Empirical Analysis of Fault-proneness in Methods by Focusing on their Comment Lines

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Japan 790–8577
Email: aman@ehime-u.ac.jp

Sousuke Amasaki
Faculty of Computer Science
and Systems Engineering
Okayama Prefectural University
Soja, Japan 719–1197
Email: amasaki@cse.oka-pu.ac.jp

Takashi Sasaki and Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Japan 790–8577
Email: sasaki@cite.ehime-u.ac.jp
Email: kawahara@cite.ehime-u.ac.jp

***Abstract*—This paper focuses on comments described in Java programs, and conducts an empirical analysis about relationships between comments and fault-proneness in the programs. The types of comments analyzed in this paper are comments described inside a method body (inner comments), and comments followed by a method declaration (documentation comments). Although both of them play important roles in the program comprehension, they seem to be described in different purposes; The inner comments are often added to present tips about code fragments, while the documentation comments usually work as a programmer's manual. In the field of code refactoring, well-written inner comments are said to be related to "code smell" since they may cover a lack of readability in a complicated code fragment. This paper analyzes the associations of comments with the code quality from the aspect of fault-proneness, with using four popular open source products. The empirical results show that a method having inner comments tends to be 1.8 – 3.0 times likely to be faulty. The key contribution of this work is to reveal the usefulness of inner comments to point at faulty methods.**

I. Introduction

Programming is one of the most basic activities in software development, and a successful quality management of programming is necessary to develop a high-quality software system. Except for the cases that the programs can be automatically generated from their formal specifications, the programming activities are done by human beings (programmers). That is to say, programmers' minds and experiences about the programs being developed would have much influence on the quality of their products.

The most important products produced by programmers are sequences of executable statements to implement functions required in the target system. Programmers often describe some comments along with executable statements in their source files as well. Since those comments are eliminated or ignored when the source files are compiled or interpreted, any comments are independent of the software functionality and performance. However, in many cases, comments may be helpful in understanding the program [1], [2]. Comments are important documentation artifacts [3], and have a certain level of impact on the software quality.

In general, comments improve the readability of programs, so they are harmless to the software quality. However, comments may sometimes be added to compensate for the lack of readability in complicated programs. Some programmers want to add in-depth comments to their code fragments which are hard to be understood by other developers. In other words, an appearance of well-written comments might reflect a lack of confidence about the program clearness in the programmer's mind. In such cases, Kernighan et al. [4] recommended rewriting such confusing code rather than adding careful comments onto those parts. In the field of code refactoring, well-written comments are known as artifacts related to "code smells" (signs of poor-quality programs to be refactored) [5]. While well-written comments themselves are harmless, they can play roles as "deodorant" beside bad source code. These pros and cons of well-written comments motivated our study.

In this paper, we focus on comments described for methods in source files, and conducts an empirical work using some open source products written in Java to statistically analyze the fault-proneness of well-commented methods. The key contribution of this paper is to show a notable relationship between the fault-proneness and the commenting manner in methods. The remainder of this paper is organized as follows: Section II describes the comments of interest in this paper. Section III presents our empirical study using four popular open source software projects, and discusses the results. Section IV gives brief descriptions of related work. Finally, Section V describes our conclusions and future work.

II. Target Comment Types

This section defines comment types to be analyzed in the following section.

There are various types of comments described in a source file. For instance, we often see the following types of comments [6]:

- A property information including the file name, copyright notice, version(s), author(s), change history, etc.
- A programmer manual explaining how to use the program, or how to use and/or call the function/method.
- A programmer note giving an information about the data structure and algorithm, or a tip to understand the code fragment—how the code fragment works, what the variable is, why the programmer wrote the code fragment, etc.
- A comment disabling a code which is referred to as "comment out."

- A comment to represent a separator line between functions/methods.

