IEICE TRANS. INF. & SYST., VOL.E98–D, NO.12 DECEMBER 2015

PAPER

Lines of Comments as a Noteworthy Metric for Analyzing Fault-Proneness in Methods*

Hirohisa AMAN[†a)], ***Member***, **Sousuke AMASAKI**[††], **Takashi SASAKI**[†], ***Nonmembers***, ***and*** **Minoru KAWAHARA**[†], ***Member***

SUMMARY This paper focuses on the power of comments to predict fault-prone programs. In general, comments along with executable statements enhance the understandability of programs. However, comments may also be used to mask the lack of readability in the program, therefore well-written comments are referred to as "deodorant to mask code smells" in the field of code refactoring. This paper conducts an empirical analysis to examine whether Lines of Comments (LCM) written inside a method's body is a noteworthy metric for analyzing fault-proneness in Java methods. The empirical results show the following two findings: (1) more-commented methods (the methods having more comments than the amount estimated by size and complexity of the methods) are about 1.6 – 2.8 times more likely to be faulty than the others, and (2) LCM can be a useful factor in fault-prone method prediction models along with the method size and the method complexity.

key words: *product metrics, fault-prone method prediction, comments, regression model*

1. Introduction

The successful development of high-quality software systems requires well-organized management of development activities [1], [2]. "Programming" is one of the most fundamental activities to produce high-quality systems. However, the programming activity is also easily affected by the programmer's mind and experience, i.e., human factors, and the program could vary considerably in quality—different programmers may write different programs for the same specification. While it is natural that there would be some variations made by the differences among programmers, there may also be human errors behind their variations. Thus, it is important to perform quality management during the programming activity in order to control the quality of programs being developed.

To effectively perform quality management in the programming activity, there have been many studies about code metrics [3]–[5]. Size metrics and complexity metrics are known as especially useful metrics for assessing the quality of the programs and for detecting poor-quality programs. This is because programs that are too large and/or too complex are often problematic, i.e., they are likely to have more potential faults or to be costly in their maintenance phases [6]–[11]. Such problematic programs should be carefully reviewed and tested as early as possible.

While the size and the complexity are useful criteria in assessing the program's quality, there is also another aspect that is related to the human factors described above: a program usually consists of not only executable code but also "comments" written by the programmer. The comments are important elements in that they express the programmer's mind or purpose in regard to their program. In general, comments are well known as useful artifacts for enhancing the program comprehensibility, and they may be worthy of attention in quality management. There are some studies showing that comments are helpful in understanding a program [12], [13]. However, comments might be added to mask the lack of readability in problematic programs; some programmers want to insert expletive comments into parts which seem to be hard to understand. Kernighan et al. [14] recommended rewriting such complex programs rather than adding careful comments onto those parts. Fowler [15] said well-written comments are related to "code smells" to be refactored. While comments themselves are helpful in understanding the program, they can play roles as "deodorant" masking a code smell. We consider these pros and cons of well-written comments to be challenging topics of research.

In this paper, we focus on comments written in a software module: we will consider a method in a class to be a module. In the past, comments have not been major targets of code metric studies because no comment can affect the software's functionality, performance or structure. However, we consider that comments can also be notable artifacts for assessing the program quality. The key contribution of this paper is to show that Lines of Comments (LCM) can be a useful metric for analyzing fault-proneness in modules.

The remainder of this paper is organized as follows: Sect. 2 describes the comments of interest in this paper and our research motivation. Section 3 presents our empirical analysis for four OSS products, and discusses the results. Section 4 gives brief descriptions of related work. Finally, Sect. 5 describes our conclusions and future work.

Manuscript received March 26, 2015.
Manuscript revised July 30, 2015.
Manuscript publicized September 4, 2015.

[†]The authors are with the Center for Information Technology, Ehime University, Matsuyama-shi, 790–8577 Japan.
[††]The author is with the Faculty of Computer Science and Systems Engineering, Okayama Prefectural University, Soja-shi, 719–1197 Japan.

*The earlier versions of this paper were partly presented at the 4th International Workshop on Empirical Software Engineering in Practice (IWESEP 2012) [16], the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM2014) [20] and the 2nd International Workshop on Quantitative Approaches to Software Quality (QuASoQ2014) [21].

a) E-mail: aman@ehime-u.ac.jp

DOI: 10.1587/transinf.2015EDP7107

```
 1:  public static void processFiles(final PMDConfiguration configuration, final RuleSetFactory ruleSetFactory,
 2:          final List<DataSource> files, final RuleContext ctx, final List<Renderer> renderers) {
 3:      sortFiles(configuration, files);
 4:
 5:      /*
 6:       * Check if multithreaded support is available.  ExecutorService can also
 7:       * be disabled if threadCount is not positive, e.g.  using the
 8:       * "-threads 0" command line option.
 9:       */
10:      if (SystemUtils.MT_SUPPORTED && configuration.getThreads() > 0) {
11:          new MultiThreadProcessor(configuration).processFiles(ruleSetFactory, files, ctx, renderers);
12:      } else {
13:          new MonoThreadProcessor(configuration).processFiles(ruleSetFactory, files, ctx, renderers);
14:      }
15:  }
```

Fig. 1 An example of a method written in Java.

2. Lines of Comments, Research Motivation and Aim

This section defines our key metric for comments—Lines of Comments (LCM)—, and provides clarification of our research motivation and aim along with a brief description of our previous work.

2.1 Comments in Methods

We focus on the part describing the sequence of executions, i.e., the functions/methods. Comments written inside a function's/method's body, except for commented-out code[†], are often programmer's notes that explain the content. Some programmers want to add in-depth explanations as comments when their code fragments look complicated.

In general, comments help to enhance the readability of programs [17]. However, well-written comments may be related with "code smell" in the field of code refactoring [15]. While comments themselves are helpful in understanding the program, they are sometimes used as "deodorant to mask code smells."

