An Empirical Analysis of the Impact of Comment Statements on Fault-Proneness of Small-Size Module

Hirohisa Aman
Graduate School of Science and Engineering
Ehime University
Matsuyama, Japan 790–8577
Email: aman@cs.ehime-u.ac.jp

***Abstract*—Code size metrics are commonly useful in predicting fault-prone modules, and the larger module tends to be more faulty. In other words, small-size modules are considered to have lower risks of fault. However, since the majority of modules in a software are often small-size, many "small but faulty" modules have been found in the real world. Hence, another fault-prone module prediction method, intended for small-size module, is also required. Such a new method for small-size module should use metrics other than code size since all modules are small size. This paper focuses on "comments" written in the source code from a novel perspective of size-independent metrics; comments have not been drawn much attention in the field of fault-prone module prediction. The empirical study collects $11,512$ small-size modules, whose LOC are less than the median, from three major open source software, and analyzes the relationship between the lines of comments and the fault-proneness in the set of small-size modules. The empirical results show the followings: 1) A module in which some comments are written is more likely to be faulty than non-commented ones; the fault rate of commented modules is about 1.8 – 3.5 times higher than that of non-commented ones. 2) Writing one to four lines of comments would be thresholds of the above tendency.**

***Index Terms*—Comment; metric; fault-prone module; small-size module**

I. Introduction

Early fault detection is key in developing and maintaining software systems. Although a thorough code review [1], [2] of all software components is one of the most effective means for detecting faults, such an exhaustive review would be almost impossible because of some practical restrictions such as a tight budget, insufficient manpower and lack of time. Hence, the prediction of the presence of faults in modules is a promising approach to promote efficient review activities.

There has been considerable research on predicting fault-prone modules by various metrics and mathematical models (e.g., [3]–[7]). In many studies, the module size can be a useful metric to predict fault-prone modules [3], [4]. That is to say, the larger module tends to be more faulty. For instance, in the case of Eclipse [8] ver.3.0, the rate of faulty modules becomes higher when increasing the module size (lines of code: LOC) as shown in Fig.1. Such data may lead to the conclusion that the small-size module has a low risk of fault. This trend seems to be common in the development of software.

However, we often see many faulty modules even if they are small-size ones as shown in Fig.2. Although a small-size module has a low risk of fault, there are a lot of small-size modules in a software; the distribution of code size (LOC) is usually right-skewed (right tail is longer) as shown in Fig.3. That is, the majority of modules are small-size modules. Therefore, we do not have to downplay the fault-proneness of small-size modules, and we need to develop another method to predict fault-prone modules, intended for small-size modules.

If we focus on only small-size modules, the code size metrics might be less effective because all modules are small. Thus, the metrics other than code size ones may become relatively useful in predicting fault-prone modules. In this paper, we will develop a new perspective of size-independent metric focusing on "comments" written in source files. While comments are familiar to practitioners, there have not been many studies about the relationship between the comments and the fault-proneness of software modules in the past.

Since any comments have no contribution to the program behavior, they have often been neglected in analyzing the fault-proneness of modules. Meanwhile, there have been discussions about the indirect impact of comments on the code quality. Kernighan and Pike [9] pointed out that we should not comment bad code; we should rewrite it in order to be easily understood without comments. Fowler [10] described the comments may be related to "bad smells" for refactoring. More precisely, comments are said to be the deodorant rather than the bad smell, so comments are good entity for enhancing the code understandability, but they may also cover up a problematic code. That is to say, the presence of comments may be a sign of complicated code, and be a useful clue to find faults during the visual code review. The contribution of this paper is to statistically show the impact of comments on the fault-proneness of small-size modules. The results will provide a useful information about fault-proneness to practitioners.

The remainder of this paper is organized as follows: Section II describes the definitions of comments and metrics used in this paper. Then, section III presents our empirical study analyzing the impact of comments on the fault-proneness of small-size modules. Section IV and V discuss the related work

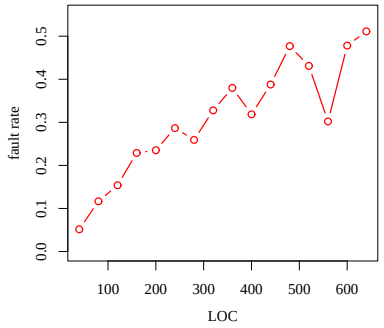

Fig. 1. Fault rate in Eclipse ver.3.0.

