Empirical Study of Change-Prone and Fault-Prone Method Prediction Focusing on Comment Ownership

Aji Ery Burhandenny, Takashi Nakano
Graduate School of Science and Engineering
Ehime University
Matsuyama, Ehime, Japan 790–8577
Email: {aji, nakano}@se.cite.ehime-u.ac.jp

Hirohisa Aman and Minoru Kawahara
Center for Information Technlogy
Ehime University
Matsuyama, Ehime, Japan 790–8577
Email: {aman, kawahara}@ehime-u.ac.jp

ABSTRACT

The existence of comments in method bodies is a double-edged sword, at one side it helps code reviewers to comprehend complex code while at the same time it could reflect the lack of confidence by the programmer concerning the clearness of their code. While comments can be useful clues to find problematic code, the effects may be vary from person to person. This paper proposes a novel metric "Ownership- Considered Comment Rate (OCCR)" that is enhanced from conventional comment rate by considering the difference among developers. The empirical study collects 75,204 Java methods from five popular open source software to understand the relationship between the developer and their code and its contribution to change-prone and/or fault-prone method prediction. The results confirmed the significant influence of OCCR for predicting changes and faults in methods.

Keyword: comment; developer; fault-prone; open source software

INTRODUCTION

The vast evolution in open source software (OSS) projects enforces the development team to be efficient in managing code evolutions and fault-related problems especially in code review phase. In order to shorten time in understanding source code, programmers often rely on the documentation provided with the code (Steidl, Hummel, & Juergens, 2013). One of the most fundamental documents is a comment statement written in a source file. Comments have been widely known as useful and important artifacts familiar to programmers (Souza, Anquetil, & Oliviera, 2005). While comments have no impact on the software systems' behavior and performance at all, they play a significant role in the program comprehension.

Aside from automatically generated comments, naturally written comments are humanly readable, descriptive and often express programmer's thought. These characteristics alongside programmer's manner become an interesting subject for study of software engineering due to the programmer-specific characteristic can be a potential factor either to increase or to reduce the chance of finding latent bugs in their

source code. This phenomenon ultimately triggering the "clean code practice" among the programmers exclusively when deciding whether to conduct code refactoring to their source code (Martin, 2008). Indeed, carefully-written (detailed) comments are said to be "deodorant" to mask code smell (Fowler, 1999); some empirical studies reported that well-commented programs tend to be more fault-prone in popular OSS products (Aman, Amasaki, Sasaki, & Kawahara, 2015; Aman et al., 2015).

While the previous work (Aman et al., 2015) empirically showed the value to focus comments in order to predict fault-prone and/or change-prone programs, they missed diversity of commenting, i.e., "who wrote those comments." One programmer may to write many comments with their code, but another programmer may prefer the code having no or less comments. Such differences in commenting preference would have significant impact on the change-prone and fault-prone program prediction. The key contribution of this paper is to empirically analyze the impact of considering ownership" toward a better performance of the change and/or fault prediction. We introduce a novel software quality prediction method based on the abnormal level of individual developer's comments. Moreover, we evaluate our proposed method with the primitive metrics that have been commonly used for the prediction model.

The remainder of the paper is organized as follows: Section 2 presents our research motivation and research question with empirical background supporting our work in this paper. Then, Sect.3 conducts an empirical analysis with five popular open source software projects and discusses the results. Section 4 describes previous work related to this paper. Finally, Sect.5 gives our conclusions and future work.

FAULT PREDICTION FOCUSING ON COMMENTS

Comments and Fault-proneness

Comments are useful artifacts to enhance the readability of the code as mentioned in Sect.1. However, they are sometimes used to compensate the lack of clearness in complicated code (Buse & Weiner, 2008). That is to say, comments have an aspect like a double-edged sword, and they are not always recommended to be written a lot. The code refactoring practice also warns detailed comments as deodorant for masking code smell (Fowler, 1999).

In our latest work (Aman et al., 2015), we had managed to prove that comment existence in Java method can significantly differ faulty methods and non-faulty ones. Nevertheless, a limitation of our approach was that we did not take into account the human factor such as the difference in each developer. For example, let us consider two developers A and B: developer A prefers to describe many comments in their code, and developer B does not like to write comments. Suppose developer A's average of comment rate (lines of comments per lines of code) and B's one are 0:2 and 0:01, respectively.