Now we define the following metric of comments, Lines of Comments (LCM):

LCM = the number of lines of comments in a method body, except for commented-out code. □

This metric is to quantify the amount of comments in a method. Figure 1 shows an example of a method written in Java, which is included in a source file of PMD [18]. The method has a block of comments from the 5th line to the 9th line, so LCM= 5.

Note that the LCM does not count comments written "outside" a method's body. Since comments written outside a method's body are often descriptions of how to use the method like Javadoc, those comments may not work as "deodorant" mentioned above. While there might also be important comments explaining a complicated code, it is not easy to automatically find such meaningful comments. Moreover, we need to associate those comments with the target method in order to analyze the relationship between the lines of comments and fault-proneness in methods (see Sect. 3). Therefore, we decided to ignore comments written outside a method's body in our measurement of the LCM. A further analysis considering such comments is one of our significant future work.

2.2 Research Motivation and Aim

There have been some empirical studies showing relationships between comments and fault-proneness [16], [19]–[21]. Aman [16], [19] conducted some statistical analyses at source file level, and reported that commented programs are more likely to be faulty than the others. While those results are interesting, the unit of analysis is a bit coarse; a finer-grained analysis is better to discuss its application to effective code reviews [22]. There are also studies to analyze relationships between comments and fault-proneness at method level rather than source file level [20], [21]. However, the literature [20], [21] did not consider the case in which LCM is used along with both the program size (Lines of Code: LOC) and the program complexity (Cyclomatic Complexity: CC) [23]; LOC and CC are well-known promising metrics for predicting fault-prone programs[††]. Thus, we will analyze the power of LCM at method level

[†]It is hard to perfectly discriminate the commented-out code from other comments. A perfect discrimination requires thorough follow-up investigations for each comment, for each programmer. However, it would be impossible in practice, so we use Aman's algorithm [16] instead. While Aman's algorithm is a simple algorithm, most of the comments (98.9%) can be successfully discriminated.

[††]The LOC value of a method is measured by counting the lines of executable code in the method's body, except for the comment lines and the blank lines. The CC value of a method is calculated as the number of branches plus one in the method's body. For example, in the case of the method shown in Fig. 1, LOC = 6 and CC= 2 because the method has six lines of executable code in line 3 and 10–14, and it has one branch statement (`if` statement) in line 10.

when used together with LOC and CC, with considering the following two cases.

Case 1: we have no data of faults

When we analyze new projects or projects which have not tracked fault data or file changes, we cannot use any fault data to build a model for predicting fault-prone methods. In such cases, we would try to extract methods whose metric values were abnormal in terms of their size (LOC), complexity (CC) and the amount of comments (LCM), in order to predict fault-prone methods. If LCM helps to detect faulty methods along with LOC and CC, we can say that LCM is one of generally useful metrics for analyzing fault-proneness in methods.

Case 2: we have data of faults

When we know which methods were faulty in the older version of a product, we can apply a supervised learning to predict fault-prone methods in the product. In this paper, we will use the logistic regression model, which is one of the most popular supervised learning models in fault-prone module prediction studies. The fitness of a logistic regression model to the actual data can be quantitatively evaluated by Akaike Information Criterion (AIC) [24]. Thus, we can easily see which set of explanatory variables (metrics) is the best for a logistic regression model, among all possible combinations of metrics: since LOC and CC have been widely known as good predictors of fault-prone programs, we evaluate the logistic regression model using LCM together with LOC and CC, in order to examine whether LCM is a noteworthy metric for analyzing fault-proneness in methods. This ease of analysis is the main reason why we use the logistic regression model. While a further examination using other models such as the random forests [10] would be useful to discuss the practical power of LCM for predicting fault-prone methods, such a further empirical study is our significant future work.

3. Empirical Analysis

This section conducts an empirical analysis of OSS products, and discusses the usefulness of LCM using the results.

3.1 Research Questions

We start with the clarification of our research questions in this empirical analysis.

RQ1: Can LCM contribute to predictively discriminate fault-prone methods from the others?

RQ2: Can LCM contribute to enhance the fitness of the logistic regression model for detecting faulty methods? □

RQ1 and RQ2 correspond to case 1 and 2, described in Sect. 2.2, respectively. If both of the answers to RQ1 and RQ2 are "Yes," we can conclude that LCM is an important metric along with LOC and CC. Therefore, we will have to pay attention to not only the size and the complexity of the method but also the comment lines in order to build useful mathematical models for analyzing fault-proneness in methods.

Table 1 OSS projects analyzed in the empirical work.

Project	Data collection period	Number of methods
Eclipse Checkstyle Plugin	2003/05/05 – 2014/02/15	1,051
Hibernate ORM	2007/06/29 – 2014/02/13	18,483
PMD	2002/06/21 – 2014/05/02	3,970
SQuirreL SQL Client	2001/06/01 – 2014/05/02	6,116
total	—	29,620

3.2 Experimental Objects

Our research objects are the methods developed in four OSS projects: Eclipse Checkstyle Plugin [25], Hibernate ORM [26], PMD [18] and SQuirreL SQL Client [27]. Table 1 shows the data collection periods from the projects and the numbers of methods developed in the projects. We selected these projects since they satisfy all of the following three requirements:

1) their programs are written in Java,

2) their source files are maintained with Git, and

3) their products are popular OSS products.