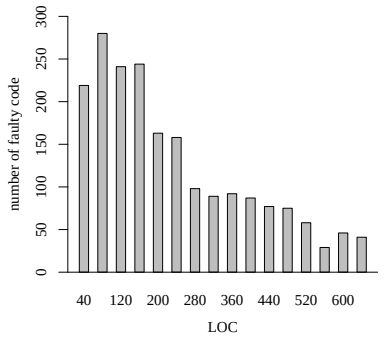

Fig. 2. Number of faulty modules in Eclipse ver.3.0.

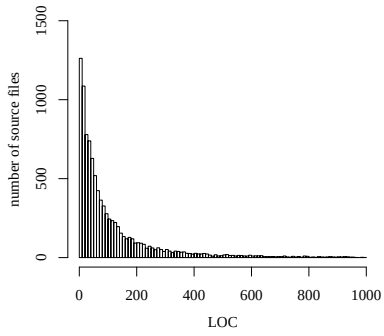

Fig. 3. Distribution of LOC values in Eclipse ver.3.0 (Note: LOC $\geq$ 1000 omitted).

and the threats to validity of our work, respectively. Finally, section VI provides our conclusion and future work.

II. Definition

This paper analyzes comments in Java source programs, quantitatively. The Java language specification [11] defines the following two kinds of comments:

- Traditional comment: a text from "/*" to "*/"; a doc comment for Javadoc [12], written in from "/**" to "*/," is a special type of traditional comment.
- End-of-line comment: a text from "//" to the end of line.

Now we introduce a comment-related metric in order to analyze the impact of comments on the fault-proneness.

Definition 1 (Lines of Comments: LCM)

Given a source file. A source line in which any comments appear is a "commented line." Then, let the lines of comments (LCM) be the total number of commented lines written inside the method bodies[1] in the source file. However, the following types of comments are excluded from the LCM measurement:

- Comments written outside method bodies (e.g., Javadoc).
- Comment out—disabling a code fragment.

□

The larger LCM indicates that the more comments are written to explain the source code, i.e., more memos by the developers. It should be noted that LCM does not count "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 the LCM measurement as they have different meanings—they are not developer's memos. 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 code fragment is a comment out or not, we should query each developer for each code fragment. However, such a follow-up investigation is practically impossible, so we will use the following simple detection algorithm [13], [14] instead of such follow-up study.

Algorithm 1 (Simple Detection of Comment Out)

Given a comment. Let C be the content of given comment, which excludes the special strings indicating the start and end of comment; C is the string from just behind "/*" to just before "*/" in the case of traditional comment, 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 the followings: (1) semicolon ";", (2) left curly brace "{" and (3) right curly brace "}", then regard C as a comment out; a non-white-space character is a character excluding the followings: the ASCII space (SP), horizontal tab (HT), form feed (FF), line feed (LF), and carriage return (CR).

□

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, it has a practical level of accuracy detecting comment outs[4]. We will use the above simple algorithm in our empirical work.

To discuss the fault-proneness quantitatively, we introduce the notion of fault rate (FR) [14] as well.

[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.

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

[4]An empirical work reported that the algorithm successfully detected about 99% of comments/comment outs for a middle-scale open source software, Azureus 4.2.0.4 [13], [14].

Definition 2 (Fault Rate: FR)

Given a set of source files. Define the fault rate (FR) in the set as the following equation:

$$\mathrm{FR} = \frac{\text{Number of source files in which a fault is found}}{\text{Number of source files}}.$$

□

FR is the ratio of faulty source files (modules) in a given set of source files. Hence, the higher FR means that the source files in the set are more suspect to have faults, i.e., fault-prone.

FR is a simple metric based on the presence or absence of fault in a module, and we focus on the following simple question in this paper: "whether the comment statement can actually be a significant sign of potential fault in small-size module or not." While we can also consider more detailed metrics of fault-proneness such as the number of faults or the fault density (bug density), first we should answer the above simple question using FR. We would like to conduct further analysis using such detailed metrics in the future.