Now, if both of them developed programs with comment rate 0:2, then A's program looks at a normal level of commenting but B's program seems to be at a significantly abnormal level. Therefore, we can suspicious that the B's program may be something wrong, and the development manager should examine the reason why developer B

wrote such many comments this time. Such differences in commenting preference among developers may have a serious impact on fault-prone and/or change-prone method prediction studied in the previous work (Aman et al., 2015). This concern is our research motivation in this paper.

Research Question

Our main research question is: "Is the difference in comment rates among developers valuable to focus for predicting problematic code?"

Our primary goal is to study whether taking into account the average amount and variation of comment description per developer will effectively influence early prediction of code-change and bug fix. In order to answer the research question, we examine the predictive accuracy of bug fixes and code modifications with and without considering the abnormality comment rates by developers in the following section.

EMPIRICAL WORK

This section elaborates our experiments and analytical results on data sets from popular OSS projects to explore our research question regarding the value of developer's diversity for early prediction of code-change and bugs fix. In order to quantitatively measure the relationship between per-developer in comment rate and its worthiness to predict prospective code changes and bug fixes in Java methods, we perform data collection for the comment rate of each developer, and statistically analyze the trends.

We intended to choose the OSS systems for analysis with the following criteria: 1) To eliminate commonalities that might be valid only for particular OSS systems, the chosen products should be varied in their size and domains. 2) The selected OSS should have similar implementation language, therefore the differences between languages would not affect the results. For our study purpose, Java was the first choice due its compatibility with our mining tools. 3) Each project should be relatively popular confirmed by having a reasonable number of releases and currently active up to nowadays. 4) For each project, all the source code and relevant information concerning change history and their logs, and other feature requests should be available in their repository.

Table 1. Surveyed Software

Software	Investigation Period	#Source Files
Free Mind	Feb. 2011 – Sept.2015	507
Squirrel SQL Client	June 2001 – Sept.2015	3,855
Hibernate	June 2007 – Sept.2015	3,861
JabRef	Oct. 2003 – Sept.2015	585
eXo Platform	Mar.2007 – Sept.2015	172
Total		8,980

Based on those criteria, we selected five prominent OSS products which all are managed using Git (see Table 1). The following metrics are harvested from the surveyed products: lines of comments in a method body (inner comments; LIC) that

each developer has created (first edition), lines of comments followed by a method declaration (documentation comments; LDC), lines of code (LOC), cyclomatic complexity (CC), and frequency of code changes and bug fixes occurred in the method after their first release.

Data Collection

To perform our empirical analysis for the above research question, we conducted a data collection in the following procedure:

1. In order to perform our analysis efficiently, we made a local copy (clone) of the targeted repository.
2. By utilizing JavaMethodExtractor[1] we perform a syntax analysis to extract methods' bodies from the source files.
3. Then we examine the change history of each method using diff utility[2]. We track the frequency of changes and bug fixes up to the latest version according to the information provided in their commitment log. By this way, we are able to distinguish which method has been changed during each commitment. Similar to the previous work, we identify bug fixes, when bug-fix-related keywords or bug IDs appeared in commitment log (Sliwerski, Zimmermann, & Zeller, 2005).
4. Next, we decide the owner of each method: we define a method's owner is who created the method. Method owners are identified by their email addresses appeared in the commitment logs.
5. Finally, we measure the initial version of each method by using the above metrics including LIC, LDC, LOC and CC.

Table 2 shows the number of methods and the number of developers who are the owners of the initial version of methods.

Table 2. Number of Methods and Number of Developers Who Created Initial Version of Methods

Software	#Developers	#Methods
Free Mind	3	6,975
Squirrel SQL Client	10	29,309
Hibernate	91	31,824
JabRef	10	5,606
eXo Platform	36	1,490
Total	150	75,204

Preliminary Study: Comment Ownership

Prior to analyze the comments rate, in order to justify whether to consider the differences in the developer or not, we need to confirmed that the comment rates vary among developers for each tested product. We check the comment rates to LOC in each developer for both documentation and inner comment rates (LDC/LOC and LIC/LOC) using analysis of variance (ANOVA), and confirmed the trend statistically.

