our work in table 1.

2.4.1 Exploration Oriented

An OSS project usually becomes an Exploration Oriented when the project itself has been received some new ideas. Normally, new ideas are emerged by the senior developers or the project leader whom know the project very well so that the new ideas are usually discussed among the core members. Being an Exploration Oriented project, core developers or committers seem to more closed comparing between other periods. It's more likely to be happened when the project was just founded or after it has been mature for a while.

2.4.2 Utility Oriented

An OSS project usually becomes this type when a project needs of some new features or enhancements. It's different from Exploration Oriented, because requiring for a new feature or enhancement can be a requirement from anyone. Reforming into a Utility Oriented is possible anytime, either the project is under developing and has a plenty of bugs or the project has been being mature for a long time. Being a Utility Oriented project, core developers or committers seem to be most opened among other periods, since they need more suggestions to be discussed.

2.4.3 Service Oriented

An OSS project is usually reformed into Service Oriented after a project released a stable version. The goal of a Service Oriented project is to provide service as stable as possible. Being this type, core developers or committers are slightly less open than a Utility Oriented project and the

Table 1: Three Types of OSS Project

	Exploration-Oriented	Utility-Oriented	Service-Oriented
Objective	Sharing Innovations and Knowledge	Satisfying an individual need	Providing stable service
Control style	Cathedral-like central control	Bazaar-like decentralized control	Council-like central control
Community structure	Project leader and core members	Core members and many peripheral developers	Core members and Many passive users
Major problems	Finding a novel innovation	Difficult to choose the right program	Less innovation

openness is likely to be continually decreased, because the project has been more stable that would normally have less number of defects.

From the three classified types of the OSS project proposed by Nakakoji et.al [7], they believed that the evolution pattern of any OSS will be alternating from one type to others as illustrated in figure 1. They also gave some examples of an idea for the transition between each types (i.e. When a Service-Oriented project has some new needs, it will reform into a Utility-Oriented project.) However the conclusion to support if the evolution pattern is respected to this pattern as shown in the figure 1 are still waiting for a proof.

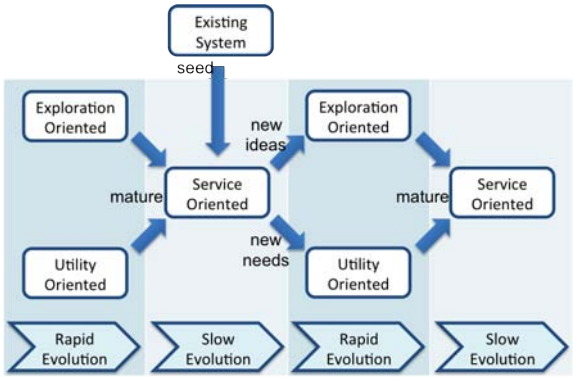


Figure 1: The Evolution Pattern of OSS Projects [7]

3. RESEARCH QUESTION

To develop an understanding of the transitory changed of OSS committers' openness through analyzing the patch acceptance and OSS evolution pattern, we need to answer two research questions.

3.1 What is the key factor that makes the OSS committers changed their openness?

Tackling this question, we would understand clearer about the patch acceptance. Achieving the key factor that induce an open season will allow us to briefly foretell the non-committers when should they exert more contribution. Furthermore, if the open season characteristic is likely to be in common, it could be a feature for predicting the open season, which is more promising. On the other hand, we also need a sound reason to explain why do the committers decrease their openness in some periods. Without a sounded reason, the non-committer may consider that the committer teams are having some trouble that they are unable to thoroughly verify the submitted requirements and patches.

3.2 How can an OSS evolution pattern explain the trend of OSS committers' openness?

Unlike the other routine works, the openness of the OSS committers does not always stay at the same level [14]. At least it still respected with some trends (i.e. keeps increasing or decreasing for a period of time). OSS evolution pattern may have an supportive explanation since it explains the change of OSS projects. If it is existed, we would reach a better insight of the OSS openness and the patch submission activity.

4. METHODOLOGY

For the case study, we develop a method to gather the patch acceptance traces divided into period of time. The method has three phases that are Patch extraction, Diff file creation, and Patch acceptance identification.

