An Empirical Analysis on Fault-proneness of Well-Commented Modules

Hirohisa Aman
Department of Computer Science
Graduate School of Science and Engineering, Ehime University
Matsuyama, Japan
aman@cs.ehime-u.ac.jp

***Abstract*—Comment statements are useful to enhance the readability and/or understandability of software modules. However, some comments may adjust the readability/understandability of code fragments that are too complicated and hard to understand—a kind of code smell. Consequently, some well-written comments may be signs of poor-quality modules. This paper focuses on the lines of comments written in modules, and performs an empirical analysis with three major open source software and their fault data. The empirical results show that the risk of being faulty in well-commented modules is about 2 to 8 times greater than non-commented modules.**

***Keywords*-comment; code smell; fault-prone module; prediction**

I. Introduction

Writing readable program modules is the most basic quality-assurance activity in software development. To enhance the readability of source code, comments are widely known as effective entities [1], and programmers are encouraged to write helpful comments into their program modules [2].

While comments impact the readability of modules, an excessive amount of comments are discouraged [3], [4]. If a module needs many comments to improve its readability, the module includes code fragments that are complex and hard to understand, i.e., bad code. Kernighan and Pike [3] recommended to rewrite such bad code and make it clearer, rather than adding comments to explain the bad code. Fowler [4] pointed out the presence of many comments is a sign of "code smell" to be refactored. While comments are not "bad smell", they seem to be a kind of "air freshener (deodorant)" for a smelled code. Therefore, a well-commented module might be a complicated module that is hard to understand without its well-written comments, and such a complicated module might be a haven for potential faults.

In this paper, we empirically examine if the well-commented modules are fault-prone. For many modules in popular open source software, we measure the amount of comments and collect data assessing the presence or absence of faults which are found after the software release. Then, we perform an empirical analysis on the fault-proneness of well-commented modules, statistically.

The remainder of this paper is organized as follows. Section II introduces two metrics to quantify the amount of comments and the fault-proneness. Section III proposes a statistical method to automatically extract "well-commented" modules by considering the distribution of metric values according to the amount of comments and the module size. Then, Section IV presents our empirical analysis to examine the fault-proneness of well-commented modules. Section V describes our related work. Finally, Section VI concludes the paper.

II. Metrics

This section introduces two metrics: the lines of comments (LCM) and the fault rate (FR). LCM is a metric for the amount of comments written in a module, and FR is a metric for the fault-proneness of modules.

A. Lines of Comments (LCM)

We define a metric to quantify the amount of comments written in a module; in this paper, we consider a Java source file to be a module.

Definition 1 (Lines of Comments: LCM)

For a source file, let the lines of comments (LCM) be the total number of source lines in which one or more comments are written. However, the following types of comments are excluded from the LCM measurement:

- Any comments written outside the method[1] bodies (e.g., Javadoc).
- Comment out—disabling a code fragment.

□

LCM is a measure of comments written in method bodies. The larger LCM value indicates that more comments are written to explain the code fragments, i.e., more memos by the developers. It should be noted that LCM does not count any "doc" (Javadoc) comments[2], since a doc comment is used to make a manual and has a different purpose from the above "comments." Further analysis of the doc comments will be our future work.

Comment outs are also excluded from our LCM measurement as they have different meanings—they are not memos

[1]The term "method" may be replaced with "function" or "procedure" according to the programming language used in the source file.

[2]A doc comment is described just before (outside) the corresponding method's definition.

by the developers. Since the comment outs have different purposes and meanings from the above normal comments, we should also exclude the comment outs in our analysis. In order to accurately determine whether a comment fragment is a "comment out" or not, we should query each developer for each fragment. However, such a follow-up investigation is practically impossible, so we use the following simple detection algorithm [5] instead of such a follow-up study.

Algorithm 1 (Simple Detection of Comment Out [5])

Let C be the content of given comment, which excludes the special strings indicating the start and the end of comment; C is the string from just behind "`/*`" to just before "`*/`" in the case of traditional comment [6], and that is the string from just behind "`//`" to the end of line in the case of end-of-line comment.