Requirement 1) is from our data collection tools [28]: JavaMethodExtractor, CommentCounter, LOCCounter, and CyclomaticNumberCounter. Requirement 2) is for the ease of access to data: we can easily make a clone of Git repository in our local hard disk, so our data collection can be quickly performed with a low cost. While Subversion is also a popular repository system, it uses a centralized revision control model, so we have to access the server whenever we check the repository. Requirement 3) is for enhancing the generality of our empirical results. We selected popular OSS products which are differently sized and are from different domains; all of them are ranked in the top 20 Java products at SourceForge.net[†] which is one of the most popular OSS community sites. Since it is easy to sort OSS products in decreasing order of "popularity" at SourceForge.net, we sorted Java products by using condition "Sort By: Most Popular" and selected ones whose source files are managed with Git. We could have selected popular products at GitHub[††] by searching products with their sort option "most starts" or "most forks" as well.

[†]http://sourceforge.net/
[††]https://github.com/

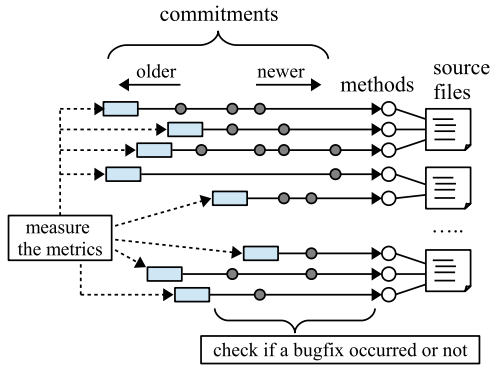


Fig. 2 Data collection scheme.

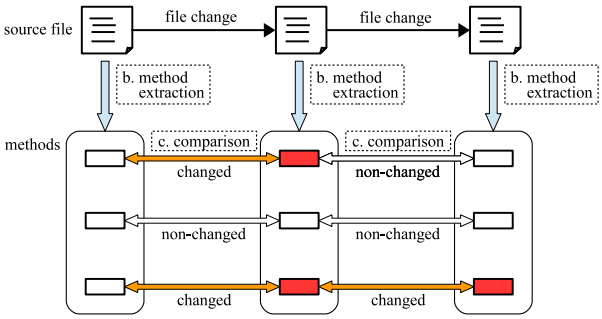


Fig. 3 Tracing the history of method changes.

3.3 Analysis Procedure

We perform our empirical analysis for the above research questions in the following procedure.

1. Data Collection:

   For each OSS product, obtain the latest version of the source files from the repository, and make a list of the methods which appear in the files, except for all abstract methods. Then, get the "initial" version of the methods by tracing their histories of changes, and measure the LCM, LOC and CC values in those initial version of methods. Moreover, for each method, examine whether the method is faulty or not by checking all changes (their evolutions) in which the method was involved (see Fig. 2): we decided that a change was for fixing a bug if the corresponding commitment log included one of the words related to bug fixing—"bug," "fix" and "defect."

   Notice that we require a suite of data processing in order to trace the history of method changes. Since the repository manages the source programs at the units of source files, their commitment logs provide the history of file-level changes, i.e., not method-level changes. To determine if a method was changed or not, we need to compare the method bodies before and after the source file change. Specifically, we performed the following three steps for each source file change to collect the history of method changes (see Fig. 3).

   a. Make copies of the source file before and after the change, respectively.

   b. Extract method's bodies from the source files through a syntax analysis—the method extraction can be performed using our data collection tool `JavaMethodExtractor`[†].

   c. Compare the method bodies having the same signature before and after the change, using `diff` utility[††]. If there is a difference between the method bodies, the method is considered to be changed at that time; otherwise, the method is regarded as a non-changed one and not included in the history of the method changes.

2. Correlation Analysis:

   Calculate the correlation coefficient for each pair of metrics, and examine the quantitative independence of the metrics. If LCM has a strong correlation with LOC or CC, LCM is considered to be a worthless metric because LCM cannot provide useful information which is independent of LOC and CC. We will check the correlations in their log-transformed forms as well, because some metric-value distributions may be right-skewed.

3. Fault-prone Method Classification (for RQ1):

   Build a linear regression model whose objective variable is LCM or log(LCM+1), and the pair of the explanatory variables is { LOC, CC }, { LOC, log(CC) }, { log(LOC), CC } or { log(LOC), log(CC) } as:

$$
\begin{aligned}
\widehat{\mathrm{LCM}} &= a\,\mathrm{LOC} + b\,\mathrm{CC} + c\,, \\
\widehat{\mathrm{LCM}} &= a\,\mathrm{LOC} + b\,\log(\mathrm{CC}) + c\,, \\
\widehat{\mathrm{LCM}} &= a\,\log(\mathrm{LOC}) + b\,\mathrm{CC} + c\,, \\
\widehat{\mathrm{LCM}} &= a\,\log(\mathrm{LOC}) + b\,\log(\mathrm{CC}) + c\,, \\
\log(\widehat{\mathrm{LCM}} + 1) &= a\,\mathrm{LOC} + b\,\mathrm{CC} + c\,, \\
&\cdots \\
\log(\widehat{\mathrm{LCM}} + 1) &= a\,\log(\mathrm{LOC}) + b\,\log(\mathrm{CC}) + c\,,
\end{aligned}
$$

   where a, b and c are constant numbers. We will use the model best fitted to actual data, in the following analysis.

   The linear regression model can be a discriminator to classify the methods into the set of "more-commented methods" and the set of "others" (see Fig. 4). Then, by comparing the ratio of faulty methods between these two sets, examine the power of LCM to predict fault-prone methods in an unsupervised manner.

4. Logistic Regression Analysis (for RQ2):

   Build logistic regression models with all possible combinations of explanatory variables (see Table 2 and

[†]http://se.cite.ehime-u.ac.jp/tool/
[††]We ignored all differences of white space characters.

Table 2 Explanatory variable(s) used in logistic regression analysis.