4.1 Patches extraction

We have two types of data source; Mailing list and Bugzilla. Mailing list is where committer and non-committer discuss about patch and bug, and Bugzilla is a well-known bug tracking system. Their structures are totally different, so we need two specific patch extraction methods.

For the mailing list, we choose to improvise and extend the Weißgerber et al.'s proposed [13]. It is the closest method that fulfilled our requirement in patch extraction from email. On the other hand, we have to develop a specific web site crawler to extract patches from Bugzilla from the scratch since such an explicated proposed method is nonexistent.

In both methods, we denote our patch format with a tuple $(I_p, P_p, t_p, L_p, [c_p])$ where I_p is the patch's index, P_p is the patch's absolute path that we can identify the corresponded target file in the repository, t_p is the patch's submitting time, which we has already converted into a common timezone (UTC) for avoiding an incorrect time-stamp identification. L_p is the total number of the changed lines of code, and $[c_p]$ is a list contained all the individual changed lines of code. Note that, we collapse all white spaces in $[c_p]$, because in practical the committer may apply patches manually. It then may produce some white spaces shifted that would lead us to an inaccuracy analysis. After we has extracted all the information in $(I_p, P_p, t_p, L_p, [c_p])$ from the raw-patch data , we stored them in a database.

4.2 Diff files creation

We denote a Diff file as a file that contains the changes information between each committed version at the repository side. These changed information make us able to track whether the submitted patch are accepted. We create diff files between each adjoining revision of all the target files in the repository in order to know when do the patches are committed. Moreover, creating a diff file between each adjoining revision make us able to include the gradual patch acceptance case into our study.

In this phase, we also have two types of repository data source: CVS and SVN. For both types of repository, at first we extract the revision number and the timestamp of each target file. A pair of revision number and timestamp then

guide us to obtain all the change information of target files. Next, we create diff files from each pairs of adjoining revision and then extract the required information similar to the patch extraction. We also denote the diff files as a tuple (I_r, P_r, t_r, $[c_r]$). I_r is committed source code's index, P_r is the its absolute path, t_r is its timestamp. We also parse the time zone into a common timezone as the patches. $[c_r]$ is the list of changed line in a revision which white spaces are also collapsed . At last, we compose (I_r, P_r, t_r, $[c_r]$) into records and store them in a database same as the extracted patches.

4.3 Partial Acceptance Identification and Commitment Counting

Since our analysis includes the gradually accepted case, we conclude a $patch_i$ as an accepted patch if there is an existed line of the submitted code that has been committed with the corresponded target file to the repository within a time limit Δt.

$$\begin{aligned} & (I_p = I_r \vee P_p \sim_{match} \ P_r) \\ \wedge \ & t_p \ + \Delta t \leq t_r \\ \wedge \ & (\exists l \, | l \in [c_p]) \subseteq [c_r] \end{aligned}$$

We count up the number of patch commitments of the accepted patches to study the openness. The counter of patch commitment increases each time either when a patch is fully committed or partially committed.

5. CASE STUDIES

We study on two well-known OSS projects: Apache HTTP Server and Eclipse Platform. Apache HTTP Server has been introduced as a web server since 1996 and is still popular. Eclipse Platform is a part of Eclipse project, which is a well-known interface development environment (IDE) project. We analyze the patch submission and acceptance on Apache HTTP Server between its mailing list system and its SVN repository. For another Eclipse platform project, we analyze between its bug-tracking system named Bugzilla and its CVS repository. Table 2 shows the quantitative information on both case-study data sources.

Table 2: The characteristic of the case-study datasets

(a) Repositories

	Apache HTTP Server	Eclipse Platform
Repository	SVN	CVS
Observing period	1998/06 - 2003/06	2002/06 - 2008/06
#File	6283	46,004
#Changed line	1,994,030	9,532,211

(b) Patches Data Source

	Apache HTTP Server	Eclipse Platform
BTS	Mailing list	Bugzilla
Observing period	1998/06 - 2002/06	2002/06 - 2007/06
#Patch	5,212	75,808
#Target file	1,926	71,153
#Changed line	140,391	9,919,338

We assign the time scope Δt as 365 days on both data sets because we concern the gradual patch accepted case. Larger patches are normally composed with more components so that they will probably take longer time to be fully accepted. Note that the Patches Data Sources contain one year less data than the repositories to make the experimental results reflected with our max Δt as 365 days. Figure 2 is illustrated our experimental results performed on both datasets. In both graphs the y axis denotes the total number of patch commitment, and the x axis denotes the timeline in month interval. We will explain about the label above the graph later during the subsidiary of research questions.