Since the aim of this paper is to analyze the impacts of comments on the fault-proneness, we focus on the part describing the sequence of executions, i.e. the functions/methods. Then, we will focus on the following two types of comments which are directly related to function/method implementations (see Fig. 1 for example):

1) *Documentation comments*: comments followed by a function/method declaration. Documentation comments usually provide the information about how to use the function/method. Those comments are known as Javadoc in Java. Notice that, as shown in Fig. 1, this paper does not limit the documentation comments to Javadocs (/∗∗ · · · ∗/ style comments). Indeed, some Java products provide their documentation comments in different styles using "// · · ·" type comments or "/∗ · · · ∗/" type comments.

2) *Inner comments*: comments written inside a function/method body, except for comment outed code[1]. Inner comments are often programmers' notes that explain the contents. Some programmers want to add inner comments when they feel their code fragments become complicated. This type of comment would be related to the "code smell" and "deodorant" problem mentioned in Sect.I.

■

Although the other types of comments would provide useful information for programmers, the focus of this paper is only on the above two types; Further analysis using other types of comments would be our future work.

III. Empirical Study

This section presents our empirical work, results and discussions. Our research objects are four popular open source products shown in Table I. We selected different sized products from different domains for the generality of our empirical results. We selected them to satisfy the following all requirements: They are 1) open source products, 2) written in Java, and 3) maintained with Git.
Requirement 1) is essential to analyze all comments appeared in all source files. Requirement 2) is from our data collection tools[2]: `JavaMethodExtractor`, `CommentCounter`, and `LOCCounter`. Requirement 3) is for the convenience of our fault-data collection scheme. Since Git provides various powerful functions for analyzing its repositories, it is easier to perform our empirical analysis than using conventional version control systems such as CVS and Subversion. For example, "`git log`" command can easily and fast extract commit logs which specific keywords (e.g. "bug") appeared in.

[1]It is hard to perfectly discriminate the comment-outed code from the normal comments (non comment-outed code). 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 [7] instead. While Aman's algorithm is a simple algorithm, the most of comments (98.9%) can be successfully discriminated.

[2]`http://se.cite.ehime-u.ac.jp/tool/`

```
public class Example
{
  /** [1. Documentation comments]
   * Returns true if this set contains no elemements.
   * @return true if this set contains no elemements
   */
  public boolean isEmpty()
  {
    // [2. Inner comments]
    // check if it's null, first
    if ( ary == null ) return true;
    if ( size == 0 ) return true;
    return false;
  }

  // [1. Documentation comments]
  // Reinitialize this set.
  public void reInit()
  {
     ......
```

Fig. 1. Examples of comments in Java.

TABLE I. Research objects.

product	domain	#files	size (KLOC)
Eclipse Checkstyle Plug-in[4]	coding style checker	239	21.1
Hibernate (O/R Mapping functionality)[5]	O/R mapper	7501	522.1
PMD[6]	code analyzer	1132	68.8
SQuirreL SQL Client[7]	SQL client	3770	387.5

We collected these four open source products from SourceForge.net[3], which is one of the most popular open source software development community sites. They were ranked in the top 20 of Java products at SourceForge.net.

A. Research Questions

We have a hypothesis that a more complicated, less quality program requires more additional information for the comprehension, so such a problematic program must need some comments to compensate for the lack of its comprehensibility. We designed our empirical work on the following research questions:

RQ1: Do the fault-proneness of methods significantly differ by the commenting style?
If there is a significant difference in the fault-proneness, our comment categorization and analysis are worthwhile.

RQ2: Does the amount of comments have any relationships on the fault-proneness?
If they have remarkable relationships with fault-proneness, it is valuable to care about the amount of comments during the development or code reviewing.

■

[3]`http://sourceforge.net/`
[4]`http://eclipse-cs.sourceforge.net/`
[5]`http://hibernate.org/`
[6]`http://pmd.sourceforge.net/`
[7]`http://www.squirrelsql.org/`

B. Metrics and Data Collection

To quantitatively analyze relationships between comments and fault-proneness in methods, we collect related data for each method by using the following metrics:

- Lines of Inner comments (LOI)
  The number of lines giving inner comments in the method.
- Lines of Documentation comments (LOD)
  The number of lines providing documentation comments for the method.

■

Metrics LOI and LOD are for capturing the amount of comments in a method. For example, method `isEmpty` in Fig.1 has LOI $= 2$ and LOD $= 4$.