Model no.	explanatory variable						
	log(LCM+1)	LCM> 0	LCM	log(LOC)	LOC	log(CC)	CC
1							√
2						√	
3					√		
4					√		√
5					√	√	
6				√			
7				√			√
8				√		√	
9			√				
10			√				√
11			√			√	
12			√		√		
13			√		√		√
14			√		√	√	
15			√	√			
16			√	√			√
17			√	√		√	
18		√					
19		√					√
20		√				√	
21		√			√		
22		√			√		√
23		√			√	√	
24		√		√			
25		√		√			√
26		√		√		√	
27	√						
28	√						√
29	√					√	
30	√				√		
31	√				√		√
32	√				√	√	
33	√			√			
34	√			√			√
35	√			√		√	

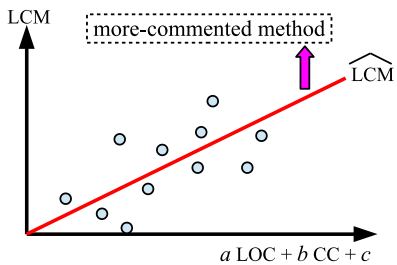
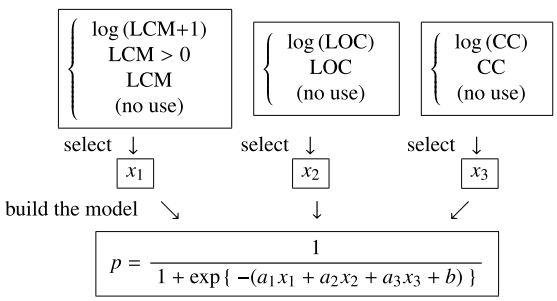


Fig. 4 Regression-based decision of more-commented methods.



(*p*: probability of faulty method; a_1, a_2, a_3 and *b*: constant numbers)

Fig. 5 Explanatory variable selection for a logistic model.

Fig. 5), where the objective variable is whether the method is faulty (1) or not (0). In Table 2, each row corresponds to each model using the marked variable(s) as their explanatory variable(s), where

- log(LCM+1) is the log-transformed LCM instead of log(LCM) since there may be non-commented methods—LCM= 0;
- LCM > 0 is a dummy variable as to whether the LCM value is non-zero or not;
- "LCM," "log(LCM+1)" and "LCM > 0" are exclusively selected;
- log(LOC) is the log-transformed LOC;
- "LOC" and "log(LOC)" are exclusively selected;
- log(CC) is the log-transformed CC;
- "CC" and "log(CC)" are exclusively selected.

Now we have to consider an imbalance between the number of "faulty methods" and the number of "non-faulty methods": the number of faulty methods tends

Table 3 Pearson's correlation coefficients among LCM, LOC, CC and their log-transformed ones in Eclipse Checkstyle Plugin.

r	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.622	0.571	0.484	0.456
LCM	0.514	0.537	0.384	0.380
log(LOC)	—	—	0.813	0.734
LOC	—	—	0.689	0.762

Table 4 Pearson's correlation coefficients among LCM, LOC, CC and their log-transformed ones in Hibernate ORM.

r	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.534	0.560	0.511	0.504
LCM	0.444	0.589	0.429	0.502
log(LOC)	—	—	0.856	0.703
LOC	—	—	0.699	0.835

Table 5 Pearson's correlation coefficients among LCM, LOC, CC and their log-transformed ones in PMD.

r	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.535	0.635	0.526	0.545
LCM	0.394	0.623	0.390	0.494
log(LOC)	—	—	0.880	0.688
LOC	—	—	0.774	0.877

Table 6 Pearson's correlation coefficients among LCM, LOC, CC and their log-transformed ones in SQuirreL SQL Client.

r	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.416	0.434	0.336	0.325
LCM	0.334	0.473	0.301	0.436
log(LOC)	—	—	0.744	0.485
LOC	—	—	0.647	0.748

to be significantly lower than the number of non-faulty methods in many cases. Such an imbalance may lead to a poor logistic regression model which classifies almost all of the methods as non-faulty ones. To avoid the influence of such an imbalance on our model building, we will use an oversampling algorithm, Synthetic Minority Over-sampling Technique (SMOTE) [29].

Then, examine if LCM is a useful explanatory variable to detect faulty methods through this analysis.

3.4 Result and Discussion: Correlation Analysis

Tables 3 – 6 show Pearson's correlation coefficients for each pair of metrics, for four OSS projects; for example, the correlation coefficient between LCM and log(LOC) in Eclipse Checkstyle Plugin is $r(\text{LCM}, \log(\text{LOC})) = 0.514$ (see Table 3). The coefficients which appear in the upper half of the tables show the correlations of LCM with LOC or CC, including their log-transformed forms. The coefficients in the lower half of the tables signify the correlations between LOC and CC, including their log-transformed forms.

For each project, all correlation coefficients in the upper half are less than all the ones in the lower half. That is to say, LCM has weaker correlation with LOC and CC than the correlations between LOC and CC: $r(\text{LCM}, \text{LOC}) < r(\text{LOC}, \text{CC})$ and $r(\text{LCM}, \text{CC}) < r(\text{LOC}, \text{CC})$. Similar relations hold for log-transformed ones as well.

Table 7 Spearman's rank correlation coefficients among LCM, LOC, CC and their log-transformed ones in Eclipse Checkstyle Plugin.

ρ	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.613	0.613	0.485	0.485
LCM	0.613	0.613	0.485	0.485
log(LOC)	—	—	0.805	0.805
LOC	—	—	0.805	0.805

Table 8 Spearman's rank correlation coefficients among LCM, LOC, CC and their log-transformed ones in Hibernate ORM.

ρ	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.403	0.403	0.422	0.422
LCM	0.403	0.403	0.422	0.422
log(LOC)	—	—	0.768	0.768
LOC	—	—	0.768	0.768

Table 9 Spearman's rank correlation coefficients among LCM, LOC, CC and their log-transformed ones in PMD.