[1] http://se.cite.ehime-u.ac.jp/tool/
[2] https://www.gnu.org/software/diffutils/

Table 3. ANOVA Results (Documentation Comment Rate)

Software	Degree of Freedom	p-value
Free Mind	2	0.008
Squirrel SQL Client	9	$< 2 \times 10^{-16}$
Hibernate	90	$< 2 \times 10^{-16}$
JabRef	9	0.352
eXo Platform	35	0.049

Table 4. ANOVA Results (Inner Comment Rate)

Software	Degree of Freedom	p-value
Free Mind	2	0.016
Squirrel SQL Client	9	$< 2 \times 10^{-16}$
Hibernate	90	$< 9.44 \times 10^{-11}$
JabRef	9	0.573
eXo Platform	35	0.001

Tables 3 and 4 shows the ANOVA results for documentation comment and the inner comment respectively. Significant developer differences in both documentation and inner comment rate have been confirmed, except for JabRef. JabRef is the only product that has inner comment rate's pvalue exceeded a significance level (5%); JabRef seems not to be a worthy subject in the following analysis, so we will analyze the remaining four OSS products. We can say that the difference of developer has an impact on the amount of comments written on inside and outside of a method (inner comments and documentation comments).

Analysis (1) Ownership-Considered Comment Rate ($OCCR$) VS Conventional Comment Rate

We have confirmed the differences of comment rates among developers in tested products. In the rest of this section, we assess the worth of comment rates for predicting change-prone and/or fault-prone methods. We start our analysis by comparing two types of comment rates:

1. Conventional comment rates (the lines of comments divided by the lines of code),
2. Proposed comment rates that take into account the difference among developers.
3. We call the first type as Conventional Comment Rates (CCR) and the later one "Ownership-Considered Comment Rate ($OCCR$)."

First, we calculate the mean and standard deviation of the comment rate for each developer. Based on this measurement, we evaluate the amount of comments of each method as follows:

Let $r_{in}(m)$ to be the inner comment rate of method m, i.e., $r_{in}(m) = LIC/LOC$ of m, and method m's owner is developer d. then we define $\text{Score}(m)$ by the following equation:

$$\text{Score}\ (m) = \frac{r_{in}(m) - \mu_d}{\sigma_d}$$

Where μ_d and σ_d are the mean of d's comment rates and the standard deviation of d's ones, respectively. When $\mathrm{Score}(m)$ is a positive value, it indicates the comment rate is greater than the developer d's average (μ_d). Similarly, when its value is negative, it means that the comment rate is less than their average. We want to normalize the variation by dividing the standard deviation of d's comment rate (σ_d). Then, we consider $|\mathrm{Score}(m)|$ to be an index of abnormality with regard to $\mathrm{r}(m)$ by the developer d. In this paper, we define $|\mathrm{Score}(m)|$ as a novel metric, Ownership-Considered Inner Comment Rate (OCICR):

$$\mathrm{OCICR}(m) = |\mathrm{Score}(m)|$$

We can define a similar metric for documentation comment rate—Ownership-Considered Documentation Comment Rate (OCDCR) — by replacing "inner comments" with "documentation comments" in the above definition. From hereafter, we refer both $OCICR$ and $OCDCR$ as "Ownership-Considered Comment Rate ($OCCR$) for the sake of convenient.

On the other hand, we can consider another type of $\mathrm{Score}(m)$ with using the mean of all developer's comment rates (μ_{all}) and the standard deviation of them (σ_{all}), as follows:

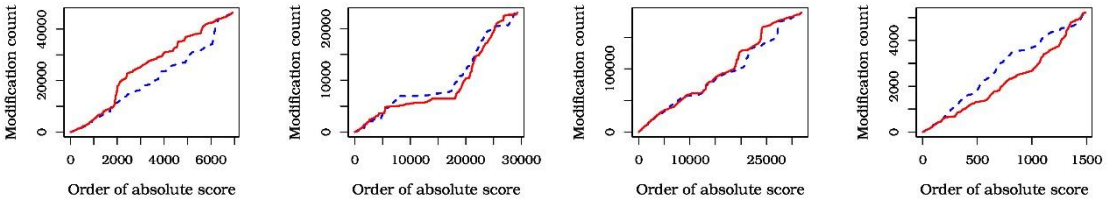
$$\overline{\mathrm{Score}}(m) = \frac{r_{in}(m) - \mu_{all}}{\sigma_{all}}$$