Then, we collected the change history of each source file from Git, and extracted the change history of each method in the file. This activity consists of the following three steps:

1) Obtain the latest version of the source files, and make the complete list of methods appeared in the source files, except for any abstract methods; We used our tool `JavaMethodExtractor` to extract the methods. Table II shows the number of methods in each product.

2) For each method, collect its change history. In the concrete, examine whether the method is changed or not, by checking all commits corresponding to the source file which the target method is declared in.
   Let a method be faulty if one of its changes was a bug fixing; We decided a change to be a bug fixing when the corresponding commit's log includes one of bug-fix-related words—"bug, fix, defect." Table II shows the numbers of faulty methods and non-faulty methods.

3) For each method of its initial version, collect LOI value and LOD value. Tables III and IV show the distributions of LOI values and LOD values: the minimum, the 25 percentile, the median, the 75 percentile and the maximum of metric values. These tables reveal that there are many non-commented methods in our target products. The ratios of non-commented methods are shown in Table V. From Table V "the presence or absence" of comments can be a key category for observing the commenting styles, for examining RQ1.

■

C. Analytical Procedures

We analyze our empirical data through the following manner.

1) Analysis for RQ1

Our RQ1 is "Do the fault-proneness of methods significantly differ by the commenting style?" To examine this question, we conduct our empirical analysis by the following two steps.

TABLE II. NUMBERS OF METHODS, NON-FAULTY METHODS AND FAULTY METHODS.

product	#methods	non-faulty	faulty
Eclipse Checkstyle	1,051	971	80
Hibernate	18,483	18,321	162
PMD	3,970	3,821	149
SQuirreL	6,116	5,802	314

TABLE III. DISTRIBUTIONS OF LOI VALUES: MINIMUM, 25 PERCENTILE, MEDIAN, 75 PERCENTILE AND MAXIMUM.

product	min	25%	median	75%	max
Eclipse Checkstyle	0	0	0	1	26
Hibernate	0	0	0	0	42
PMD	0	0	0	0	69
SQuirreL	0	0	0	0	36

TABLE IV. DISTRIBUTIONS OF LOD VALUES: MINIMUM, 25 PERCENTILE, MEDIAN, 75 PERCENTILE AND MAXIMUM.

product	min	25%	median	75%	max
Eclipse Checkstyle	0	3	3	5	17
Hibernate	0	0	0	0	49
PMD	0	0	0	3	53
SQuirreL	0	0	0	3	95

TABLE V. RATIOS OF METHODS HAVING NO INNER COMMENTS (LOI $= 0$) AND ONES HAVING NO DOCUMENTATION COMMENTS (LOD $= 0$).

product	LOI $= 0$	LOD $= 0$
Eclipse Checkstyle	$\frac{724}{1051} = 0.689$	$\frac{197}{1051} = 0.187$
Hibernate	$\frac{16248}{18483} = 0.879$	$\frac{15008}{18483} = 0.812$
PMD	$\frac{3583}{3970} = 0.903$	$\frac{2891}{3970} = 0.728$
SQuirreL	$\frac{5467}{6116} = 0.894$	$\frac{3986}{6116} = 0.652$

1-1) Divide the set of all methods into subsets by their commenting styles. From the results of our data collection (see Tables. III, IV and V), categorize the methods into two subsets—"non-commented methods" and "commented methods," for inner comments ("LOI $= 0$" vs. "LOI > 0") and documentation comments ("LOD $= 0$" vs. "LOD > 0"), respectively.

1-2) For each comment type (inner comments and documentation comments), compare the fault-proneness of methods between subsets, by performing the following statistical test (χ^2 test)[8]:

[Null hypothesis]
The fault-proneness of non-commented methods is at the same level as that of commented methods;

[Alternative hypothesis]
The fault-proneness of non-commented methods differs from that of commented methods.

Note that the fault-proneness of methods is quantified using the ratio of faulty methods in the subset.

■

[8]This type of statistical test can be performed by using `prop.test` function in `R` (`http://www.r-project.org/`).

