Doctoral Dissertation

Understanding the Impact of Human Factors on Programming Activity

Aji Ery Burhandenny

June 29, 2018

Graduate School of Science and Engineering
Ehime University

A Doctoral Dissertation
submitted to Graduate School of Science and Engineering,
Ehime University
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Aji Ery Burhandenny

Thesis Committee:

Associate Professor	Hirohisa Aman	(Supervisor)
Professor	Minoru Kawahara	(Co-supervisor)
Professor	Hiroshi Takahashi	(Co-supervisor)
Professor	Yoshinobu Higami	(Co-supervisor)

Abstract

The widespread usage of computer systems and its' software components in our social life demands a successful quality management of software development. Since the software development is driven by human programmers, human factors may have large impacts on the software quality. The aim of this study is to empirically examine the important of focusing on human factors in the software quality management.

This dissertation focuses on two properties: "comment" and "coding violation" as main parameters to investigate the traits of programmer's habit in programming activities. Both properties are expected to be inter-related and totally dependent on individual programmer's preference.

The contributions of this study are: (1) To evaluate the differences in comment densities among individual programmers, and to propose to adjust the conventional code complexity metric (the cyclomatic complexity) by using the abnormality of the comment density; The findings show that the proposed metric is a better predictor of change-prone programs. (2) To examine the coding violation trend across all versions of products. The findings show that there is a diversity of important violations among projects and about 12% of disregarded violations are shared among programmers across projects; (3) To examine the influence of authoring types to coding violations. The findings show that the difference in the authoring type has significant impacts on the evaluations of violations: violations appearing in single-authored files may have big gaps with that in multi-authored files, but about 30% of violations would be commonly worthless across projects for many programmers; (4) To investigate which violation is familiar with more programmers and frequently appears in many source files (having a high coverage), and which violation is really related to bugfixes (having a high importance). The findings show that the familiar violations tend to differ among projects, and only 25 violations are common to all surveyed projects, while the trends of their importance vary from project to project.

Through the above studies, we empirically proved the importance of focusing on human factors, i.e., the individual differences among programmers for evaluating the software quality. By combining well-defined general guidelines with a mechanism of automatic tuning based on the individual programmer's data, such as "comments" and "coding preferences/habits" we presented, we would obtain a more sophisticated software development environment.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Prof. Hirohisa Aman for the continuous support of my doctoral study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of researches and writing of this thesis. I could not have imagined having a better advisor and mentor for my doctoral study.

I would like to thank the rest of my co-supervisors and thesis committee: Prof. Minoru Kawahara, Prof. Hiroshi Takahashi, and Prof. Yoshinobu Higami, for their insightful comments and encouragement to widen my research from various perspectives.

My sincere thanks to the Directorate General of Resources for Science Technology and Higher Education (DG-RSTHE) Ministry of Research Technology and Higher Education of The Republic of Indonesia, Mulawarman University, Hasanuddin University who provided me an opportunity and monetary support to pursue doctoral study in Japan, and JSPS KAKENSHI Grant Number 16K00099 for researches and publications funding.

My special thanks to Engineering staff, International Office staff, and General Information Media Center staff for all supports and assistances.

I also thank my fellow lab mates especially for Nakano San, Suzuki San and Yamauchi San in for the patience, supports and assistances as members of Software Engineering Lab. And PPI Commissariat Ehime in for the supports and assistances as fellow Indonesian students.

Last but not the least, I would like to thank all family members: my father “Aji Burhanuddin”, my mother “Suwanti”, brother “Aji Rawindra Handanny, sisters “Aji Fika Trisnawaty, and Aji Ayunita Kristiningrum”, my lovely wife “Nimas Ayu Dewi Murni” and daughters “Aji Ayyashi Dzakiyyah Nabila & Aji Reyna Dzikra Nuhaa” for supporting me spiritually throughout writing this thesis and my life in general.

List of Publications

Journal Papers

[1] Aji Ery Burhandenny, Hirohisa Aman and Minoru Kawahara, "Change-Prone Java Method Prediction by Focusing on Individual Differences in Comment Density," IEICE Transactions on Information and Systems, vol.E100-D, no.5, pp.1128–1131, May 2017.

[2] Aji Ery Burhandenny, Hirohisa Aman and Minoru Kawahara, "An Evaluation of Coding Violation Focusing on Change History and Authorship of Source File," International Journal of Networked and Distributed Computing, vol.5, no.4, pp.211–220, Oct. 2017.