If the last non-white-space character appeared in C is one of 1) semicolon "`;`"2) left curly brace "{" or 3) right curly brace "}", then regard C as a comment out.

□

The above algorithm is a simple detection algorithm focusing on the tailing character in the comment. Because of the simpleness, it can be easily implemented; MethodCommentCounter[3] is a tool based on the above algorithm, which can count lines of comments, comment outs and doc comments, separately. Although the above algorithm is not perfect, the algorithm has a high level of accuracy detecting comment outs: it has been reported that the algorithm could detect about 98.9% of comments and comment outs, successfully [5].

B. Fault Rate (FR)

To discuss the fault-proneness quantitatively, we introduce the notion of fault rate (FR) [5] as well.

Definition 2 (Fault Rate: FR)

For a set of modules (source files), define the fault rate (FR) in the set as the following equation:

$$\mathrm{FR} = \frac{\text{Number of faulty modules}}{\text{Number of modules}} ,$$

where a faulty module means a module in which a fault is found after its release.

□

FR is the proportion of faulty modules in a given module set. Hence, the higher FR means that a module included in the set is more suspect to have a fault, i.e., fault-prone.

FR is a simple metric based on the presence or absence of fault in a module, and this paper focuses on the following simple question in this paper: "whether well-commented modules are more fault-prone or not." While it is also possible to consider more detailed metrics of fault-proneness such as the number of faults or the fault density (bug density), first we should answer the simple question above using FR. We would like to conduct further analysis using such detailed metrics in the future.

[3]http://www.hpc.cs.ehime-u.ac.jp/~aman/project/tool/MethodCommentCounter.html .

III. Regression Model-Based Method to Extract Well-Commented Modules

Any modules can be classified into two categories: the non-commented module (LCM$=0$) set and the commented module (LCM>0) set. This section proposes a method to pick out well-commented modules from the latter (commented; LCM>0) set.

In general, the larger modules (having higher LOC values) tend to become harder to understand, and result in more chance of having comments (higher LCM values) as their supplemental explanations or memos by the developers. From our experience, we can expect a certain level of positive correlation between LCM and LOC. Then, we propose to take advantage of such a correlation to make a criterion for determining "well-commented" modules whose LCM values are relatively higher than the others of the same size (same LOC).

For the commented modules, let us consider a scatter diagram whose horizontal axis and vertical axis correspond to LOC and LCM, respectively. Then, we draw the regression line on the scatter diagram, where LOC is the independent variable and LCM is the dependent variable (see Fig.1); we might have to perform the log transformations for LOC and LCM if they have skewed distributions [7].

The modules plotted in the upper side of the regression line correspond to "well-commented" modules whose LCM values are relatively high. We will call the other modules as "less-commented" modules, for convenience of classification (see Fig.1).

The equation of the regression line can be written as:

$$\widehat{\mathrm{LCM}} = a \cdot \mathrm{LOC} + b \ , \quad \text{or} \tag{1}$$

$$\log(\widehat{\mathrm{LCM}}) = a \cdot \log(\mathrm{LOC}) + b \ , \tag{2}$$

where a is the slope of the line, and b is the intercept (Y-intercept); $\widehat{\mathrm{LCM}}$ means the estimate of LCM, calculated by LOC.

Consequently, we can consider the following three sets of modules:

- Well-commented modules: LCM $> \widehat{\mathrm{LCM}}$,
- Less-commented modules[4]: LCM $\leq \widehat{\mathrm{LCM}}$,
- Non-commented modules: LCM$=0$.

A well-commented module means a module whose (actual) LCM value is greater than its estimate.

[4]Although a module of LCM $= \widehat{\mathrm{LCM}}$ should not be "less" but "average"-commented one, we will include such module in the category of less-commented module, for convenience of classification.