ρ	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.430	0.430	0.445	0.445
LCM	0.430	0.430	0.445	0.445
log(LOC)	—	—	0.828	0.828
LOC	—	—	0.828	0.828

Table 10 Spearman's rank correlation coefficients among LCM, LOC, CC and their log-transformed ones in SQuirreL SQL Client.

ρ	log(LOC)	LOC	log(CC)	CC
log(LCM+1)	0.353	0.353	0.264	0.264
LCM	0.353	0.353	0.264	0.264
log(LOC)	—	—	0.694	0.694
LOC	—	—	0.694	0.694

We also checked Spearman's rank correlation coefficients in view of a skewness of a metric-value distribution. Tables 7 – 10 show Spearman's rank correlation coefficients for each pair of metrics. Since the correlation coefficients are computed by using the ranks of the metric values, the log-transformed metrics have the same ranks with non-transformed ones, and the coefficients with the log-transformed metrics result in the same coefficients with the non-transformed ones. The results of Spearman's rank correlation coefficients show a trend that is similar to the results of Pearson's correlation coefficients: all coefficients in the upper half of Tables 7 – 10 are less than all the ones in the lower half, i.e., the correlations of LCM with LOC and CC are weaker than the one between LOC and CC.

In both Pearson's correlation and Spearman's correlation, LCM has moderate positive correlations with LOC and CC, but the correlations are not strong [30]. Thus, we consider that LCM is not dominated by LOC or CC, so LCM can provide information which is independent of LOC and CC. We can use LCM as an explanatory variable along with LOC and CC in the logistic regression analysis for RQ2.

Table 11 Best fitted regression models.

Project	model
Eclipse Checkstyle Plugin	$\widehat{\text{LCM}} = \exp\{\, 0.471\log(\text{LOC}) - 0.0617\log(\text{CC}) - 0.573\,\} - 1$
Hibernate ORM	$\widehat{\text{LCM}} = 0.0684\text{LOC} + 0.0994\log(\text{CC}) - 0.240$
PMD	$\widehat{\text{LCM}} = 0.198\text{LOC} - 0.950\log(\text{CC}) - 0.699$
SquirreL SQL Client	$\widehat{\text{LCM}} = 0.0364\text{LOC} + 0.109\text{CC} - 0.245$

Table 12 Ratios of faulty methods.

Project	(a) more-commented ($\text{LCM} > \widehat{\text{LCM}}$)	(b) others ($\text{LCM} \le \widehat{\text{LCM}}$)	(a)/(b)	p value
Eclipse Checkstyle Plugin	$\frac{36}{267}$ (0.135)	$\frac{44}{784}$ (0.0561)	2.41	5.01×10^{-5}
Hibernate ORM	$\frac{37}{1771}$ (0.0209)	$\frac{125}{16712}$ (0.00748)	2.79	1.86×10^{-8}
PMD	$\frac{21}{288}$ (0.0729)	$\frac{128}{3682}$ (0.0348)	2.09	0.00181
SQuirreL SQL Client	$\frac{45}{577}$ (0.0780)	$\frac{269}{55391}$ (0.0486)	1.60	0.00319

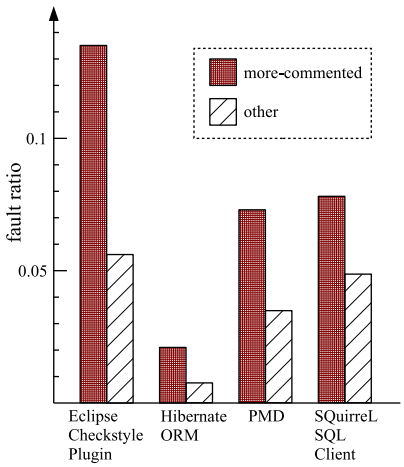


Fig. 6 Fault ratios in each project.

Table 13 Explanatory variables and AIC values of best logistic regression models.

Project	model no.	Explanatory variables	AIC
Eclipse Checkstyle Plugin	25*	(LCM> 0), log(LOC), CC	2423.372
Hibernate ORM	26*	(LCM> 0), log(LOC), log(CC)	48720.310
PMD	14*	LCM, LOC, log(CC)	9424.186
SQuirreL SQL Client	25*	(LCM> 0), log(LOC), CC	15093.100

3.5 Result and Discussion: RQ1

As we saw in Tables 3 – 6, there are weak or moderate positive correlations of LCM with LOC and CC. By utilizing these weak or moderate correlations, we can establish a criterion to decide whether a method has more comments for its size (LOC) and complexity (CC) or not. We built linear regression models with all possible combinations of explanatory variables and an objective variable: the explanatory variables are LOC, CC, log(LOC) or log(CC), and the objective variable is LCM or log(LCM+1). Then, we selected the best fitted model with the maximum adjusted R^2 as shown in Table 11. $\widehat{\text{LCM}}$ in Table 11 signifies an expected value of LCM by LOC and CC. That is to say, a method whose LCM is greater than $\widehat{\text{LCM}}$ can be regarded as a "more-commented method." By using the regression expressions in Table 11, we can classify all methods into two subsets—more-commented methods and the others.

Table 12 shows the ratios of faulty methods (fault ratio) by the method subsets. The table also presents the p values of χ^2 tests whose null hypothesis is "two ratios are the same," and the alternative hypothesis is "two ratios differ." For all the projects, all null hypotheses are rejected at the level of significance 0.01. That is to say, we can consider that the ratios of faulty methods significantly differ between the set of more-commented methods and the set of others. Indeed, for each OSS project, the fault ratio in more-commented methods is about 1.6 – 2.8 times higher than the fault ratio in the others (see Table 12 and Fig. 6). Therefore, the answer to RQ1 results in "Yes": LCM can play an important role to discriminate fault-prone methods from the others, along with LOC and CC. Note that this analysis is conducted in an unsupervised manner, i.e., the discriminant models are built by using LCM, LOC and CC only; they do not use the data of fault.