Conference Papers

[3] Aji Ery Burhandenny, Takashi Nakano, Hirohisa Aman and Minoru Kawahara, "Empirical Study of Change-Prone and Fault-Prone Method Prediction Focusing on Comment Ownership," in Proceedings of the International Conference on Business and Information, vol.13, no.2, pp.219–230, July 2016.

[4] Aji Ery Burhandenny, Hirohisa Aman and Minoru Kawahara, "Examination of Coding Violations Focusing on Their Change Patterns over Releases," in Proceedings of the 23rd Asia-Pacific Software Engineering Conference, pp.121–128, Dec. 2016.

[5] Aji Ery Burhandenny, Hirohisa Aman and Minoru Kawahara, "Investigation of Coding Violations Focusing on Authorships of Source Files," in Proceedings of the 5th International Conference on Applied Computing and Information Technology/4th International Conference on Computational Science/Intelligence and Applied Informatics/2nd International Conference on Big Data, Cloud Computing, Data Science & Engineering, pp.254–259, July 2017.

Contents

1 Introduction **1**
1.1 Overview 1
1.2 Outline of Dissertation 2
1.2.1 Comments in Program (Chapter 2) 2
1.2.2 General Trend of Coding Violation in Program (Chapter 3) 2
1.2.3 Individual Difference on Violation (Chapter 4) 3
1.2.4 Coverage and Importance of Coding Violation (Chapter 5) 3
1.2.5 Conclusion (Chapter 6) 4

2 Comments in Program **5**
2.1 Comment Density Considering Individual 6
2.1.1 Cyclomatic Complexity 6
2.1.2 Comment Density 8
2.2 Empirical Study 14
2.2.1 Aim and Dataset 14
2.2.2 Assessment Criterion 15
2.2.3 Results 19
2.2.4 Discussions 23
2.2.5 Threats to Validity 23
2.3 Conclusion 24

3 General Trend of Coding Violation **25**
3.1 Static Testing and Code Checker 26
3.2 Empirical Work 30
3.2.1 Aim and Studied Projects 30
3.2.2 Data Collection 31
3.2.3 Analysis 1: Distribution of Violations Sorted by Predefined Priority 32
3.2.4 Analysis 2: Trend of Each Violation 37

3.2.5 Threats to Validity 49
3.3 Related Work 49
3.4 Conclusion 50

4 Individual Difference on Violation **53**
4.1 Violation and Authorship of Source File 54
4.1.1 Evaluation of Violation 54
4.1.2 Impact of Authorship on Evaluation 55
4.2 Empirical Study 57
4.2.1 Aim and Dataset 57
4.2.2 Data Collection 57
4.2.3 Analysis 1 (for RQ1): Comparison of Violations Appearing in Single-Authored Files vs. Multi-Authored Files 58
4.2.4 Analysis 2 (for RQ2): Comparison of Violations across Projects 65
4.2.5 Threats to Validity 68
4.3 Related Work 69
4.4 Conclusion and Future Work 69

5 Coverage and Importance of Coding Violation **71**
5.1 Coding Standard Violation 72
5.1.1 Change of Coding Standard Violations through Commits 72
5.1.2 Research Questions 75
5.1.3 Metrics 76
5.2 Empirical Analysis 78
5.2.1 Aim and Data Collection 78
5.2.2 Results 79
5.2.3 Discussions 82
5.2.4 Threats to Validity 82
5.3 Related Work 83
5.4 Conclusion 83

6 Conclusion **85**

Chapter 1

Introduction

1.1 Overview

Computer systems have been widely used in our social life, so their quality managements are crucial activities today. While computers accurately work in accordance with the software, software themselves are usually designed and developed by human beings. That is to say, human factors indirectly have large impacts on the successful work of software-based computer systems.

Within last decade, the importance of human role in software development has gained more attention in the software engineering research area. Some researchers have exploited its impact on the productivity by using the psychology perspective [1],[2], and organizational perspective [3],[4]. Furthermore, many studies have been performed in order to reveal potential links among technical properties of software measurements and human aspects that can be identified through developmental trends and communications [5],[6],[7],[8],[9], [10],[11],[12],[13],[14],[15],[16],[17],[18],[19],[20],[21].