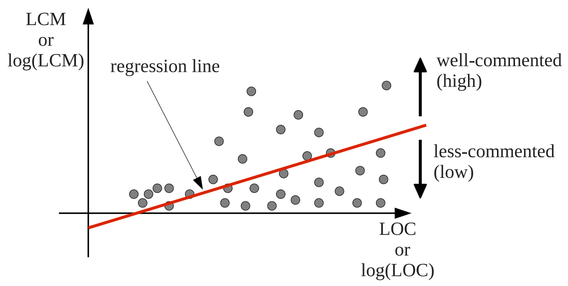

Figure 1. Well-commented modules and less-commented modules.

IV. Empirical Analysis

This section presents an empirical analysis with three popular large-scale or medium-scale open source software. For each software, the fault rate (FR) in well-commented modules is compared with FR in less-commented ones and FR in non-commented ones. The goal of our analysis is to show statistically that well-commented modules are more fault-prone than the others.

A. Empirical Object

This study uses the source code of Eclipse[5] (ver. 2.0, 2.1 and 3.0), Apache Tomcat[6] (ver.6.0.0) and Apache Ant[7] (ver. 1.3, 1.4, 1.5, 1.6 and 1.7). The reasons why we use them as our empirical objects are that they are popular software and satisfy the following three requirements:

1) The source code is available to anyone for free.
2) The source code is written in Java.
3) The fault data—whether a fault was found after its release or not—is easily available for each module.

The first two requirements are from our measurement tool—MethodCommentCounter. The third requirement is necessary to analyze the fault-proneness of modules. We got their fault data from PROMISE data repository [8]. For the sake of argument, this study regards a Java source file as a module.

The following two types of modules are filtered out beforehand, since they might be noise in our analysis.

1) The modules overlapped with earlier version:
   A module overlapped with earlier version may have a similar LCM value with its earlier version, but its fault(s) had probably been fixed already. Thus, such a module might be a noise data to understand the relationship between LCM and fault-proneness of our modules.
2) Large modules—the outliers[8] in LOC:
   The outliers in LOC are the minority modules, but they can affect the regression analysis between LCM and LOC. Thus, this study omits such outliers in the analysis.

Table I
Number of modules (classes) used in our analysis.

software	version	number of modules	number of faulty modules	data set
Eclipse	2.0	5521	650	(1)
	2.1	7	2	
	3.0	5447	680	(2)
Tomcat	6.0.0	669	38	(3)
Ant	1.3	104	11	(4)
	1.4	48	16	
	1.5	111	7	
	1.6	65	21	
	1.7	349	38	

Table I shows the numbers of modules used in our analysis. While we wanted to perform our analysis for each version of each software, we grouped them into the the following four sets, due to the small number of modules.

- Data set (1): Eclipse 2.0 & 2.1,
- Data set (2): Eclipse 3.0,
- Data set (3): Tomcat,
- Data set (4): Ant.

B. Categorization Criteria based on Regression Model

For each data set, we checked the distributions of LOC values and LCM values, and derived the regression line for extracting "well-commented" modules.

1) Eclipse 2.0 & 2.1:

In the modules of Eclipse 2.0 & 2.1, 2579 out of 5528 modules were no-commented ones (LCM = 0). To categorize the remaining 2949 modules into the well-commented ones and less-commented ones, we define the boundary between them by using the regression line as mentioned in Sect.III. The regression equation between LCM and LOC was obtained from the data as:

$$\log(\widehat{\mathrm{LCM}}) = 0.831 \cdot \log(\mathrm{LOC}) - 2.142, \qquad (3)$$

where both LCM and LOC are performed the log transformations[9] because both of their distributions are right-skewed[10]. Then, we can categorize the modules into the following three groups:

- Well-commented modules ($\widehat{\mathrm{LCM}} < \mathrm{LCM}$): 1590 ones,

[5]http://www.eclipse.org/ .
[6]http://tomcat.apache.org/ .
[7]http://ant.apache.org/ .
[8]Let Q_1 and Q_3 be the 25 percentile and 75 percentile in LOC distribution, respectively. A module whose LOC is greater than $Q_3 + 1.5(Q_3 - Q_1)$ or less than $Q_1 - 1.5(Q_3 - Q_1)$ is an outlier [7].