TABLE VI. RATIOS OF FAULTY METHODS AND STATISTICAL SIGNIFICANCES OF THEIR DIFFERENCES (LOI).

product	LOI = 0	vs	LOI > 0	significance
Eclipse Checkstyle	$\frac{38}{724}$ (0.0524)	$<$	$\frac{42}{327}$ (0.1284)	***
Hibernate	$\frac{115}{16248}$ (0.0071)	$<$	$\frac{47}{2235}$ (0.0210)	***
PMD	$\frac{119}{3583}$ (0.0332)	$<$	$\frac{30}{387}$ (0.0775)	***
SQuirreL	$\frac{259}{5467}$ (0.0474)	$<$	$\frac{55}{649}$ (0.0847)	***

(significance: "***" means p-value < 0.001)

TABLE VII. RATIOS OF FAULTY METHODS AND STATISTICAL SIGNIFICANCES OF THEIR DIFFERENCES (LOD).

product	LOD = 0	vs	LOD > 0	significance
Eclipse Checkstyle	$\frac{12}{197}$ (0.0609)		$\frac{68}{854}$ (0.0796)	
Hibernate	$\frac{137}{15008}$ (0.0091)		$\frac{25}{3475}$ (0.0072)	
PMD	$\frac{237}{2891}$ (0.0439)	$>$	$\frac{22}{1079}$ (0.0204)	***
SQuirreL	$\frac{198}{3986}$ (0.0497)		$\frac{116}{2130}$ (0.0545)	

(significance: "***" means p-value < 0.001)

If the above statistical test shows a significant difference in the fault-proneness of methods, our comment categorization and analysis are considered to be worthwhile.

2) Analysis for RQ2

Our RQ2 is "Does the amount of comments have any relationships on the fault-proneness?" To answer this question, we analyze the changes of fault-proneness in methods with increasing lines of comments, through the following two steps.

2-1) Categorize the set of commented methods into finer subsets, e.g. the set of methods whose LOI $= 1$, the set of methods whose LOI $= 2$, and so on.

2-2) Compare the ratios of faulty methods in the subsets, and check the changes in the ratio of faulty methods to explore if the amount of comments has an impact on the fault-proneness.
It is ideal to perform statistical tests like "step 1-2)" in this step as well. However, by dividing the set of commented methods into finer subsets such as LOI $= 1$ and LOI $= 2$, the number of methods belonging to each category becomes smaller. Therefore, no proper conclusion could be derived from a statistical test on such a small set of methods.

■

If there is a clear trend of the fault-proneness over the lines of comments, the amount of comments would be noteworthy during the development or code reviewing.

D. Results and Discussions

We present analytical results and discussions about our research questions in the followings.

1) RQ1: Do the fault-proneness of methods significantly differ by the commenting style?

Tables VI and VII, and Figs. 2 and 3 show ratios of faulty methods and statistical significances of their differences, by LOI and LOD, respectively.

For all products, the methods having one or more inner comments (LOI > 0) are about 1.8 – 3.0 times likely to be faulty than the ones having no inner comments (LOI $= 0$) (see Table VI and Fig. 2). These differences in the ratios of faulty methods are statistically significant (at $\alpha = 0.001$). Therefore, there is a trend which a method with inner comments is more faulty than a non-commented method.

These results support our hypothesis that a more complicated, less quality program requires more additional information for the comprehension. Furthermore, the results are consistent with one of the "code smells" that comments added to programs may work as "deodorant" beside poor quality code to be refactored [5].

The presence or absence of documentation comments did not show any specific tendency: while PMD showed a statistically significant difference in the ratio of faulty methods, the remaining products did not so (see Table VII and Fig. 3). Although documentation comments may provide useful information about the method, our empirical data did not show strong association with the fault-proneness. Further analysis of documentation comments will be our key future work.