Although previous research efforts demonstrated positive signs toward the connectivity between human aspects and software development in general, they have not deeply taken into account programmers' habits while conducting programming activities. Despite the advancement of artificial intelligent, programming activity is mainly conducted by human beings. Therefore, programmer's personality is strongly related to his/her programming styles. Hence, it would heavily influence the result of programming activities. This dissertation attempts to fill the above gap by investigating and reporting the results through empirical studies.

This dissertation focuses on two properties: "comment" and "coding violation" as main parameters to investigate the traits of programmer's habit toward programming activities. Both properties are expected to be inter-

related and totally dependent on individual programmer's preference.

Comments as a mean to enhance code readability, and they produce a good indicator for identifying which code is more error-prone. This point of view will be discussed in Chapter 2. The coding violation as a by-product of programming activity can be assessed by utilizing a static code checker. While coding violations are warnings that the corresponding code fragments are not compliant with the predefined coding rules, these violations also reflect the programmer's preference and habit. That is to say, they can be useful clues to analyze the impacts of human factors on the code quality. Coding violation-related empirical studies will be presented in Chapters 3, 4, and 5.

1.2 Outline of Dissertation

This dissertation consist of 4 research stages that have their own objectives, methodologies and results.

1.2.1 Comments in Program (Chapter 2)

Chapter 2 presents our study focusing on the comments in source programs. Objective: To evaluate the abnormality of a method from the perspective of the comment density with consideration for differences among individual programmers, and to empirically examine if such an abnormality can help in a change-prone method prediction.

Methodology: We quantify the abnormality of comment density based on the data distribution of individual programmer. Then, we introduce "the adjusted cyclomatic complexity" (ACC) metric as an improved version of the traditional cyclomatic complexity (CC) metric by using the above abnormality. As an assessment criterion in comparison, we computed the area under the curve (AUC) of the receiver operating characteristic (ROC) curve.

Results: The AUC values showed that ACC tends to be better than CC, so it is worth to focus on the individual abnormality of comment density.

1.2.2 General Trend of Coding Violation in Program (Chapter 3)

Chapter 3 presents an empirical analysis of changes in coding violations over releases.

Objective: To examine coding violations that are related to the parts that many programmers tend to improve or are likely to be disregarded.

Methodology: We perform code analysis using a static code checker, PMD, and examine the trend of violation changes over release versions. We introduce "the index of programmers' attention"(IPA) as an assessment criterion to distinguish important violations from disregarded ones.

Results: We analyzed violations appearing in 7 open source software (OSS) projects, and showed important violations and disregarded ones in each project.

1.2.3 Individual Difference on Violation (Chapter 4)

Chapter 4 gives a further analysis of coding violations by focusing on the differences among individual programmers.

Objective: To examine the influence of authoring types toward coding violations.

Methodology: We perform in-depth analysis on single-authored files and multi-authored files throughout releases. The we introduce "the normalized IPA (NIPA)" as IPA extension and analyze correlations and similarities of important/disregarded violation sets.

Results: We analyzed the same 7 OSS projects with Chapter 3, and presented the differences of violation trends between "single-authored file" and "multi-authored file": the authorship can be a noteworthy factor to evaluate the importance of violations.

1.2.4 Coverage and Importance of Coding Violation (Chapter 5)

Chapter 5 presents two perspectives other than Chapter 3 and 4: "coverage" and "bug-proneness."

Objective: To analyze the importance of frequently-appearing coding violations from the perspective of bug-proneness.

Methodology: We perform in-depth analysis on code change histories at commit level which take into account individual programmer preferences. We introduce "file coverage (FC)","developer coverage (DC)" and "the importance of violation (IMP)" to measure the familiarity and importance of violation.

Results: We analyzed 6 popular OSS projects, and performed data categorization using the above metrics. The results showed that set of familiar and bug-related violations tends to be vary from project to project. But we found 25 frequently-appearing violations common to all projects.

1.2.5 Conclusion (Chapter 6)

The final chapter ties together all research stages and findings into a conclusion.

Chapter 2

Comments in Program

Comments are both well-known artifacts for enhancing the readability of programs and useful embedded documents [22]. They can provide various information including the copyright designation, the programmers' memos on the code fragments and the manuals of their functions [23]. While comments are independent of program executions, we cannot ignore the impact of comments on the code quality due to their helpful effects on the program comprehension. However, there have also been concerns that some programmers might add detailed comments to their complicated code in order to compensate for a lack of readability [24]; In the code refactoring world, well-written comments are said to be "deodorant" for masking code smells [25]. There are empirical reports showing that the comments within a Java method body—hereinafter referred to as "inner comments"—are noteworthy artifacts in analyzing the code quality, and more-commented methods are more likely to be fixed after their releases (i.e., change-prone) [26, 27, 28].