[9]The correlation coefficient between log(LCM) and log(LOC) was 0.613.
[10]The skewness of a distribution can be calculated as $(Q_3 + Q_1 - 2Q_2)/(Q_3 - Q_2)$, where Q_1, Q_2 and Q_3 are the 25, 50 and 75 percentiles in the distribution, respectively [7]. If the distribution is right-skewed, its skewness is positive. The skewness of LCM and LOC were 0.667 and 0.4212, respectively.

Table II
NUMBER OF MODULES CATEGORIZED INTO THREE GROUPS: NON-COMMENTED MODULES, LESS-COMMENTED ONES AND WELL-COMMENTED ONES.

data set	comment category		
	non	less	well
Eclipse 2.0 & 2.1	2579	1359	1590
Eclipse 3.0	2476	1435	1536
Tomcat	391	133	145
Ant	368	155	154

- Less-commented modules ($0 < \text{LCM} \leq \widehat{\text{LCM}}$): 1359 ones,
- Non-commented modules ($\text{LCM} = 0$): 2579 ones.

2) Eclipse 3.0, Tomcat and Ant:

Similar to the data set of Eclipse 2.0&2.1, we calculated the regression equation between LCM and LOC by considering their distributions for each data set Eclipse 3.0, Tomcat and Ant. The regression equations were as follows[11]:

For the data set Eclipse 3.0,

$$\log(\widehat{\text{LCM}}) = 0.896 \cdot \log(\text{LOC}) - 2.446, \tag{4}$$

for the data set Tomcat,

$$\log(\widehat{\text{LCM}}) = 0.764 \cdot \log(\text{LOC}) - 1.905, \tag{5}$$

and for the data set Ant,

$$\log(\widehat{\text{LCM}}) = 0.641 \cdot \log(\text{LOC}) - 1.468. \tag{6}$$

Then, the modules can be categorized as shown in Table II.

C. Statistical Comparison in FR

In order to quantitatively compare the fault-proneness among non-commented modules, less-commented ones and well-commented ones, we calculated FR value for each group of modules in each data set. Table III and Fig.2 show the calculation results.

Although there was an inter-individual variability in FR, all the sets showed increasing tendencies of FR values: the well-commented modules showed the highest FR, followed by the less-commented modules and non-commented ones for each data set. We performed the Cochran-Armitage test [9] for their increasing tendencies[12]. The results are as follows.

- Eclipse 2.0 & 2.1: $\chi^2 = 358.1,\ df = 1,\ p < 0.001$;
- Eclipse 3.0: $\chi^2 = 151.4,\ df = 1,\ p < 0.001$;
- Tomcat: $\chi^2 = 35.2,\ df = 1,\ p < 0.001$;
- Ant: $\chi^2 = 11.1,\ df = 1,\ p < 0.001$.

[11]The all skewness were greater than 0.2. The correlation coefficients between log(LCM) and log(LOC) were 0.618, 0.550 and 0.423 for data sets Eclipse 3.0, Tomcat and Ant, respectively.

[12]We performed the test by using the function `prop.trend.test` of R (http://www.r-project.org/).

Table III
FR VALUES IN THREE GROUPS: NON-COMMENTED MODULES, LESS-COMMENTED ONES AND WELL-COMMENTED ONES.

data set	comment category		
	non	less	well
Eclipse 2.0 & 2.1	0.035 $(\frac{90}{2579})$	0.149 $(\frac{203}{1359})$	0.226 $(\frac{359}{1590})$
Eclipse 3.0	0.064 $(\frac{158}{2476})$	0.160 $(\frac{229}{1435})$	0.191 $(\frac{293}{1536})$
Tomcat	0.020 $(\frac{8}{391})$	0.053 $(\frac{7}{133})$	0.159 $(\frac{23}{145})$
Ant	0.098 $(\frac{36}{368})$	0.168 $(\frac{26}{155})$	0.201 $(\frac{31}{154})$