In general, a larger program is more faulty, so we must consider the effect of code size (LOC) on the fault-proneness as well. We thus perform the logistic regression analysis with using all of LOC, LOI and LOD, to take their impacts apart:

$$\Pr(\ m_i \text{ is faulty} \mid \boldsymbol{x_i}\ ) = \frac{1}{1+e^{-y(\boldsymbol{x_i})}}, \tag{1}$$

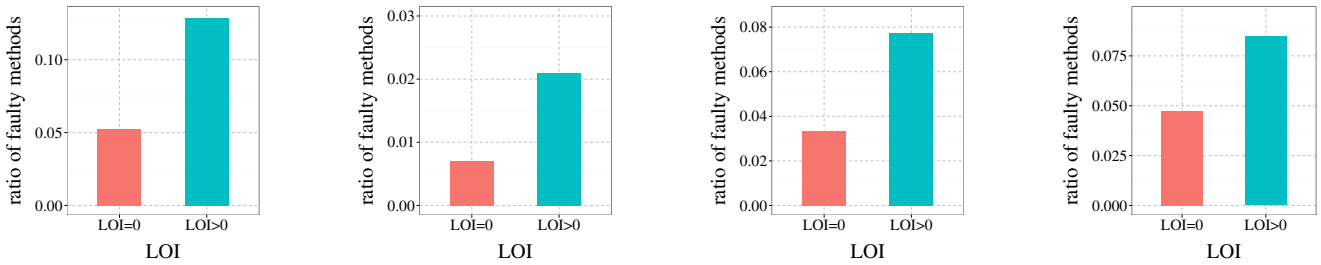
$$y(\boldsymbol{x_i}) = \beta_0 + \beta_1 \mathrm{LOC}_i + \beta_2 \overline{\mathrm{LOI}_i} + \beta_3 \overline{\mathrm{LOD}_i}, \tag{2}$$

where m_i is the i-th method and $\boldsymbol{x_i}$ is the vector of metric values in m_i, $\boldsymbol{x_i} = (\mathrm{LOC}_i, \overline{\mathrm{LOI}_i}, \overline{\mathrm{LOD}_i})$; LOC_i signifies LOC value of m_i; $\overline{\mathrm{LOI}_i}$ and $\overline{\mathrm{LOD}_i}$ are binary variables about m_i as follows:

$$\overline{\mathrm{LOI}_i} = \begin{cases} 0 & (\text{LOI} = 0 \text{ in } m_i), \\ 1 & (\text{LOI} > 0 \text{ in } m_i), \end{cases}$$

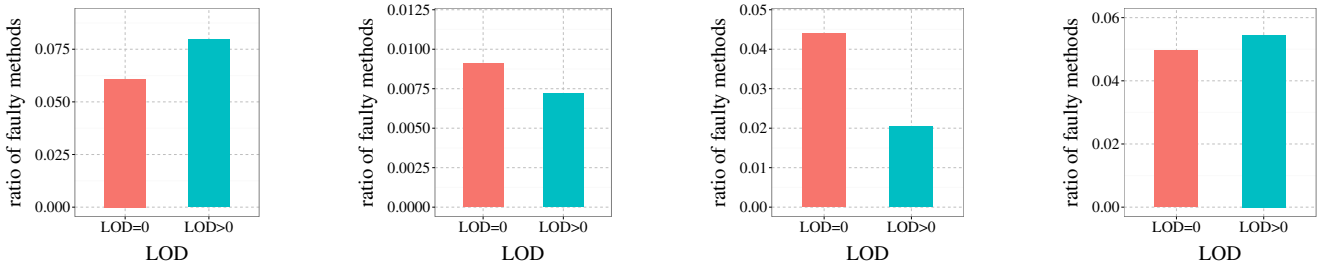
$$\overline{\mathrm{LOD}_i} = \begin{cases} 0 & (\text{LOD} = 0 \text{ in } m_i), \\ 1 & (\text{LOD} > 0 \text{ in } m_i). \end{cases}$$

Equations (1) and (2) express the logistic regression model which provides the probability of that "m_i is faulty" when three metric values ($= \boldsymbol{x_i}$) are given. In Eq. (2), β_0, β_1, β_2 and β_3 are constants, and β_1, β_2 and β_3 mean the impact level of LOC_i, $\overline{\mathrm{LOI}_i}$ and $\overline{\mathrm{LOD}_i}$ on the probability (fault-proneness), respectively. If β_j is not zero, the corresponding metric has a significant impact on the probability computation (for $j = 1, 2, 3$).