However, a programmer's preference may also play an important role in commenting source code. While one programmer prefers to write detailed comments, another programmer does not like to write comments at all. If the latter programmer wrote a well-commented method, it would be an abnormal case and we should review the method preferentially. That is to say, such an abnormality depends on who wrote the program. The aim of this chapter is to evaluate the abnormality of a method from the perspective of the comment density with consideration for differences among individual programmers, and to empirically examine if such an abnormality can help in a change-prone method prediction.

2.1 Comment Density Considering Individual Difference and Its Application to Program Complexity Metric

In general, more complex methods tend to be harder to understand, and may have more inner comments than simpler ones. To evaluate the amount of comments regardless of the complexity, we focus on the comment density, the lines of inner comments normalized by the complexity. In this chapter, we will use the cyclomatic complexity [29], which has been a well-known complexity metric, as our complexity criterion.

2.1.1 Cyclomatic Complexity

The cyclomatic complexity (CC) was proposed by McCabe. Its purpose is to identify the complexity of a program by the number of linearly independent paths in the program's control flow graph. For example, if a program does not have any decision point such as While/FOR Loops or IF Statements, its CC value is 1 because there is only one execution path through the code. The use of decision point would added the execution paths and hence increases the complexity of the code.

Mathematically, the complexity of a method m, $\mathrm{CC}(m)$, is defined as

$$\mathrm{CC}(m) = E(m) - N(m) + 2P(m) \ , \tag{2.1}$$

where $E(m)$ is the number of edges of the graph, $N(m)$ is the number of nodes of the graph, and $P(m)$ is the number of connected components in m, respectively.

For a single program or method, $P(m)$ is always equal to 1. Therefore the equation for a single method is:

$$\mathrm{CC}(m) = E(m) - N(m) + 2 \ . \tag{2.2}$$

For example, Fig. 2.1 is a Java method. The control flow graph is shown in Fig. 2.2.

```
       public Map<String, Integer> getSummary()
(01)     Map<String, Integer> summary = new HashMap<>();
(02)        for (RuleViolation rv : violations) {
(03)            String name = rv.getRule().getName();
(03)            if (!summary.containsKey(name)) {
(04)                summary.put(name, NumericConstants.ZERO);
                 }
(05)                Integer count = summary.get(name);
(05)                summary.put(name, count + 1);
            }
(06)        return summary;
       }
```

Figure 2.1: Example of Java program.

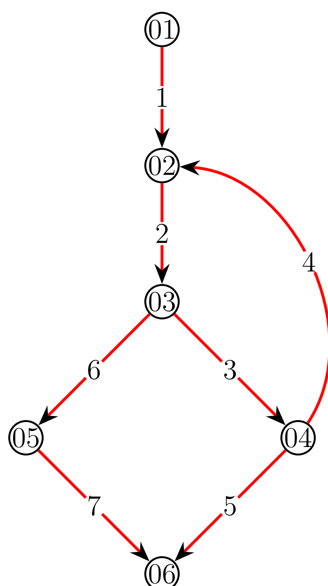


Figure 2.2: Control flow graph of the code shown in Fig.2.1.

As shown in Fig. 2.2, the example method has 7 edges (represented by arrows) and 6 nodes (represented by circles). Using Eq. (2.2), we can calculate the complexity as follows:

$$\begin{aligned} \text{CC}(m) &= E(m) - N(m) + 2 \\ &= 7 - 6 + 2 \\ &= 3 \; . \end{aligned}$$

Therefore, the cyclomatic complexity of the example method is 3.

However, a more complex code has a more complex flow graph which is sometime too expensive to do in terms of time and memory (the case using automation). Therefore, McCabe also provided an easily computation way of the cyclomatic complexity metric, using the following equation:

$$\mathrm{CC}(m) = D(m) + 1 \ , \tag{2.3}$$

where $D(m)$ is the number of decision point in the method m.