III. Empirical Study

In this section, we conduct an empirical study to analyze the impact of comments on the fault-proneness of small-size modules. Our experimental procedure is as follows.

(1) Source Code Collection (Sect.III-A):
Collect small-size modules (source files) from some projects. We will consider the module whose LOC is less than the median LOC to be "small size," reflecting the right-skewed distribution as shown in Fig. 3.

(2) Measurement of LCM (Sect.III-B):
Measure LCM values of all the collected small-size modules using our metric tools, JavaMethodExtractor[5] and MethodCommentCounter.

(3) Impact Analysis of Comment on FR (Sect.III-C):
Compute the FR of non-commented modules (whose LCM $= 0$) and the FR of modules in which some comments are written (LCM $\geq \tau$) (for $\tau = 1, 2, \ldots$). Then, perform a chi-square (χ^2) test to see whether the two FRs show a significant difference or not.

A. Source Code Collection

We collected our experimental objects (small-size modules) from three major open source software shown in Table I — Ant [15], Eclipse and Xerces [16]. We selected them because they satisfied the following requirements (a)–(c):

(a) The source files are available.
(b) The source files are written in Java.
(c) The information about the presence of post-release fault is available for each source file; a post-release fault is a fault found after the software was released.

□

The requirement (a) is essential for measuring LCM. Any open source software satisfies (a). The requirement (b) comes from our metric tools. The requirement (c) is absolutely necessary to analyze the fault-proneness of modules. We used PROMISE data [17] as our data source satisfying (c); it is possible to use other data repositories, or extract such fault data from a pair of a code repository and a bug tracking system.

Table II shows the distribution of code sizes (LOC values) in our data set. We can see that the majority are small-size modules for each software, as like Fig.3. Then, we will consider a module whose LOC is less than the median LOC (67 for Ant, 58 for Eclipse and 36 for Xerces; see Table II) to be a small-size module, and our experimental object in this paper. In this study, $11,512$ source files are collected from the above three open source software projects as the small-size modules for our analysis (see Table III).

B. Measurement of Lines of Comments (LCM)

We measured LCM values of our $11,512$ small-size modules. Figure 4 and Table IV show the distribution of LCM values in our data set. The majority of the modules are "non-commented" modules as shown in Fig.4 and Table IV: LCM $= 0$ for about 65 – 94% of the modules. Table V shows the distribution of LCM values, excluding non-commented (LCM $= 0$) modules. The most of small-size modules have low LCM values: the 75 percentiles of LCM are 7 to 10. That is to say, most small-size modules have a few or no lines of comments.

C. Impact Analysis of Comment on Fault Rate (FR)

We conducted a statistical analysis on the difference between FR of non-commented modules and FR of commented

TABLE I
Open source software projects used in our empirical study.

software	version	number of modules	total LOC
Ant	1.3	122	14,230
	1.4	173	20,499
	1.5	286	33,694
	1.6	343	45,796
	1.7	720	87,334
Eclipse	2.0	6,143	765,765
	2.1	4,838	690,736
	3.0	9,727	1,262,893
Xerces	1.2	411	60,032
	1.3	425	64,503
total		23,188	3,045,482

TABLE II
Distribution of LOC values.

software	min	25%	median	75%	max
Ant	2	30	67	147	1398
Eclipse	4	22	58	142	5228
Xerces	4	17	36	103	3578

TABLE III
Experimental objects.

software	Ant	Eclipse	Xerces	total
number of modules	811	10,288	413	11,521

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

modules, for each software. Let FR_0 be the FR of the modules whose LCM $= 0$, and let $FR(\tau)$ be FR of the modules whose LCM $\geq \tau$. Then, for each threshold[6] $\tau = 1, 2, 3, \ldots$, we performed a chi-square (χ^2) test to determine whether there is a significant difference between FR_0 and $FR(\tau)$ or not. Our hypotheses are as follows:

- H_0: The null hypothesis is that the two populations from which the modules were drawn have the same true proportion of faulty ones ($FR_0 = FR(\tau)$).
- H_1: The alternative is that those proportions are different ($FR_0 \neq FR(\tau)$).