(a) Eclipse CheckStyle (b) Hibernate (c) PMD (d) SQuirreL

Fig. 2. Ratios of faulty methods: LOI= 0 vs. LOI> 0.



(a) Eclipse CheckStyle (b) Hibernate (c) PMD (d) SQuirreL

Fig. 3. Ratios of faulty methods: LOD= 0 vs. LOD> 0.

TABLE VIII. RESULTS OF LOGISTIC REGRESSION ANALYSIS: COEFFICIENTS AND THEIR SIGNIFICANCES.

product	Intercept (β_0)	LOC (β_1)	LOI> 0 (β_2)	LOD> 0 (β_3)
Eclipse Checkstyle	-3.25 (***)	0.00 ( )	1.01 (***)	0.41 ( )
Hibernate	-4.92 (***)	0.01 (*)	1.01 (***)	-0.34 ( )
PMD	-3.39 (***)	0.03 (***)	0.26 ( )	-0.90 (***)
SQuirreL	-3.16 (***)	0.01 (***)	0.34 (*)	0.08 ( )

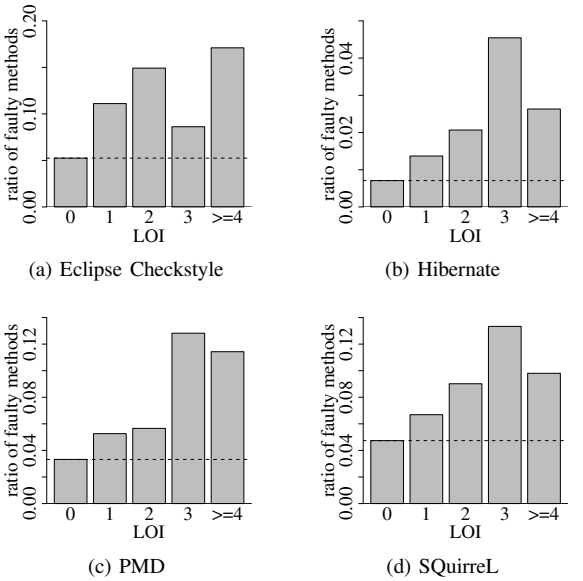
(significance: "***" means p-value < 0.001; "*" means p-value < 0.05)

Table VIII shows the results of logistic regression analysis. The asterisk(s) indicate the level of statistical significance of that the corresponding constant β_j is not zero (for $j = 1, 2, 3$). For each Eclipse Checkstyle, Hibernate and SQuirreL, the presence of inner comments (LOI $>$ 0) has a significant positive impact on the fault-proneness, even though LOC is also one of the explanatory variables. On the other hand, for PMD, the presence of inner comments has no significant impact, but the presence of documentation comments has a significant negative impact on the fault-proneness. This result is corresponding to the result in Table VII and Fig. 3.

The result of logistic regression analysis (Table VIII) signifies that the comparisons in Tables VI and VII are not dominated by the code size (LOC). That is to say, it is worthwhile to see associations of comments with fault-proneness.

2) RQ2: Does the amount of comments have any relationships on the fault-proneness?

The results for RQ1 showed the trends that the presence of inner comments may be related to the fault-proneness in methods. Then, we focus on the amount of inner comments to answer to RQ2. Since most of LOI values in our data sets are less than 4, we examined the ratios of faulty methods for LOI $= 0$, 1, 2, 3 and LOI ≥ 4.



(a) Eclipse Checkstyle (b) Hibernate (c) PMD (d) SQuirreL

Fig. 4. Changes in the ratio of faulty methods by LOI.

Figure 4 shows the results. The ratios of faulty methods monotonically increase from LOI $= 0$ to LOI $= 2$. The ratios in LOI $= 3$ and LOI ≥ 4 are not changed monotonically but both of them keep higher levels than LOI $= 0$. From these results, we can conclude that more inner comments do not result in higher fault-proneness. Nevertheless, the presence of inner comments is associated with the fault-proneness in methods.