While $|(Score(m)|$ also denotes an abnormality of comment rate for method m, it does not take into account its ownership. We consider $|(Score(m)|$ to be a conventional way to evaluate the comment rate, and compare it with our proposed $OCCR$.

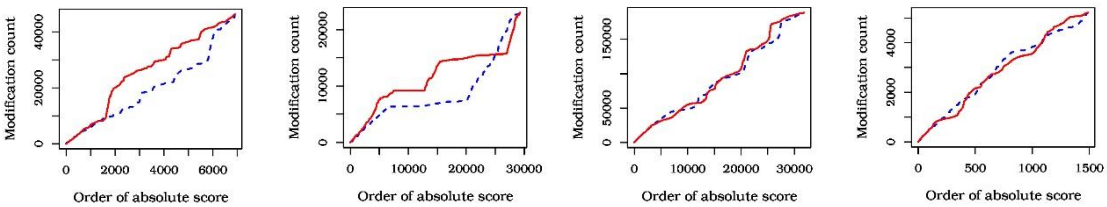
Figure 1 compares the powers of change-prone method predictions by $OCCR$ and the conventional score. Its horizontal axis corresponds to the methods in the decreasing order of comment rate's abnormality, and the vertical axis signifies the cumulative number of code-change events occurred in the methods. That is to say, an earlier growth of a curve means a better performance in predicting change-prone methods. In the figure, solid red lines correspond to the results of $OCCR$ and dashed blue lines signifies the results by using the conventional scores. Similarly, Fig.2 compares the power of fault-prone method predictions by $OCCR$ and the power of that by the conventional score.

In order to evaluate the effect to take into account the differences in comment rate among developers, we introduce the following improvement rate δ :

$$\delta = \frac{a_1 - a_0}{a_0}$$

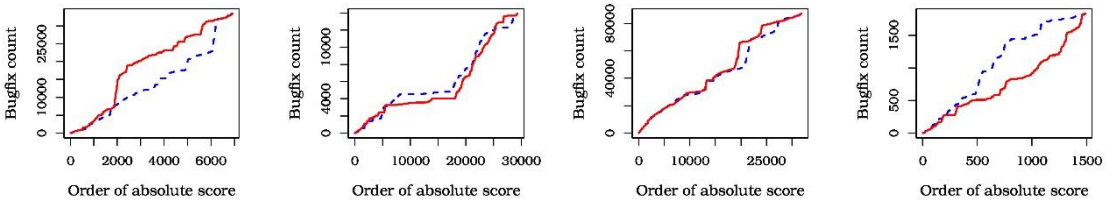


(a) Doc Comment (FreeMind) **(b) Doc Comment (SQuirrel SQL Client)** **(c) Doc Comment (Hibernate)** **(d) Doc Comment (Platform)**

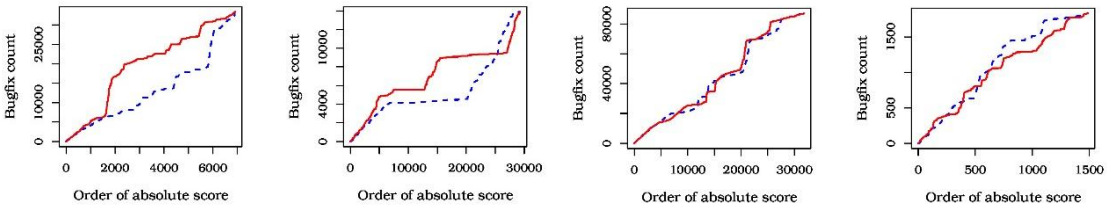


(e) Inner Comment (FreeMind) **(f) Inner Comment (SQuirrel SQL Client)** **(g) Inner Comment (Hibernate)** **(h) Inner Comment (Platform)**

Figure 1. Cumulative number of modifications occurred at methods that are sorted in the decreasing order of their comment rates' abnormality.