□

The results are shown in Figs.5–7 and Table VI–VIII. For all τ, we see significant differences between FR_0 and $FR(\tau)$. That is to say, the FR of non-commented modules significantly differs from the FR of modules who has one or more comment lines. Moreover, $FR(\tau)$ slightly increases as τ gets higher.

D. Discussion

The above results show that LCM has an impact on the fault-proneness of small-size modules. The modules having some comments are double or triple fault-prone than non-commented ones. The threshold of LCM to statistically show a significant difference in FR is 1 for Eclipse and Xerces, and that is 4 for Ant, with the level of significance of 1% (see Figs.5–7). In each case, even if LCM is a small number, the presence of comment makes a definite impact on the fault-proneness of small-size module. Our empirical results may relate to one of the bad code smells [10] for refactoring.

We do not mean that comments are bad entities for the code quality. Since comments have no contribution to the behavior nor the structure of software, the comments themselves can not be the cause of faults. Thus comments have no direct relationship with any faults in modules. However, an indirect relationship between the comments and the fault-proneness is present. Comments written in a module may be useful to enhance the readability and/or understandability of the module, and will be good entities for the code. Because of the favorable effect of comments, some comments are often written for covering up problematic parts which are complicated. Therefore, a comment may be a useful sign of potential faults. Such a commented code should be sophisticated (rewritten) so as to

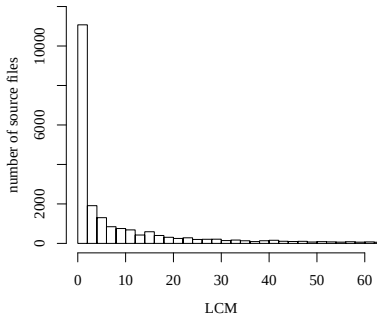

Fig. 4. Distribution of LCM values (Note: LCM ≥ 60 omitted).

TABLE IV
DISTRIBUTION OF LCM VALUES.

software	min	25%	median	75%	max
Ant	0	0	0	3	120
Eclipse	0	0	0	2	183
Xerces	0	0	0	0	14

TABLE V
DISTRIBUTION OF LCM VALUES (EXCLUDING LCM$= 0$).

software	min	25%	median	75%	max
Ant	1	3	5	10	120
Eclipse	1	2	3	8	183
Xerces	1	4	6	7	14

[6]To ensure a sufficient number of samples, we limited τ to the 75 percentile shown in Table V.

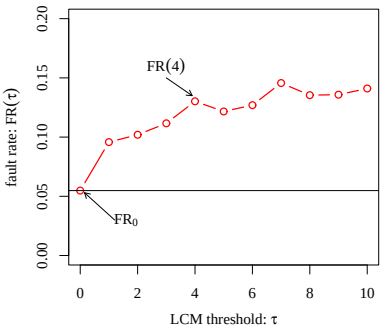

Fig. 5. $FR(\tau)$ in Ant.

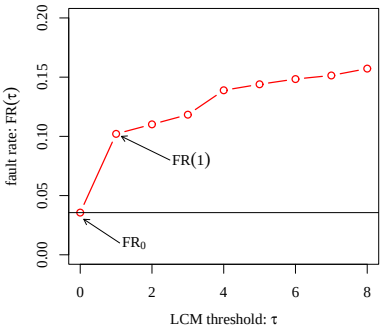

Fig. 6. $FR(\tau)$ in Eclipse.

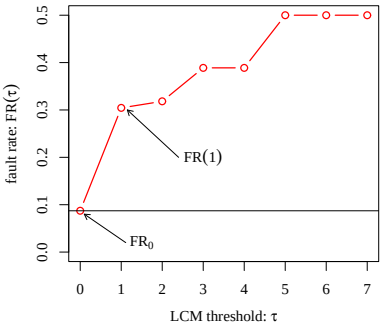

Fig. 7. $FR(\tau)$ in Xerces.

be understandable without detailed comments.

In general, a larger module is more faulty. Since small-size modules are often treated as low risk ones, a lot of latent faults in small-size modules might be overlooked. To find faulty modules effectively even in small-size modules, metrics other than code size should also be developed. The contribution of this paper is to show that LCM can be one of useful metrics related to the fault-proneness of small-size modules.