5.1 What is the key factor that makes the OSS committers changed their openness?

Observing through the both timelines 2(a) and 2(b), we are massively captivated by the frequent occurrences of pulses. It's rather unusual that needs a satisfactory explanation. There are several good metrics analogous to the temporal factors [5]. Special event occurrence and trend are very interesting factors. However, the pulses' appearance seems to have high impact and be in transitory that make us prefers a hypothesis about the occurrence of special events over the trend.

We have noticed two different characteristics of the pulses in both figures 2(a) and 2(b). First is the number of commitment has continually increased or decreased after a pulse is occurred. Another one is a pulse is surrounded by lower number of commitment that are approximate equivalence. Therefore, there should be more than one type of event occurred.

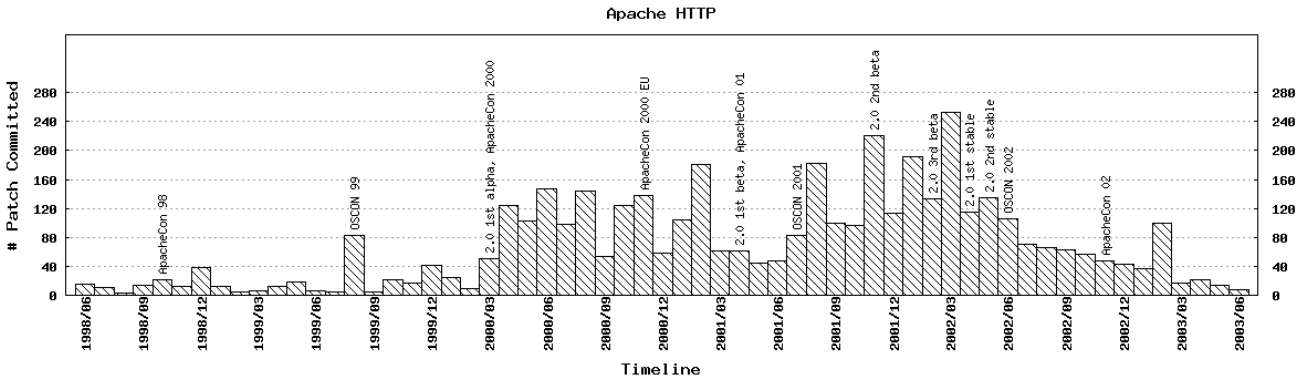
5.1.1 The number of patch commitment has continually increased or decreased after a pulse is occurred

We start a deduction from the most ordinary case. We thought the decreasing (i.e. 2003/02 in Apache HTTP Server and 2006/02 in Eclipse platform) is tend to more unusual than the increasing. Because the increasing number of patch acceptance may as usual as the project growth. One plausible explanation is during a decreasing patch commitment period, the project was stable enough and had the less number of severe bugs.