In our example case shown in Fig. 2.1, the method has 2 decision points which are 1 FOR-LOOP statement and 1 IF statement. Hence, using Eq. (2.3), the cyclomatic complexity can be calculated as follows:

$$\begin{aligned} \mathrm{CC}(m) &= D(m) + 1 \\ &= 2 + 1 \\ &= 3 \ . \end{aligned}$$

The result is same with the previous equation. In general, an acceptable upper bound of the cyclomatic complexity is said to be 10. Thus, the above example method seems to have a normal or lower complexity.

2.1.2 Comment Density

In order to consider individual differences in commenting code, we need to make a link between a method and its programmer. In general, two or more programmers may be involved in a method development through its maintenance. Thus, we focus on the initial version of method because we can always determine a specific programmer for each method; if we focus on later versions of methods which two or more developers have made modifications, it is hard to evaluate the amount of comments for each developer. A further analysis taking care of multi-developer methods is our significant future work.

Now, let N be the number of programmers, and denote them by p_i (for $i = 1, \ldots, N$). Suppose p_i has developed M_i methods, m_{ij} (for $j = 1, \ldots, M_i$). Then, we define the comment density of method m_{ij} "$\mathrm{CD}(m_{ij})$" as follows:

$$\mathrm{CD}(m_{ij}) := \log \left\{ \frac{\mathrm{LCM}(m_{ij})}{\mathrm{CC}(m_{ij})} + 1 \right\} \tag{2.4}$$

where $\mathrm{LCM}(m_{ij})$ and $\mathrm{CC}(m_{ij})$ are the lines of inner comments of m_{ij} and the complexity of m_{ij}, respectively. While a comment density can also be obtained by the simple ratio "$\mathrm{LCM}(m_{ij})/\mathrm{CC}(m_{ij})$," the distribution of such

values is likely to be right-skewed, so we will use a logarithmically transformed form shown in Eq.(2.4)[1].

Based on $\mathrm{CD}(m_{ij})$, we define the comment density considering individual difference, "$\mathrm{zCD}(m_{ij})$," as follows:

$$\mathrm{zCD}(m_{ij}) = \frac{\mathrm{CD}(m_{ij}) - \mu_i}{\sigma_i} \tag{2.5}$$

where μ_i and σ_i are the mean and the standard deviation of programmer p_i's comment densities $\mathrm{CD}(m_{ij})$ (for $j = 1, \ldots, M_i$), respectively.

$\mathrm{zCD}(m_{ij})$ is a standard score of $\mathrm{CD}(m_{ij})$, which is referred to as "z-score." A larger value of $\mathrm{zCD}(m_{ij})$ shows that m_{ij} has a higher comment density than programmer p_i's usual work. Since its scale is normalized by the dispersion (σ_i), a high zCD value signifies that the comment density is abnormally high for the programmer. That is to say, zCD can be a metric for measuring an abnormality of method from the perspective of the comment density.

Now we make a hypothesis that zCD is useful in finding problematic methods. Needless to say, the conventional complexity metric, CC, is a promising metric for evaluating programs. Then, we consider an enhancement of CC by using zCD, and define the following Adjusted CC, "ACC":

$$\mathrm{ACC}(m_{ij}) = \phi(m_{ij}) \cdot \mathrm{CC}(m_{ij}) \tag{2.6}$$

where $\phi(m_{ij})$ is the degree of attention to m_{ij}. In order to calculate $\phi(m_{ij})$ based on the abnormality of m_{ij}, we propose to use the cumulative normal distribution function as follows:

$$\begin{aligned} \phi(m_{ij}) &= \Pr\{\, x \leq \mathrm{zCD}(m_{ij}) \,\} \\ &= \int_{-\infty}^{\mathrm{zCD}(m_{ij})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \quad . \end{aligned} \tag{2.7}$$

The range of $\phi(m_{ij})$ is $0 < \phi(m_{ij}) < 1$. When a method has an average comment density, $\phi(m_{ij})$ is around 0.5. When $\mathrm{zCD}(m_{ij})$ is higher—an abnormally-high comment density—, $\phi(m_{ij})$ gets closer to 1; when $\mathrm{zCD}(m_{ij})$ is lower, $\phi(m_{ij})$ approaches 0. Figure 2.3 presents the relationship between $\mathrm{zCD}(m_{ij})$ and $\phi(m_{ij})$.

[1] Since $\mathrm{LCM}(m_{ij})$ can be zero, we inserted "+1" into the equation: if $\mathrm{LCM}(m_{ij}) = 0$, then $\mathrm{CD}(m_{ij}) = \log(0 + 1) = 0$.