IV. Related Work

Since the comments have no contribution to the execution of programs, most of software metric-based studies have omitted the comments in their program analysis. There have been a few studies focusing on the comments as their major objects.

TABLE VI
$FR(\tau)$ IN ANT AND STATISTICALLY-SIGNIFICANT DIFFERENCES WITH FR_0.

FR_0	τ	$FR(\tau)$	$FR(\tau)/FR_0$
0.0548	1	0.0957*	1.75
	2	0.102*	1.76
	3	0.112*	2.04
	4	0.130**	2.37
	5	0.122**	2.23
	6	0.127**	2.32
	7	0.146**	2.66
	8	0.135**	2.46
	9	0.136*	2.48
	10	0.141*	2.57

(*: p-value $< 5\%$; **: $< 1\%$; ***: $< 0.1\%$)

TABLE VII
$FR(\tau)$ IN ECLIPSE AND STATISTICALLY-SIGNIFICANT DIFFERENCES WITH FR_0.

FR_0	τ	$FR(\tau)$	$FR(\tau)/FR_0$
0.0356	1	0.102***	2.87
	2	0.110***	3.09
	3	0.118***	3.31
	4	0.139***	3.90
	5	0.144***	4.04
	6	0.148***	4.16
	7	0.151***	4.24
	8	0.157***	4.41

(*: p-value $< 5\%$; **: $< 1\%$; ***: $< 0.1\%$)

TABLE VIII
$FR(\tau)$ IN XERCES AND STATISTICALLY-SIGNIFICANT DIFFERENCES WITH FR_0.

FR_0	τ	$FR(\tau)$	$FR(\tau)/FR_0$
0.0872	1	0.304**	3.49
	2	0.318**	3.65
	3	0.389***	4.46
	4	0.389***	4.46
	5	0.500***	5.73
	6	0.500***	5.73
	7	0.500**	5.73

(*: p-value $< 5\%$; **: $< 1\%$; ***: $< 0.1\%$)

Tenny [18] has empirically studied the impacts of the procedure structure and the presence of comments on the program readability. The empirical results showed that comments help to improve the readability. Although the study discussed the impact of comments, the empirical study was conducted on a program which has no fault, so Tenny's study had a different standpoint with our study. Buse *et al.* [19] have empirically discussed the ability of comments for enhancing the program readability as well. They pointed out that the comments could be used to adjust the readability upward for an unreadable code. Their basic viewpoint about the comments is close to our study, but there is a fundamental difference in the focus: while their study focused on the program readability, our study addresses the fault-proneness of small-size modules.

Fluri *et al.* [20] have analyzed co-changes between the modules and their comments during the software evolution. They reported some useful findings such that some updates of comments lagged behind the updates of code. While their study was interesting in analyzing the writing-comment activities, the impacts on fault-proneness were not discussed.

This work further studies the previous analysis [14]. Its empirical work classified a set of modules into three subsets: non-commented ones, low-commented ones, and high-commented ones. Then, it statistically showed a basic trend that the high-commented modules was the most fault-prone. In this paper, to discuss such a trend in more detail, we focused on the small-size modules and analyzed the relationship between the lines of comments (LCM) and the fault-proneness.

Some studies in the mining software repositories [21], [22] said that a module which required many bug fixes in the past would have another bug (fault), so such a module is fault-prone. Such a module may cause more comment descriptions and/or comment outs as well. Further investigation from such a point of view will be our significant future work.

V. Threats to Validity

This study focused on the lines of comments (LCM) as a novel measure related to the fault-proneness of small-size modules, since comments may be an important sign regarding "code smell." Our empirical work used the data about whether a fault was found in the module or not, so we did not take into account the severity of faults and the number of faults in the modules. While the primary contribution of this paper is to introduce yet another perspective of fault-prone module prediction, such roughness of data may decrease usefulness of our empirical results. That is to say, our results might not be useful in finding problematic modules who have many critical faults; we need to discuss the fault severity and the number of faults as well. Moreover, we could not associate comments with faults, directly. Since our data collection scheme is insufficient for conducting further analysis regarding the number of faults, severity of faults and comment-fault connections, we have to reconsider our scheme first. We will need to conduct another enhanced data collection and more detailed analysis.