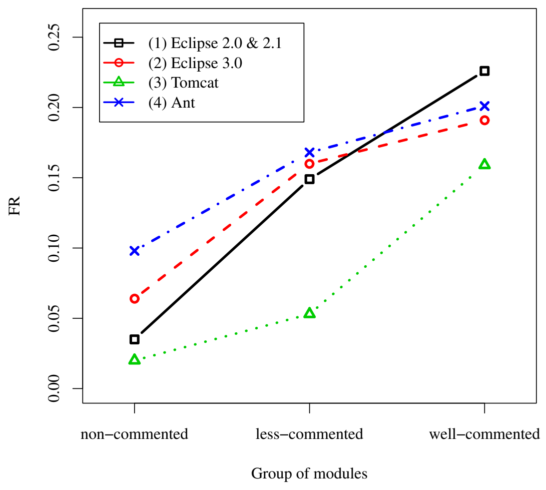

Figure 2. FR values in three groups: non-commented modules, less-commented ones and well-commented ones.

The Cochran-Armitage test is a test for linear trend in proportions, and we could confirm that there is a kind of increasing trend in FR of each data set.

Since there may be great variability among the above increasing tendencies, we also compared FR values for each pair of groups, (*non*, *less*), (*non*, *well*) and (*less*, *well*). The results of statistical tests (χ^2 test[13]) are shown in Table IV; The table presents only p values for each pair, for lack of space. For instance, the table shows the p value was 0.0062 in the test of the difference between FR value in well-commented modules and FR value in non-commented modules of data set Ant.

From Table IV, the differences between FR values in the pair (*non*, *well*) are all statistically-significant (the level of significant of 0.05). Thus, well-commented modules are

[13]We performed the test by using the function `pairwise.prop.test` of R.

Table IV
RESULTS OF STATISTICAL TESTS (p VALUES) FOR EACH PAIR OF MODULE GROUPS.

data set		non	less
Eclipse 2.0 & 2.1	less	$< 0.001^{***}$	—
	well	$< 0.001^{***}$	$< 0.001^{***}$
Eclipse 3.0	less	$< 0.001^{***}$	—
	well	$< 0.001^{***}$	0.029^{*}
Tomcat	less	0.105	—
	well	$< 0.001^{***}$	0.016^{*}
Ant	less	0.0696	—
	well	0.0062^{**}	0.5394

($***$: $p < 0.001$; $**$: $0.001 \leq p < 0.01$; $*$: $0.01 \leq p < 0.05$)

Table V
SOME THRESHOLDS OF WELL-COMMENTED MODULES IN TERMS OF LCM.

	LOC			
data set	100	200	300	400
Eclipse 2.0 & 2.1	5.4	9.6	13.4	17.1
Eclipse 3.0	5.4	10.0	14.4	18.6
Tomcat	5.0	8.5	11.6	14.5
Ant	4.4	6.9	8.9	10.7

more likely to be faulty than non-commented modules. However, the differences in some pairs of (*non*, *less*) or (*less*, *well*) are not significant for data sets Tomcat and Ant.

D. Discussion

As shown in Table III, the FR in well-commented modules is about 6.5 times higher than FR in non-commented ones of Eclipse 2.0 & 2.1. Similarly, FR values in well-commented modules are about 3 times, 8 times and 2.1 times higher than FR values in non-commented modules of data sets Eclipse 3.0, Tomcat and Ant, respectively.

Table V shows some thresholds of LCM over LOC for extracting the well-commented modules, calculated by Eqs.(3), (4), (5) and (6), respectively[14]. As will be noted from Table V, the thresholds of well-commented modules are about 5 to 19 LCM. Thus, a module in which even some few lines of comments are written may be a well-commented module in reality, and it is about 2 to 8 times more likely to be faulty than non-commented modules.

Although the overall trends of FR values have increasing tendencies (see Fig.2), the following three pairs did not show statistically-significant differences: (*non*, *less*) in Tomcat and Ant, and (*less*, *well*) in Ant; their p values are 0.105, 0.0696 and 0.5394, respectively. Therefore, such increasing tendencies in Tomcat and Ant may be weaker than Eclipse. Since the number of modules in Tomcat and Ant are fairly less than the others, such limited numbers might affect the results of our statistical tests. Alternatively, the above difference in results might be due to the difference in development group policy—Eclipse foundation vs. Apache software foundation. In order to confirm the above increasing tendencies more accurately, we have to collect more samples. That will be our important future work.