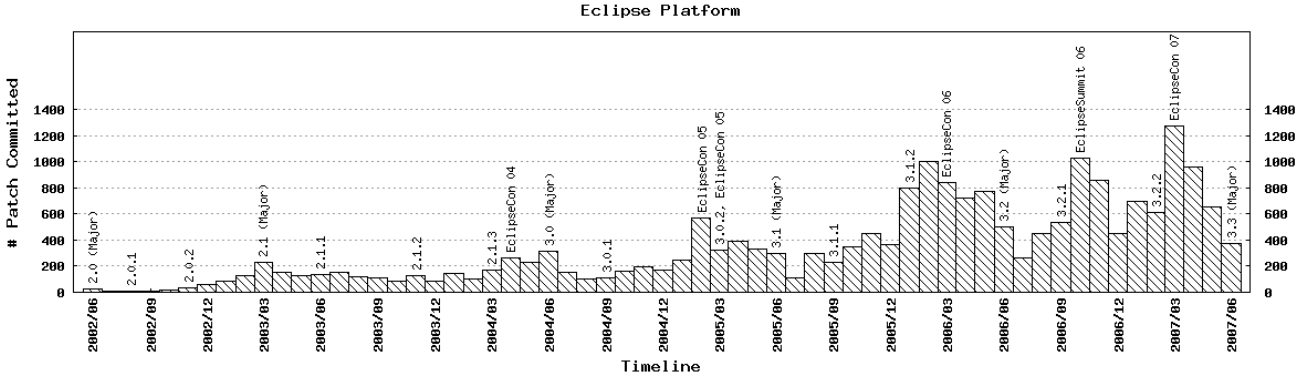
We investigated the released dates of the project in prior versions, and the released dates we have found are likely matched with the pulses related to this case. We label the corresponded released version over the graph in figure 2. It supports our hypothesis about the stable released versions. Moreover, it also tells us about the increasing period of patch commitment after a pulse is occurred (i.e. 2000/03 in Apache HTTP Server and 2005/02 in Eclipse platform). The pulses belonged to this case are matched with the release of some minor versions (i.e. alpha and beta version). Because the release of minor version still probably has many topics needed to be discussed. Note that, here the released of minor versions may not produce a pulse, but the number of patch commitment is always increased for a short period after that. (i.e. 2002/02 in Apache HTTP Server and 2002/11 in Eclipse platform)

5.1.2 A pulse is surrounded by a lower commitment periods that are approximate equivalent level

We have outlines a hypothesis about the events supporting this case. The supportive event for this type of pulse should be held in a short period and probably has a transitory effect, since it does not induce any change of the patch



(a) Analyzing Patch Acceptance from Apache HTTP Server project between June 1998 and June 2003



(b) Analyzing Patch Acceptance from Eclipse Platform project between June 2002 and June 2007

Figure 2: The Patch Acceptance Analysis Results from Apache HTTP Server and Eclipse Platform

acceptance.

We start our investigation on the first significant pulse of Apache HTTP Server in August 1999. Focusing on that time point, we turn up to a record of the first O'Reilly Apache Conference held in that month. It is perfectly matched with our hypothesis that caused by the OSS community must have placed some advertisements for the forthcoming conference. (i.e. calling for paper) It then captivated more developer to participate with the OSS community. We also track for more corresponding conference shown on the figure 2. Eclipse platform is more explicit to conclude this hypothesis with the conference occurrence.

5.2 How can an OSS evolution pattern explain the trend of OSS committers' openness?

Focus on the control style column of each type of OSS project in table 1, the three different styles tell us obviously that we can distinguish each type of the OSS projects by the amount of openness. Since patch acceptance is directly related with the openness, a relationship between OSS evolution pattern and the patch acceptance would exactly be existed. In figure 2, if we omit the special-event pulses, there will be only the trends of the patch acceptance left: increasing, decreasing, and stay-the-same.

The most simple case is the increase of patch acceptance. Since the Bazaar-like decentralized control [6] is the only one type of OSS project that could generate it, we could assume the project must have been a Utility-Oriented during the increasing of patch acceptance. We then also believe that the project was transiently Utility-Oriented during a special event pulse, because of the drastically increment of the accepted patch. Next, the decreasing trend of the patch commitment is usually happened after a major released. It is caused by the project is more stable, then fewer defects needed to be fixed. Consequently, the objective of the Service-Oriented as shown in table 1 makes it perfectly matched with this case.

However, the stay-the-same case is quite complicated. It can be occurred in several situations. The most simple case for the stay-the-same is when the project has just been founded or been well-known. This case is simply concluded as an Exploration-Oriented type. Then, what would be happened if the stay-the-same is occurred after the increasing (Utility-Oriented) period? After many needs and requirements were satisfied and many features also have been appended; the project should be more mature. We will conclude the stay-the-same after an increasing of patch acceptance period as a transition period between Utility-Oriented and Service-Oriented. At last, what if the stay-the-same was occurred after the decreasing (Service-Oriented) period. After a project has been mature for while, the need of expansion should become an issue. It may be a transition to become an Exploration-Oriented or a Utility-Oriented. If new ideas and innovations are more important, it will be a transition to be an Exploration-Oriented. On the other

hand, if the expansion is inspired by new requirement, it will probably be a transition to be a Utility-Oriented.