While some previous work revealed that well-commented source files are likely to be faulty, those results were derived from a rough-grained analysis at source file level. We performed a finer-grained analysis at method level, and found that even one or two inner comments can point to faulty methods. This is a novel finding about the relationships between comments and fault-proneness in methods.

E. Threats to Validity

While our empirical work used popular open source products, the other products written in languages other than Java were not examined. Since our data collection tools can work for only Java, our empirical work was limited to Java products. However, our metrics and empirical analysis approach are available when the target product is written in C, C++ or C# without any modifications. In many cases, writing program elements (statements and conditions) is common among Java, C, C++ and C#, so the difference in language might not invalidate the contribution of this paper.

Since the programming activities for commercial products are usually managed by their companies/organizations, their commenting styles would depend on the companies/organizations, so they may bring different trends of comments. Further study on commercial products is one of our important future work.

Our fault data collection was based on a keyword matching in commit logs, so we might miss some of the true faulty methods. However, many studies analyzing code repositories have adopted such a keyword matching approach in the past, and we also followed those manner to detect bug fixing commits. We would like to use a richer data-collection mechanism collaborating with a bug tracking system in our future work.

Our analysis was based on the number of comment lines, not on the contents of comments. The consistency between a program and corresponding comments [6] may affect our results. While "what the inner comments say" would be an important factor, we empirically showed that even "the presence or absence" of inner comments has a significant impact on the fault-proneness in this paper.

IV. Related Work

Tenny [1] and Woodfield et al. [2] 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 impacts of comments on fault-proneness.

Steidl et al. [6] proposed a framework to evaluate the quality of comments. They evaluated the coherence between the documentation comments (which are called "member comments" in [6]) and the name of the corresponding method. Their coherence evaluation presents whether the documentation comments provide useful information about the method. They discussed the inner comments (which were called "inline comments" in [6]) as well, and proposed to use short inner comments as signs of parts to be refactored. Moreover, they described that the existence of long inner comments may infer a lack of external documents, so adding many inner comments were not recommended. While they studied valuable relationships between comments and source code, their focus was on the quality of comments, but not on the code quality.

Aman [8], [7] conducted some empirical studies on relationships between the amount of comments and fault-proneness in source files. Those empirical studies reported that well-commented programs were likely to be faulty. Those reports are our previous work, but they focused only on the inner comments and analyzed the fault-proneness at the source file level. This paper studies not only the inner comments but also the documentation ones, and performs finer analysis at the method level rather than the source file level[9].

V. Conclusions

This paper focused on two types of comments related to methods—inner comments and documentation comments—. We conducted an empirical analysis using four popular open source products written in Java, to examine the relationships between comments and fault-proneness in their methods. Our empirical results revealed a novel finding that even one or two inner comments can point to faulty methods. That is to say, comments written inside a method body may tell us complicated code fragments that we should check carefully.

Our future work includes further analysis using various products from different domains and organizations, and written in other languages, time series analysis of relationships between comments and fault-proneness, and further analysis focusing on contents of comments.

Acknowledgments

This work is supported by JSPS KAKENHI Grant #25330083.

References

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

[2] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proc. 5th Int'l Conf. Softw. Eng.*, San Diego, California, 1981, pp. 215–223.

[3] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Int'l Conf. Design of Communication*, Coventry, UK, Sept. 2005, pp. 68–75.

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

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

[6] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int'l Conf. Program Comprehension*, San Francisco, California, May 2013, pp. 83–92.

[7] H. Aman, "An empirical analysis on fault-proneness of well-commented modules," in *Proc. 4th Int'l Workshop Empir. Softw. Eng. in Practice*, Osaka, Japan, Oct. 2012, pp. 3–9.

[8] ——, "An empirical analysis of the impact of comment statements on fault-proneness of small-size module," in *Proc. 19th Asia-Pacific Softw. Eng. Conf.*, Hong Kong, Dec. 2012, pp. 362–367.

[9]The basic idea of this work was presented at the poster session of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM 2014), Sept.18-19, 2014 in Torino, Italy.