3.6 Result and Discussion: RQ2

Next, we show the results of the supervised learning by using the logistic regression model. Table 13 lists explanatory variables of the best logistic regression models. The quality of the model is evaluated by Akaike Information Criterion (AIC) [24] which considers the level of fitness to actual data and the complexity of the model[†]. In general, the lower the value of AIC means the better the logistic regression model is. Now we consider not only the model whose AIC value is the minimum (AIC_{min}) among 35 models, but also the ones such that $\Delta\text{AIC} = \text{AIC} - \text{AIC}_{min} < 2$, to be the best models, because two different models are considered to be at the approximately same level of quality when $\Delta\text{AIC} < 2$ [31].

In Table 13, the model numbers are asterisked if LCM

[†]To reduce the influences of randomness in SMOTE algorithm, we executed the model building 100 times and calculated the average of their AIC values. AIC values in Table 13 are the average ones.

Table 14 Partial regression coefficients of the best logistic regression models.

Project	Explanatory variables: Partial regression coefficients		
Eclipse Checkstyle Plugin	(LCM> 0) 1.50	log(LOC) 0.228	CC −0.252
Hibernate ORM	(LCM> 0) 0.829	log(LOC) 0.177	log(CC) 0.093
PMD	LCM −0.079	LOC 0.156	log(CC) 0.913
SQuirreL SQL Client	(LCM> 0) 0.170	log(LOC) 0.547	CC −0.030

appeared in their sets of explanatory variables. For all projects, LCM appeared in the best models. Therefore, the answer to RQ2 results in "Yes": LCM can be a useful explanatory variable to enhance the fitness of the logistic regression model for detecting faulty methods along with LOC and CC. The models are built to better fit actual data by using not only LOC and CC but also LCM, so they would be more useful for predicting fault-prone methods in later versions of the products. Notice that the way of LCM's contribution differs among OSS projects: Table 14 shows the averages of the partial regression coefficients of those models. For three out of four OSS projects (Eclipse Checkstyle Plugin, Hibernate ORM and SQuirreL SQL Client), the partial regression coefficients of LCM are positive. However, for the remaining project (PMD), the partial regression coefficient is negative. That is to say, it is not always true that comments "solely" play signs of faulty methods. While LCM is a worthwhile metric for discussing fault-proneness in methods, it is important to consider the "combination" of LCM with LOC and CC—the suite of results shown in Sect. 3.5 (see Tables 11 and 12) can be a case utilizing the combination of LCM with LOC and CC in order to detect fault-prone methods.

From the results for RQ1 and RQ2, we can consider LCM to be a noteworthy metric for analyzing fault-proneness in methods, in both an unsupervised manner and a supervised manner.

3.7 Threats to Validity

Our empirical analysis was limited to Java products since our data collection tools can work for only Java. In many cases, writing program elements (statements and conditions) are common among major object-oriented languages, so the difference in language might not invalidate the contribution of this paper.

Our fault data collection was based on keyword matching in commitment logs, so we might miss some of the true faulty methods. Since many studies analyzing code repositories have adopted such a keyword matching approach in the past [8], [32], [33], we followed the same manner to detect bug fixing commitments. It would be beneficial to conduct the same analysis with a different fault-data-collection mechanism, e.g., collaborating with a bug tracking system. Moreover, our data collection scheme could not take into account the file renaming event. Since we focused only on the events related to file creations and changes (modifications) in our data collection, we might miss some methods in our data sets. While there may not be many renaming events, it can be a concern about our analysis accuracy. It would be cleared by collecting data using Git's function for tracing file renaming events. `Historage` [34]—a fine-grained version control system for Java—would also be a great help in the data collection.

Our analysis was based on the number of comment lines, not on the content of comments. While "what the comments say" would be an important factor, we empirically showed that even "the amount of comments" has a significant relation with fault-proneness in this paper. However, we still have another concern about how to measure the amount of comments accurately. While one programmer writes a comment in one line, another programmer may write the same comment in two or more lines. That is to say, LCM can vary from person to person for the same comment description. Such a difference in writing comments do not seem to produce a large variance in LCM values, but might affect on our empirical results in this paper. There has been a similar concern about measuring a program size using LOC—a diversity in programming style may cause a variance in LOC values. Since both LOC and LCM are based on the simple measure of lines in source files, there are some concerns about accurate measurements. Preprocessing like a normalization of program's style may be needed to reduce such concerns. In this paper, we have used LCM as a metric of the amount of comments because it is simple and easy to measure. We will continue to study more accurate and useful ways to measure the amount of comments in the future.

4. Related Work

Tenny [12] and Woodfield et al. [13] conducted experiments about program comprehension, with students and experienced programmers as their subjects, respectively. In their experiments, the subjects evaluated the ease of understanding some variations of a program which were different in modular design and in commenting manner (with or without comments). Their experimental results showed that comments have a positive impact on program comprehension. However, they did not evaluate the relation of comments with fault-proneness.

Steidl et al. [35] proposed a framework to evaluate the quality of comments. They evaluated the coherence between header comments[†] and the name of the corresponding method. Their coherence evaluation presents whether the header comments provide useful information about the method. They discussed the comments written inside method bodies as well, and proposed to use short comments as indicators of parts to be refactored. Moreover, they described that the existence of long comments may infer a

[†]Header comments are ones followed by a method declaration.

lack of external documents, so adding many comments was not recommended. While they studied valuable relationships between comments and programs, their focus was on the quality of comments, but not on fault-proneness. Lawrie et al. [36] also studied the correspondence of comments and code by using a natural language processing technique, for an automated quality assessment. However, the literature [36] did not discuss the fault-prone program prediction.