6. OBSERVATION

Comparing between two projects, Eclipse Platform are more predictable. Figure 2(b) shows that Eclipse platform has been continual growing respected to the almost same pattern from year to year (start in March). Most of the local relative peaks are always in the month that held a conference, and the number of patch acceptance always decreases after a major release. We believed it is caused by Eclipse Platform has a clear routine on its conference (around March) and the released date of major version (around June). Since the target checkpoint is clearly defined, all the participant will know how much should they exert their effort averse to the time to make the project reached the goal. Moreover, the clearly defined checkpoint would make the project self-comparable and let the participants know how much does the project develop from year to year. It is very interesting that when Eclipse platform has introduced a conference (Eclipse Summit) in 2006, the growing pattern is still the same but has been shorten (From March to October and October to March). This is a good proof of our caprice that the occurrence of conference has an influence to the openness, especially in a project that defines the important events very well such as Eclipse platform.

7. CONCLUSION AND FUTURE WORK

In this research, we found out what makes the OSS committers changed their openness and how could an OSS evolution pattern explain it. We analyzed two well-known OSS projects: Apache HTTP Server and Eclipse Platform. The results revealed two interesting identities of the OSS committers' openness. First, we found that at least two types of special event occurrences would affect the openness. When an official conference or workshop are forthcoming, the openness will remarkably increase. The conference's announcement and advertisement would arouse the newcomers as well as the currently inactive developers and users. The project then become more active so that wider varieties of patches were produced and submitted. The second type of special event is the released versions.After a minor version was released, there usually still has some open issues waiting to be resolved so that the committers would be still or opened. Alternately, a major version should be released after most of opening issues has already been resolved; hence, the number of patch acceptance would typically be decreased.

Our second finding is the relationship between the transitory changed level of openness and the OSS evolution pattern proposed by Nakakoji et.al [7]. The relationship we found can elucidate that the alternation of committer's openness level is an effect of the OSS project's evolution. Moreover, the result from our in temporal-base patch acceptance analysis can be a proof for the inconclusive hypothesis the evolution pattern, which they believed an OSS project would be evolved alternately from a type to others continually.

Beyond this research, we will explore more aspect and study more features to develop a knowledge of OSS committers' openness. We are ardently believed that openness and open season are entirely predictable.

Acknowledgment

The first and second authors are grateful to the internship program cooperated and supported between Kasetsart University, Thailand, and Nara Institute of Science and Technology, Japan. It bestows a grant as well as an opportunity for undergraduate student to achieve a wealth experience in abroad graduated school research.

This research is being conducted as a part of the Next Generation IT Program and Grant-in-aid for Young Scientists (B), 22700033, 2010 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

8. REFERENCES

[1] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08)*, 2008.

[2] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in oss projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*, May 2007.

[3] A. Capiluppi, J. M. González-Barahona, I. Herraiz, and G. Robles, "Adapting the "staged model for software evolution" to free/libre/open source software," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting (IWPSE '07)*, 2007.

[4] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 2000.

[5] B. Manaskasemsak, A. Rungsawang, and H. Yamana, "Time-weighted web authoritative ranking," *Inf. Retr.*, vol. 14, April 2011.

[6] K. Nakakoji, K. Yamada, and E. Giaccardi, "Understanding the nature of collaboration in open-source software development," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, 2005.

[7] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*, 2002.

[8] P. C. Rigby and A. E. Hassan, "What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list," in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*, 2007.

[9] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, 2007.

[10] B. Shibuya and T. Tamai, "Understanding the process of participating in open source communities," in *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '09)*, 2009.

[11] N. Smith and J. F. Ramil, "Agent-based simulation of open source evolution," in *Software Process Improvement and Practice*, 2006.

[12] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, 2008.

[13] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proceedings of the 2008 international working conference on Mining software repositories (MSR '08)*, 2008.

[14] M. Yamamoto, M. Ohira, Y. Kamei, S. Matsumoto, and K. Matsumoto, "Temporal changes of the openness of an oss community: A case study of the apache http server community," in *Proceedings of The Fifth International Conference on Collaboration Technologies (CollabTech '09)*, 2009.