(a) Doc Comment (FreeMind) **(b) Doc Comment (SQuirrel SQL Client)** **(c) Doc Comment (Hibernate)** **(d) Doc Comment (Platform)**



(e) cxcxInner Comment (FreeMind) **(f) Inner Comment (SQuirrel SQL Client)** **(g) Inner Comment (Hibernate)** **(h) Inner Comment (Platform)**

Figure 2. Cumulative number of bug fixes occurred at methods that are sorted in the decreasing order of their comment rates' abnormality.

Where a_1 represents the area under the curve of $OCCR$ in Figs. 1 and 2, and a_0 signifies the area under the curve using the conventional score in those figures. Since a larger area means a better performance in predicting change-prone or fault-prone methods, indexes the improvement of their prediction power by using $OCCR$, i.e., taking into account the difference in comment rate among developers.

Table 5. Improvement Rate with regard to Documentation Comments.

Software	#Modifications	#Bug Fixes
Free Mind	0.237	0.352
Squirrel SQL Client	-0.074	-0.074
Hibernate	0.057	0.046
eXo Platform	-0.199	-0.300
Average	0.004	0.003

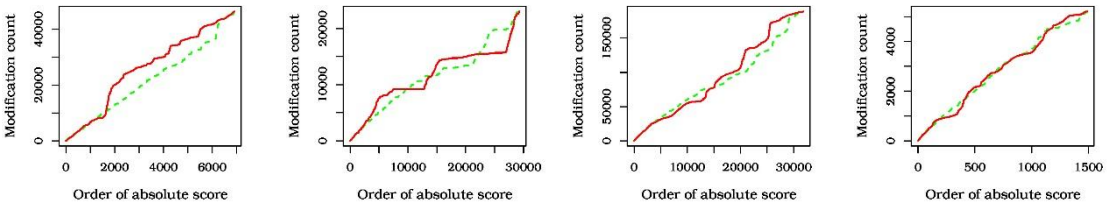
Table 6. Improvement Rate with regard to Inner Comments.

Software	#Modifications	#Bug Fixes
Free Mind	0.318	0.488
Squirrel SQL Client	0.320	0.296
Hibernate	0.026	0.017
eXo Platform	-0.009	-0.070
Average	0.131	0.146

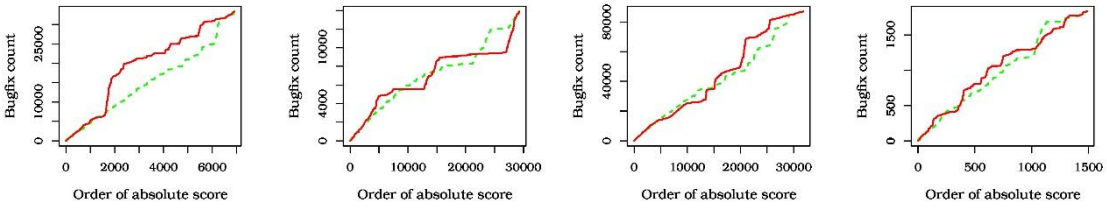
Tables 5 and 6 describe the improvement rates with regard to the documentation comments and inner comments, respectively. Table 5 shows that $OCCR$ is not resourceful for documentation comment (improvement rate $< 1\%$ on average). In contrast, Table 6 justifies a part of our research question that taking into account the differences in developer's comment rate can be useful to an early prediction of problematic code—change-prone and fault-prone methods: their improvement rates are in $13\% - 15\%$ for both change-prone method prediction and fault-prone method prediction. It seems that the differences in developer's inner comment rate have a promising effect on change-prone and/or fault-prone method prediction rather than the documentation comment rates. Therefore, we will examine the change-prone and fault-prone prediction powers of ownership-considered inner comment rate by comparing with the conventional metrics, lines of code (LOC) and cyclomatic complexity (CC) in Sect. 3.4 and Sect. 3.5, respectively.