The above results seem to be consistent with the concern about comments as a "deodorant" for code smell [4]. Well-written comments may be signs of that the developer has a lack of confidence about the code clearness. However, we still do not know enough about why well-commented modules are more likely to be faulty. While the above results would be useful empirical data showing such impact of comments on fault-proneness, they are still not clear evidence to connect well-written comments to potential faults in modules. Since well-written comments do not cause any faults directly, there must be another factor to connect them. To discuss details, we will have to collect additional metric data evaluating the code complexity and analyze the relationship of LCM with the code complexity. Such further analysis and discussion are our significant future work.

[14]We calculated $\widehat{\mathrm{LCM}}$ by the above equations for some representative values of LOC. LOC values of the all modules analyzed above are less than 400 since the outliers were filtered out beforehand.

E. Threats to Validity

Here, we discuss some threats to the validity of our work.

In order to make a criterion for deciding well-commented modules, we proposed to use the regression line on the scatter diagram between LCM and LOC. Since our proposal is made under the assumption that there is a certain level of correlation (may be weak) between LCM and LOC from our experience, our discriminant method will be poor if they have a strong correlation: if there is a strong correlation (e.g. the correlation coefficient > 0.9), most of the modules will be plotted around the regression line exemplifying a small difference among the modules in their amounts of comments. In such case, categorization will be worthless.

This work used only the comments written inside the method bodies, so both "Javadoc" and "comment out" were not analyzed in our empirical work. While we statistically showed that the well-commented modules were fault-prone, the well-written Javadoc and/or many comment outs may affect the empirical results. We will have to perform further analysis using their combination as well.

The studied software are only three open source software. Thus, although they all were popular large- or middle-scale software, our results may not be generalized to other open source software and/or commercial software. Moreover, the difference in the development groups, such as Eclipse foundation and Apache software foundation, might affect our results, since different group may have a different coding policy. Differences in programming language may also affect our results, because all of the studied software were written in Java.

V. Related Work

Many studies analyzing source programs to discuss their fault-proneness have not focused on the comments written in the programs, since comments cannot be the direct cause of faults. However, well-written comments might be clues as to the potential faults because complicated code fragments may require some comments for explaining their contents. Fowler [4] called such aspect as a sign of potential fault as "code smell" to be refactored. To be more precise, comments were considered to be "air freshener (deodorant)" beside a smelled code. While Fowler proposed such aspect of comments qualitatively, this paper analyzed such impacts of comments on the fault-proneness, quantitatively and empirically.

This empirical work is an improved version of Aman's work [5]. The previous work performed a similar empirical analysis with the ratio of LCM divided by LOC. However, such ratio has an increasing tendency as LOC gets higher in the actual data, so the previous work had an issue with the criterion for deciding well-commented modules automatically. Then, this paper proposed to use the regression model considering the distribution of LCM over LOC. Moreover, the empirical data is prepared by taking account of the overlaps among different versions, which was not taken into account in the previous work.

Buse *et al.* [1] and Tenny [10] have empirically studied the impacts of comments on the program readability. Both of their empirical results showed that comments help to improve the readability, but did not discussed the relationships with the fault-proneness. While their basic viewpoints about comments are close to our study, there is a fundamental difference in the focus of empirical work.

Fluri *et al.* [11] have analyzed comments from another viewpoint. They focused on co-evolution between source programs and their comments. Although their study reported some interesting findings about writing and growing comments, the relationship with fault-proneness was not discussed. While Aman and Okazaki [12] also focused on the relationship between the amount of comments and the stability of the programs during their evolution, they did not evaluate the association with the fault-proneness.