A Tool for Collaborative Guitar Chords Creation based on The Concept of The Distributed Version Control

Chakkrit Tantithamthavorn[1], Papon Yongpisanpop[2], Masao Ohira[2],
Arnon Rungsawang[1] and Kenichi Matsumoto[2]

[1]Department of Computer Engineering,
Faculty of Engineering,
Kasetsart University, Bangkok, Thailand
{ b5105256, fenganr } @ ku.ac.th

[2]Graduated School of Information Science,
Nara Institute of Science and Technology,
Nara, Japan
{ papon-y, masao, matumoto } @ is.naist.jp

ABSTRACT
The distributed version control system has become very popular in software development process for years. Many software projects, which are developed by a large group of programmers have been using it to manage their source code. We study on another aspect of applying a distributed version control to keep track the version of other artifacts instead of using just for tracking source code. An application for sharing and creating guitar chords is used in our case study. We apply distributed version control concept in order to interoperate musical ideas from multiple users and preserve the versions of guitar chords with their music styles introduced to the community. The results show how distributed version control system can help guitarists to collaborate on creating guitar chords and also reduce the redundancy of derivative versions. This study has approved that distributed version control system can be applied to any projects that need to keep track the version of their artifacts.

Categories and Subject Descriptors
H.5.3 **[INFORMATION INTERFACES AND PRESENTATION]:** Group and Organization Interfaces - *Asynchronous interaction, Collaborative computing, Web-based interaction;* H.5.5 **[INFORMATION INTERFACES AND PRESENTATION]:** Sound and Music Computing - *Methodologies and techniques, Systems*.

General Terms
Design, Human Factors, Experimentation

Keywords
Chord Archives, Distributed Version Control, Collaborative Music Creation and Sharing

1. INTRODUCTION
Ultimate-guitar.com is a large guitarist community website that has a large number of guitar tablature. Due to the large amount of tablatures, overlapped (same) songs and many derivative versions are usually submitted to the archives. As a result, users are having a problem on how to choose a song. Moreover, if some parts in a guitar chord are incorrect, guitarists cannot continue to play the guitar chord. Since the current guitar chord archives do not provide the permission to edit an archived chord by anonymous users, the online music communities confront with a severe shortage of contributors and lack of collaboration among guitarists. To solve those problems, we came up with new software called ChordBook, which applies distributed version control system to keep track of guitar chord versions, reduce the redundancy and allow guitarists to do the collaboration.

Today's software development is usually done in a distributed environment [1]. Many researchers focus on the area and apply this concept to their work. Both Rocco et al [2] and Ladden et al [3] apply distributed version control to the classroom for improving class teaching. One of the common purposes of distributed version control is to enable groups of people work together on any kind of files without necessarily being connected to a common network. They can change, archive, merge, branch or synchronize files of which the system keeps a history. Another purpose is archiving and backup so any files in the system should be safe and free of data-loss.

In section 2, we purpose the architecture and key designs of ChordBook. In section 3, we demonstrate the software and show the empirical study on the application of distributed version control system. And we conclude everything up in section 4.

2. DESIGN
The architecture of ChordBook is separated into two parts. The first part is the software for guitarists to create and share guitar chords. The software is developed as an iPad application due to the portability. The second part is a server repository. It acts itself as a guitar chords archive, provides necessity services for communicating with a client-side application.

The architecture of ChordBook is based on four keys as follows:

1. Contribution – According to gain more guitar chords into the archive, we need to have more contribution from guitarists. ChordBook allows guitarists to easily distribute their guitar chords to the archive.
2. Distribution – According to guitarists are spread all over the world and guitar chords can be done by many guitarists, the server side will be able to keep track of versions and also have a good method to distribute the guitar chords.
3. Availability – ChordBook allow users to check out the songs that they want and it will automatically download and save them into their local storage.
4. Collaboration – A million heads is better than one. Guitarists are able to access any song sheets, which are shared on the server via ChordBook. If other guitarists notice there is a mistake in a song sheet or if they have more information on the song, they will be able to edit the song or provide more in-depth explanations. This collaboration key allows guitarists to help each other to create a better guitar chord.