Analysis (2) OCCR vs Lines of Code (LOC)

In previous analysis, we have confirmed the helpfulness of OCICR (OCCR for inner comments) for early detection of code changes and bug fixes in Java methods. In this analysis, we compare OCICR with LOC, which is a popular metric commonly used for predicting fault-prone programs. Our procedure of the comparative study is similar to Analysis (1) in Sect.3.3: we cumulate the numbers of modifications occurred at methods that are sorted in the decreasing order of OCICR, and that are sorted in the decreasing order of LOC, respectively. We also compute the cumulative numbers of bug-fixes occurred at methods in those orders.



(a) #Modifications (FreeMind) **(b) #Modifications (SQuirrel SQL Client)** **(c) #Modifications (Hibernate)** **(d) #Modifications (Platform)**



(e) #Bug Fixes (FreeMind) **(f) #Bug Fixes (SQuirrel SQL Client)** **(g) #Bug Fixes (Hibernate)** **(h) #Bug Fixes (Platform)**

Figure 3. Cumulative numbers of modifications and bug fixes occurred at methods that are sorted in the decreasing order of their comment rates abnormality and in the decreasing order of LOC.

Figure 3 shows the curves of cumulative numbers. In the figure, solid red lines correspond to the results of OCICR and dashed green lines signifies the results of LOC. Table 7 shows the improvement rates defined in Eq.(4) where a1 and a0 are the area under the curve of OCICR and that of LOC, respectively.

From Table 7, we can say that $OCICR$ is at almost the same or better performance than LOC in predicting change-prone methods and fault-prone ones: OCICR improves the prediction power by 2.7% and 6.7% on average compared to LOC for change-prone method prediction and fault-prone method prediction, respectively. FreeMind showed the best performance with OCICR, which improves around 19% for predicting modification occurrences and nearly 29% for predicting bug fixes occurred in the methods. While the bug fix prediction in SQuirreL SQL Client and the code modification prediction in eXo Platform show negative improvement rates, both of them are at small levels (-2% and -0.1%). Therefore, for most of projects, OCICR has a usefulness at the same with or better than LOC in predicting change-prone methods and fault-prone methods.

Table 7. Improvement Rates: *OCICR* VS *LOC*.

Software	#Modifications	#Bug Fixes
Free Mind	0.192	0.391
Squirrel SQL Client	0.003	0.251
Hibernate	0.060	0.000
eXo Platform	-0.001	-0.044
Average	0.027	0.147

Analysis (3) OCCR vs Cyclomatic Complexity (CC)

Next, we conduct another comparative study with using cyclomatic complexity (CC). CC has also been widely known as a useful metric for predicting fault-prone programs. Our analysis procedure is similar to the one described in Sect.3.4, where we replace "LOC" with "CC." Due to limitations of space, we omit a figure of cumulative numbers of modifications and bug-fixes occurred at methods in the decreasing order of metric values like Fig.3, and we show only a table of improvement rates (see Table 8).

Table 8. Improvement Rates: *OCICR* VS *CC*

Software	#Modifications	#Bug Fixes
Free Mind	0.259	0.391
Squirrel SQL Client	0.270	0.251
Hibernate	0.012	0.000
eXo Platform	-0.001	-0.044
Average	0.131	0.147

As shown in Table 8, OCICR seems to be more useful than CC in predicting change-prone methods and faultprone ones: the improvement rates are about 13% and 14% on average for code modification prediction and bug fix prediction, respectively. While eXo Platform showed small disimprovements (-0.1% and -4.4%) similar to Analysis (2) (see Table 7), the remaining products showed that OCICR is at almost the same level with CC or better than CC in the predictions. Especially, FreeMind and SQuirreL SQL Client showed over 25% improvements. Therefore, we can say that OCICR can be a more promising metric than Cyclomatic Complexity.

Threats to Validity

This empirical work used five popular OSS products from different domains and in different size for a generality of our empirical results. However, we had to limit our subjects to Java products because of our data collection tool. While the language limitation may be our threats to validity, the fundamental concept of commenting code is common to almost all modern programming languages, so our results would not lose its generality for other languages.