The mining software repositories (MSR) studies may be related to this work. They reported that the module, in which many bug fixes were recently occurred, would be fault-prone (e.g. Kim *et al.* [13] and bugspots[15]). Since some bug fixes may cause the developers to add some comments, a further investigation from such a point of view will be one of our important future work.

VI. Conclusion

In general, comments written in modules assist in improving the readability of those modules. However, well-commented modules may be hard to understand without their well-written comments, so they might end up being complex and a nest for potential faults. Thus, this paper focused on well-commented modules and their fault-proneness.

First, this paper introduced the metric, lines of comments (LCM), to quantify the amount of comments. Then, the paper proposed a regression model-based method to automatically pick out "well-commented" modules by considering the distribution of LCM over the lines of code (LOC). After that, an empirical analysis with three major open source software was performed. The results showed that the well-commented modules have about 2 to 8 times greater risks of being faulty than non-commented ones. Hence, well-commented modules are more likely to be fault-prone , and they should be reviewed preferentially.

This paper does not claim that the comments reduce the code quality. In other words, we do not deny the existence of comments, and do not hope developers to omit to write comments, since comments do not cause any faults. Comments are harmless. However, well-written comments can be signs of potential faults because problematic code may need more explanatory comments: if a developer faces a code fragment to which he/she wants to write some comments, the developer should be more careful with the code fragment and ask himself/herself, "Is there any room for improvement in the code?". Our empirical study would be an empirical basis to promote such code review by the developers themselves during their coding activities. To automatically find well-commented code fragments and recommend developers to review such code may be a useful application of our results.

While our empirical work statistically showed that well-commented modules are more likely to be faulty, we still do not know enough about why they are fault-prone. Well-written comments may be signs of that the developer has a lack of confidence about the code clearness. However, we do not have sufficient clue to connect the amount of comments to the fault-proneness. Further work to figure out the reason is our significant future work. We will have to perform further analysis with other metrics related to code complexity. Such analysis might find a way to understand the impact of comments on fault-proneness. The impact analysis of Javadoc and comment out will also be our future work.

Moreover, a construction of model for predicting fault-prone modules using LCM is also our future work. Since the accuracy of faulty module prediction based only on LCM is low (< 0.23; see Table III), we will have to combine LCM with other metrics in order to produce an actionable model.

Acknowledgment

The author would like to thank the anonymous reviews for their helpful comments on the earlier version of this paper. This work was supported by the Grant-in-Aid for Young Scientists (B) 22700035, Japan Society for the Promotion of Science.

[15]https://github.com/igrigorik/bugspots .

References

[1] R. P. Buse and W. R. Weimer, “A metric for software readability,” in *Proc. 2008 Int’l Symp. Softw. Testing and Analysis*, 2008, pp. 121–130.

[2] J. C. Munson, *Software Engineering Measurement*. Boca Raton, Florida: Auerbach Publications, 2003.

[3] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Longman, 1999.

[5] H. Aman, “Quantitative analysis of relationships among comment description, comment out and fault-proneness in open source software,” *IPSJ Journal*, vol. 53, no. 2, pp. 612–621, Feb. 2012, in Japanese.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Third Edition*. Upper Saddle River, NJ: Addison-Wesley Professional, 2005.

[7] G. J. Upton and I. Cook, *A Dictionary of Statistics, Second Edition*. Oxford University Press, 2008.

[8] G. Boetticher, T. Menzies, and T. Ostrand, “Promise repository of empirical software engineering data http://promisedata.org/repository,” West Virginia University, Department of Computer Science, 2007.

[9] A. Agresti, *Categorical Data Analysis, second edition*. Wiley, 2002.

[10] T. Tenny, “Program readability: Procedures versus comments,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.

[11] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, “Analyzing the co-evolution of comments and source code,” *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[12] H. Aman and H. Okazaki, “Impact of comment statement on code stability in open source development,” in *Knowledge-Based Software Engineering*, M. Virvou and T. Nakamura, Eds. Amsterdam: IOS Press, 2008, pp. 415–419.

[13] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting faults from cached history,” in *Proc. 29th Int’l Conf. Softw. Eng.*, 2007, pp. 489–498.