In the ChordBook client, we use SQLite database as a local repository on iPad to keep track of versions of song sheets. Song sheets will be stored in the database as ASCII text based format as shown in Figure 1. ChordBook allows users to have various versions of song by creating branches for it. Figure 2 shows that a

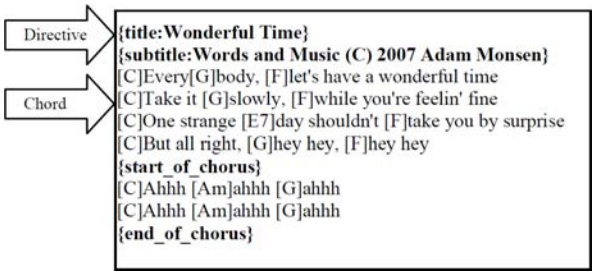


Figure 1. Guitar chord data format

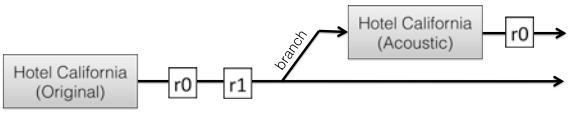


Figure 2. ChordBook Interface on iPad

user can create an acoustic version of the song, which is based on the original one. When the user is done with the song sheet and is ready to share or publish it, he/she can push that song sheet to the server repository. Users do not need moderators or system administrators to ask for a right to access repositories. The distribution process and rights management are done by guitarists among themselves. As a result, no special write access and no politic are needed.

3. DEMONSTRATION

The goal of our demonstration is to show how ChordBook can facilitate distributed version control concept in several aspects and discuss the potential applications that are feasible using distributed version control.

To combine musical ideas among guitarists, we get started from "clone" to download song sheet from server to iPad. If user notices there is a mistake or if he/she would like to provide more information on the song, then the user can edit the song, provide more in-depth explanations or discuss it among users. During the mediation, one can "commit" to his/her iPad to keep track of versions in local repository. If a user prefers to distribute or publish his/her guitar chord to the Internet, he/she can "push" the guitar chord to the server. If the server side finds a related song title name, the server will return a set of the result to the user. The user has to decide which one is a better title for the song. We believe that human decision can solve the problem of similar song titles repeatedly created. If the guitar chord is completely new to the server, it will be accepted right away. For an updated version, the server will first check whether that song version is up to date. If it is already up to date, the server will accept user's request. If it is not, this version will conflict and users will be guided to have discussions among users. ChordBook also have two ways to handle a spam and malicious users. First, the system allows users to report a spam and then will send a request to a moderator. Second, if the system detects some frequently changed or updated guitar chords within minutes, it will lock the song to be edited automatically and stop the update from users for 24 hours.

To avoid the duplication of song titles when there are more than one guitarists want to create the song and commit it into ChordBook, the distributed version control will prohibit the duplication of song titles to be created on the server.

To handle various types of a song as shown in figure 2, e.g., Hotel California will be able to create just once but ChordBook will allow this song to have many types of itself (i.e., acoustic, classic and etc.) using branch to represent types of that song.

4. CONCLUSION

In this paper we have described an empirical study on an application of distributed version control to music community. This application is novel in the sense that it fully utilizes a version control to keep track versions of guitar chord song sheets in chord archives. Our study could be applied to other artifact creations that need collaboration among users to keep track the versions of the artifact itself. For the future work we are planning to extend ChordBook to effectively treat other music instrument, enhance flexibility and scalability of the user interface.

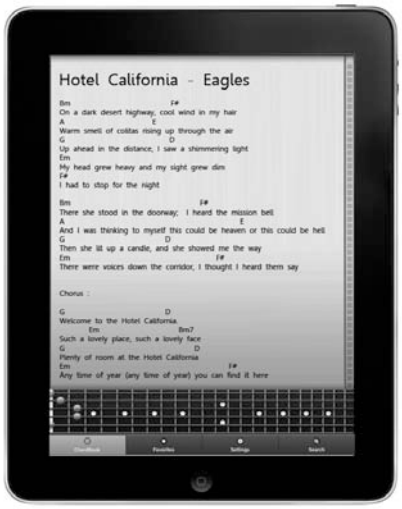


Figure 3. ChordBook Interface on iPad

5. ACKNOWLEDGMENTS

This research was supported by Nara Instituted of Science and Technology (NAIST), Japan, and Kasetsart University, Thailand. Thanks to anonymous reviewers for their comments.