Aman et al. [16], [19]–[21] conducted some empirical studies of relationships between the amount of comments and fault-proneness. While those results brought a new worth focusing on comments, the unit of analysis in [16], [19] was a bit rough grained—at the source file level. This paper makes a finer grained empirical study of comments at the method level. By analyzing finer data of comments in each method, we provide more detailed and useful findings about the relationships between comments and code quality. While the analyses conducted in [20], [21] were at the method level, they did not examine relationships with LOC and/or CC in detail. This paper analyzes the relationships among LCM, LOC and CC by using all possible combinations of explanatory variables (35 different models). Moreover, this paper classifies the methods into two sets—the more-commented ones and the others—by using LCM, LOC and CC, and compares the fault ratios between them.

The coding standards (code conventions) such as MISRA C [37] and Google Java Style [38] are also useful to control the program quality from the view point of the programming style. While coding standards are just guidelines for the programming style and do not assess the design and contents of the programs, there are some studies showing relationships between coding standard violations and program faults [39], [40]. While the point of view focusing on the relationship between the programming style and the fault-proneness is common to this work, the papers [39], [40] did not analyze the power of comments for detecting faults. The findings of this paper may contribute to an enhancement of a code convention about comments in a program.

5. Conclusion

This paper focused on comments written inside method bodies, and conducted an empirical analysis on whether Lines of Comments (LCM) can be a useful metric for analyzing fault-proneness in methods. The empirical work collected data of three metrics—LCM, Lines of Code (LOC) and Cyclomatic Complexity (CC)—of 29,620 methods from four major open source software projects, and performed statistical analyses from two different aspects: 1) predicting fault-prone methods by using only LOC, CC and LCM (an unsupervised classification), and 2) building a useful logistic regression model to detect faulty methods, by using actual fault data (a supervised learning).

Analysis 1) classified the methods into two subsets, the set of more-commented methods and the set of the others, then compared their fault ratios. The results showed that more-commented methods are about 1.6 – 2.8 times more likely to be faulty than the other methods.

Analysis 2) built logistic regression models with all possible combinations of explanatory variables (35 different combinations) to detect faulty methods. The results showed that LCM can be a worthwhile explanatory variable along with LOC and CC.

From these empirical results, we conclude that LCM can be a useful metric along with LOC and CC for analyzing fault-proneness in methods. Some programmers want to add more comments to their programs that seem to be hard to understand by others, so faulty methods may have more comments for their size and complexity. Therefore, comments may help to identify poor quality parts while comments themselves are helpful in enhancing the program understandability. Note that this paper does not say you should avoid writing comments in your programs; the primary message of this paper is to show that comments can be notable in the program quality management activities.

Our future work includes the following: (1) a further study considering the severity of faults, (2) a further analysis using various products from different domains and organizations (including commercial products), and written in other languages, e.g., C, C++ and C#, (3) a time series analysis of commenting activities, (4) a further analysis focusing on contents of comments using natural language processing techniques, (5) a further empirical study using models not only the logistic regression model but also other models such as the random forests, (6) a cost-effectiveness evaluation [34], [41] of fault-prone method prediction using LCM, toward practical use of LCM, and (7) a further study on usefulness of LCM using metrics other than LOC and CC, e.g., process metrics appeared in the mining software repository world.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 25330083.

References

[1] SQuBOK Project Team, “Guide to the software quality body of knowledge (SQuBOK).” http://www.juse.or.jp/software/pdf/squbok_eng_ver1.pdf, accessed Oct. 12, 2014.
[2] R.S. Pressman, Software Engineering: a practitioner’s appraoch, 6th ed., McGraw-Hill, Columbus, OH, 2005.
[3] N.E. Fenton and S.L. Pfleeger, Software Metrics, PWS Publishing, Boston, MA, 1997.
[4] J.C. Munson, Software Engineering Measurement, CRC Press, Boca Raton, Florida, 2003.
[5] S.H. Kan, Metrics and Models in Software Quality Engineering, Pearson Education, Upper Saddle River, NJ, 2003.
[6] T.M. Khoshgoftaar and N. Seliya, “Comparative assessment of software quality classification techniques: An empirical case study,” Empirical Softw. Eng., vol.9, no.3, pp.229–257, Sept. 2004.
[7] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” IEEE Trans. Softw. Eng., vol.31, no.10, pp.897–910, Oct. 2005.
[8] S. Kim, E.J. Whitehead, and Y. Zhang, “Classifying software changes: Clean or buggy?,” IEEE Trans. Softw. Eng., vol.34, no.2,