LCM may correlate with LOC in many cases, i.e., LOC may be a confounding factor. Since we used only small-

size modules, the influence of LOC would be less in our results: there were weak positive correlations between LOC and LCM—the correlation coefficients were 0.300, 0.337 and 0.246 for Ant, Eclipse and Xerces, respectively. Nevertheless, we have not solved the problem of confounding factor. To discuss the confounding factors and evaluate the impact of LCM more properly, we should do a stratification by LOC and perform additional analysis using LCM and other metrics.

Our data did not consider the timing of comments—when the comments are written. Some age-old comments might be no longer in use and have no impact on the code quality. Such tracking comments and their analysis would also be required.

Since our results are from only three open-source software written in Java, our findings may not be general to all kinds of software. Other open-source or commercial software projects have different coding styles and rules in their developments, so our results may not be useful for their quality managements. While we should perform further analysis on other projects including ones written in another language (not Java), our work will be a useful motive for the discussion about the comments toward the fault-prone module prediction.

We did not analyze "comment out" and Javadoc. They may also have other impacts on the fault-proneness in this study.

VI. Conclusion

Smaller modules have been considered less serious in the studies of predicting faulty modules. However, we often see many "small but faulty" modules in actual software systems. Although each small-size module has a low risk of fault, there are a lot of small-size modules—a majority of modules are small-size ones. As a result, the total number of faulty modules is relatively high even if they are small-size modules. Thus, we focused on "small-size modules" and approached their fault-proneness from a new perspective other than code size—the amount of comments written in modules.

Our empirical work collected a lot of small-size modules, whose LOC are less than the median, from three major open source software, Ant, Eclipse and Xerces. Then we statistically analyzed an impact of the comment(s) on the fault-proneness. The empirical results showed the followings: (1) Modules having some comments are more likely to be faulty than non-commented modules: the fault rate of commented modules is about 1.8–3.5 times higher than that of non-commented ones. (2) Writing one to four lines of comments would be thresholds of the above tendency.

It is generally preferable that a small-size module is understandable even if it has no comment because of the smallness. Our empirical results mean that a small-size module with some comments may contain some problematic code. It should be noted that we did not say comments are bad entities; while comments are good entities to enhance the code readability, they may cover up even problematic code's complexity.

It is easier to perform a visual inspection on smaller module. During such visual inspections, comments will be useful clues to finding faulty modules in even the set of small-size modules. Our future work will conduct further investigations into the impact of comments on the fault-proneness in order to enhance the reliability of our empirical results, using the fault severity, the number of faults (fault density) and so on. An analysis of the quality of comments themselves will also be our important future challenge.

Acknowledgment

The author would like to thank the anonymous reviews for their helpful comments on an 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.

References

[1] T. Glib and D. Graham, *Software Inspection*. Boston, MA: Addison-Wesley Professional, 1993.

[2] K. Wiegers, *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison-Wesley Professional, 2002.

[3] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[4] 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, Mar. 2009.

[5] L. C. Briand, W. L. Melo, and J. Wüst, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 760–720, Jul. 2002.

[6] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.

[7] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 675–686, Oct. 2007.

[8] The Eclipse Foundation. (2012) Eclipse - the eclipse foundation open source community website. [Online]. Available: http://www.eclipse.org/

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

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

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

[12] Oracle. (2011) Javadoc technology. [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/javadoc/

[13] H. Aman, "An empirical study on relationship between comment description and fault rate in open source software," in *Software Engineering Front 2010*, M. Matsushita and O. Shigo, Eds. Tokyo: Kindai Kagaku sha, 2010, pp. 97–100, in Japanese.

[14] ——, "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.

[15] Apache Software Foundation. (2011) Apache Ant. [Online]. Available: http://ant.apache.org/

[16] ——. (2012) The Apache Xerces Project. [Online]. Available: http://xerces.apache.org/

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] Google. (2011) Bugspots - bug prediction heuristic. [Online]. Available: https://github.com/igrigorik/bugspots