6. REFERENCES

[1] Brian de Alwis and Jonathan Sillito. 2009. Why are software projects moving from centralized to decentralized version control systems?. In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (CHASE '09). IEEE Computer Society, Washington, DC, USA, 36-39.

[2] Daniel Rocco and Will Lloyd. 2011. Distributed version control in the classroom. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (SIGCSE '11). ACM, New York, NY, USA, 637-642.

[3] Oren Laadan, Jason Nieh, and Nicolas Viennot. 2010. Teaching operating systems using virtual appliances and distributed version control. In *Proceedings of the 41st ACM technical symposium on Computer science education* (SIGCSE '10). ACM, New York, NY, USA, 480

Poster

Empirical study on Web Crawling Process Monitoring Tool

[Extended Abstract]

Tanaphol Suebchua and Arnon Rungsawang
Massive Information & Knowledge Engineering
Department of Computer Engineering, Faculty of Engineering
Kasetsart University, Bangkok, Thailand
{job,arnon}@mikelab.net

ABSTRACT

Software monitoring and management is one of the powerful development tools in software testing for managing and tracking performance, resource consumption and object from a running program. It helps developer to indicate popular programming problem, such as Resource Leak and Deadlock. In this paper, we apply Java Management Extensions (JMX) to a web crawling system. Users are allowed to keep track of the crawling status and performance of the crawler in real-time and to do the crawling operations remotely. Furthermore, Adding JMX also enhance crawling performance by allowing application to work in parallel.

Categories and Subject Descriptors

C.2.5 [**Testing and Debugging**]: [Monitor, Tracing]; D.1.3 [**Programming Techniques**]: [Distributed programming]

General Terms

Management, Design

Keywords

Monitoring and Management tools, Web Crawler

1. INTRODUCTION

Internet is currently the large portion of informative data and also continues to grow unceasingly. Collecting every web page from the internet is impractical because the resources are limited. In general, search engine providers in each country need to collect web pages which are written in their own language as much as possible to make a complete index that can return the search results which satisfy their local users. To solve this kind of problem, in our ongoing research [2] , we have proposed a "language specific website crawler" framework as a method for gathering Thai-language web pages. However, our web crawler prototype can only be operated in text-mode and was not initially designed to work in parallel. This burdens users to start/stop process and configure crawler manually from one computing node to the others. As well, monitoring crawler status in real-time is difficult for end-users.

To provide additional monitoring and management capability, in this paper, we have proposed to integrate the JMX [1], an extensions of software monitoring and management framework for Java Platform which widely use among Java software developer, to our web crawling framework so that users can easily track crawler status in real-time and manage the crawling operation remotely through our GUI Client.

2. ARCHITECTURE

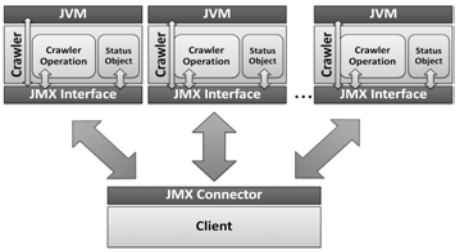


Figure 1: Architecture of Web Crawling Process monitoring tool

Figure 1 illustrates the Web Crawling Process monitoring tool architecture. The system consists of two parts, crawler and client modules. The crawler module is the part which we have mentioned earlier in introductory section. The client module is a GUI application as shown in Figure 2 which is responsible for handling request for crawling operation, crawler and crawling status, as well as JVM information through JMX extension in each crawler. Our design also supports crawlers to work in parallel by using the client as Job Dispatcher. For example, in a crawling task, if the target list of websites from which we have to collect data is too large, the client will split that list into proper smaller ones in order to dispatch to other crawling nodes in the system.

Figure 2: Web Crawling Process monitoring client

3. REFERENCES

[1] Oracle. Java management technology, 2011. http://www.oracle.com/technetwork/java /javase/tech/javamanagement-140525.html.

[2] P. Tadapak, T. Suebchua, and A. Rungsawang. A machine learning based language specific web site crawler. In *NBiS 2010*, pages 155 –161, sept. 2010.