pp.181–196, March 2008.
[9] A.G. Koru, D. Zhang, K.E. Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," IEEE Trans. Softw. Eng., vol.35, no.2, pp.293–304, March 2009.
[10] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Trans. Softw. Eng., vol.34, no.4, pp.485–496, July 2008.
[11] Y. Liu, T.M. Khoshgoftaar, and N. Seliya, "Evolutionary optimization of software quality modeling with multiple repositories," IEEE Trans. Softw. Eng., vol.36, no.6, pp.852–864, Nov. 2010.
[12] T. Tenny, "Program readability: Procedures versus comments," IEEE Trans. Softw. Eng., vol.14, no.9, pp.1271–1279, Sept. 1988.
[13] S.N. Woodfield, H.E. Dunsmore, and V.Y. Shen, "The effect of modularization and comments on program comprehension," Proc. 5th Int'l Conf. Softw. Eng. (ICSE'81), pp.215–223, March 1981.
[14] B.W. Kernighan and R. Pike, The practice of programming, Addison-Wesley Longman, Boston, MA, 1999.
[15] M. Fowler, "Refactoring: Improving the Design of Existing Code," Extreme Programming and Agile Methods — XP/Agile Universe 2002, Lecture Notes in Computer Science, vol.2418, pp.256–256, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
[16] H. Aman, "An empirical analysis on fault-proneness of well-commented modules," Proc. 4th Int'l Workshop on Empirical Softw. Eng. in Practice (IWESEP 2012), pp.3–9, Oct. 2012.
[17] D. Boswell and T. Foucher, The Art of Readable Code, Oreilly, CA, 2011.
[18] PMD, "PMD." http://pmd.sourceforge.net/, accessed May 2, 2014.
[19] H. Aman, "An empirical analysis of the impact of comment statements on fault-proneness of small-size module," Proc. 19th Asia-Pacific Softw. Eng. Conf. (APSEC2012), pp.362–367, Dec. 2012.
[20] H. Aman, T. Sasaki, S. Amasaki, and M. Kawahara, "Empirical analysis of comments and fault-proneness in methods: can comments point to faulty methods?," Proc. 8th Int'l Symp. Empirical Softw. Eng. and Measurement (ESEM2014), p.63, Sept. 2014.
[21] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of fault-proneness in methods by focusing on their comment lines," Proc 21st Asia-Pacific Softw. Eng. Conf. (APSEC2014), vol.2, pp.51–56, Dec. 2014.
[22] K.E. Wiegers, Peer Reviews in Software: A Practical Guide, Addison-Wesly, Boston, 2002.
[23] T.J. McCabe, "A complexity measure," IEEE Trans. Softw. Eng., vol.SE-2, no.4, pp.308–320, Dec. 1976.
[24] H. Akaike, "A new look at the statistical model identification," IEEE Trans. Automatic Control, vol.19, no.6, pp.716–723, Dec. 1974.
[25] L. Ködderitzsch, "Eclipse Checkstyle Plugin." http://eclipse-cs.sourceforge.net, accessed Feb. 15, 2014.
[26] Hibernate, "Hibernate ORM – Hibernate ORM." http://hibernate.org/orm/, accessed Feb. 13, 2014.
[27] SQuirreL SQL Client, "SQuirreL SQL Client Home Page." http://squirrel-sql.sourceforge.net/, accessed May 2, 2014.
[28] H. Aman, "Tools | Research | Software Eng. Lab. | Center for Information Technology, Ehime University." http://se.cite.ehime-u.ac.jp/tool/, accessed March 10, 2015.
[29] N.V. Chawla, K.W. Bowyer, L.O. Hall, and W.P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," J. Artificial Intelligence Research, vol.16, pp.321–357, 2002.
[30] C.P. Dancey and J. Reidy, Statistics Without Maths for Psychology, Prentice Hall, Upper Saddle River, NJ, 2004.
[31] C. Schwarz, "Sampling, regression, experimental design and analysis for environmental scientists, biologists, and resource managers," Tech. Rep., Simon Fraser Univ., 2011.
[32] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," Proc. 2005 Int'l Workshop on Mining Softw. Repositories, pp.1–5, May 2005.
[33] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," Proc. Int'l Workshop on Predictor Models in Softw. Eng., p.9, May 2007.
[34] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," Proc. 34th Int'l Conf. Softw. Eng. (ICSE2012), pp.200–210, June 2012.
[35] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," Proc. 21st Int'l Conf. Program Comprehension (ICPC2013), pp.83–92, May 2013.
[36] D.J. Lawrie, H. Feild, and D. Binkley, "Leveraged quality assessment using information retrieval techniques," Proc. 14th Int'l Conf. Program Comprehension (ICPC2006), pp.149–158, June 2006.
[37] MIRA, "MISRA C." http://www.misra-c.com/, accessed June 30, 2014.
[38] Google, "Google Java Style." http://google-styleguide.googlecode.com/svn/trunk/javaguide.html, accessed Oct. 12, 2014.
[39] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," Proc. 24th Int'l Conf. Softw. Maintenance (ICSM2008), pp.277–286, Sept. 2008.
[40] O. Mizuno and M. Nakai, "Can faulty modules be predicted by warning messages of static code analyzer?," Advances in Software Engineering, vol.2012, no.924923, pp.1–8, May 2012.
[41] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, and K. Matsumoto, "Assessing the cost effectiveness of fault prediction in acceptance testing," IEEE Trans. Softw. Eng., vol.39, no.10, pp.1345–1357, Oct. 2013.

Hirohisa Aman received the Dr. degree in Engineering from Kyushu Institute of Technology, Kitakyushu, Japan, in 2001. Since 2001, he has been with Ehime University. He is currently an associate professor at Center for Information Technology, Ehime University. His primary research interests are in empirical approaches to software engineering. He is a member of Information Processing Society of Japan (IPSJ), Japan Society for Software Science and Technology (JSSST), Institute of Electrical and Electronics Engineers (IEEE), and Japan Society for Fuzzy Theory and Intelligent Informatics (SOFT).

Sousuke Amasaki received the Dr. degree in Information Science and Technology from Osaka University, Suita, Japan, in 2006. He was an assistant professor at Tottori University of Environmental Studies from 2006 to 2008. Since 2008, he has been with Okayama Prefectural University. He is currently an assistant professor at the Faculty of Computer Science and Systems Engineering, Okayama Prefectural University. His primary research interests are in software development effort estimation methods and empirical approaches to software engineering. He is a member of IPSJ, IEEE and Association for Computing Machinery (ACM).

Takashi Sasaki received the M.E. from Nara Institute of Science and Technology, Ikoma, Japan, in 2000. He was a doctoral student at Kyoto University from 2000 to 2004. Since 2006, he has been with Ehime University. He is currently an assistant professor at Center for Information Technology, Ehime University. His primary research interests are in information network technologies and information support. He is a member of IPSJ and IEEE.

Minoru Kawahara received the Dr. degree in Informatics from Kyoto University, Kyoto, Japan, in 2003. He was an assistant professor at Kyoto University from 1990 to 2004. Since 2004, he has been with Ehime University. He is currently a professor at Center for Information Technology, Ehime University. His primary research interests are in information network technologies, data mining and information support. He is a member of IPSJ.