If there was a specific coding convention in the surveyed OSS projects and they controlled developers on how to write comments, such a control can be our threats to

validity. JabRef did not show statistical difference in comment rate among developers, so it might be controlled by a certain coding rule. However, we excluded JabRef from our subjects in the main analyses (1)–(3), and the remaining products showed diversities in comment rates among developers. Therefore, we can say that the influence of coding convention would not be concern in our empirical results.

RELATED WORK

Recently, there are several studies that investigate the relationship between developer activities to the fault proneness (Robbes & Rothlisberger, 2013; Rahman & Devanbu, 2011). However, all of their work do not consider comment as one of important factors in predicting fault-prone programs. Aman et al. (2015) stressed on the importance of comments and their relationship to the fault-proneness. They conducted empirical analyses to show the worth to focus on Lines of Comments (LCM) as a useful metric along with LOC and CC in analyzing fault-proneness of Java methods. While their work is our previous work, they missed to take into account the diversity in commenting manners among developers, so an empirical analysis using "ownership-considered comment rate ($OCCR$)" is our key contribution in this paper.

CONCLUSION AND FUTURE WORK

In this paper, we aimed to investigate the developer differences in commenting their code, and the influence of such a diversity to the accuracy of change-prone and/or fault-prone method prediction. In our first analysis, we had confirmed the efficiency of our metric "Ownership-Considered Comment Rate (OCCR)" in detecting change-prone and fault (bug)-prone methods compare to the conventional comment rate. Our second and third analyses revealed that OCICR (OCCR for inner comments) can be useful metric in predicting change-prone methods and fault-prone methods, and its usefulness is at almost the same level with or better than conventional popular metrics, LOC (Lines of Code) and CC (Cyclomatic Complexity). Therefore, this paper successfully showed the importance of taking into account the developers' differences in commenting their code for more accurate change-prone and/or fault-prone method prediction.

Our future work includes:

1. Examining the evolution of comments by developers through the software development and maintenance;
2. Analyzing the contents of comments with the natural language processing methods toward a better understanding the relationship between a developer and their comments.

ACKNOWLEDGEMENT

This work was supported by JSPS KAKENSHI Grant Number 25330083.

REFERENCES

Aman, H., Amasaki, S., Sasaki, T., & Kawahara, M. 2015. Empirical Analysis of Change-Proneness in Methods Having Local Variables with Long Names and Comments. ***Proc. 2015 ACM/IEEE Int'l Symp. Empirical Softw. Eng. and Measurement***: 50–53.

Aman, H., Amasaki, S. Sasaki, T., & Kawahara, M. 2015. Lines of Comments as a Noteworthy Metric for Analyzing Fault-Proneness in Methods. ***IEICE Trans. Inf. & Syst.*** E98-D(12): 2218–2228.

Buse, R.P. & Weiner, W.R. 2008. A metric for software readability. ***Proc. Int'l Symp. Softw. Testing and Analysis***: 121–130.

Fowler, M. 1999. ***Refactoring: Improving the Design of Existing Code***. Addison-Wesley Longman Publishing.

Martin, R.C. 2008. ***Clean Code: A Handbook of Agile Software Craftsmanship***. Prentice Hall PTR.

Rahman, F., & Devanbu, P. Ownership, Experience and Defects: A Fine-Grained Study of Authorship. ***Proc. 2011 33rd Int'l Conf. Softw. Eng***: 491–500.

Robbes, R., & Rothlisberger, D. 2013. Using Developer Interaction Data to Compare Expertise Metrics. ***Proc. 2013 10th IEEE Working Conf. Mining Softw. Repositories***: 297–300.

Sliwerski, J., Zimmermann, T., & Zeller, A. 2005. When do changes induce fixes? ***ACM SIGSOFT Softw. Eng. Notes***. 30(4): 1–5.

Souza, S. C. B., Anquetil, N., & Oliveira, K. M. 2005. A Study of the Documentation Essential to Software Maintenance. ***Proc. 23rd Annual Int'l Conf. Design of Communication: Documenting &Amp; Designing for Pervasive Information***: 68–75.

Steidl, D., Hummel, B., & Juergens, E. 2013. Quality analysis of source code comments. ***Proc. 2013 IEEE 21st Int'l Conf. Program Comprehension***: 83–92.