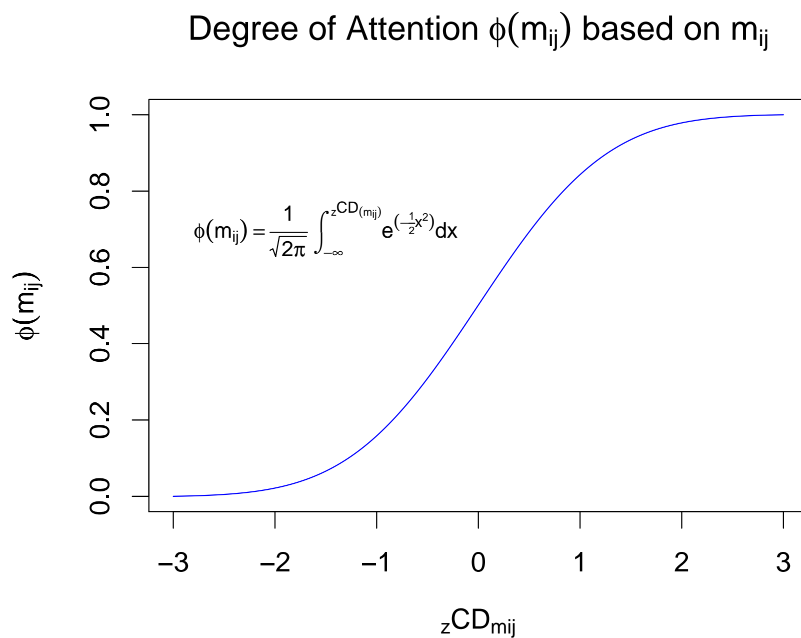


Figure 2.3: Degree of attention $\phi(m_{ij})$ based on the abnormality of m_{ij}.

Let us consider an example of a programmer p_i who has maintained 7 methods m_{ij} (for $j = 1, \ldots, 7$). Table 2.1 shows the metrics values.

Table 2.1: Metrics values of methods m_{ij} (for $j = 1, \ldots, 7$).

Method Name	CC	LCM
m_{i1}	1	0
m_{i2}	2	1
m_{i3}	10	6
m_{i4}	1	0
m_{i5}	1	0
m_{i6}	1	0
m_{i7}	5	1

Using Eq.(2.4), the comment density (CD) of each method is computed as follows:

$$\mathrm{CD}(m_{i1}) = \log\left\{\frac{0}{1}+1\right\} = 0,$$
$$\mathrm{CD}(m_{i2}) = \log\left\{\frac{1}{2}+1\right\} = 0.4054651,$$
$$\mathrm{CD}(m_{i3}) = \log\left\{\frac{6}{10}+1\right\} = 0.4700036,$$
$$\mathrm{CD}(m_{i4}) = \log\left\{\frac{0}{1}+1\right\} = 0,$$
$$\mathrm{CD}(m_{i5}) = \log\left\{\frac{0}{1}+1\right\} = 0,$$
$$\mathrm{CD}(m_{i6}) = \log\left\{\frac{0}{1}+1\right\} = 0,$$
$$\mathrm{CD}(m_{i7}) = \log\left\{\frac{5}{1}+1\right\} = 0.1823216.$$

Suppose $\mu = 0.1511129$ and $\sigma = 0.2076458$, then the standard score of comment density (zCD) is computed based on Eq.(2.5) as follows :

$$\mathrm{zCD}(m_{i1}) = \frac{0 - 0.1511129}{0.2076458} = -0.7277435,$$
$$\mathrm{zCD}(m_{i2}) = \frac{0.4054651 - 0.1511129}{0.2076458} = 1.2249329,$$
$$\mathrm{zCD}(m_{i3}) = \frac{0.4700036 - 0.1511129}{0.2076458} = 1.5357434,$$
$$\mathrm{zCD}(m_{i4}) = \frac{0.4700036 - 0.1511129}{0.2076458} = 1.5357434,$$
$$\mathrm{zCD}(m_{i5}) = \frac{0.4700036 - 0.1511129}{0.2076458} = 1.5357434,$$
$$zCD(m_{i6}) = \frac{0.4700036 - 0.1511129}{0.2076458} = 1.5357434,$$
$$zCD(m_{i7}) = \frac{0.1823216 - 0.1511129}{0.2076458} = 0.1502975.$$

In order to find ACC values, we need to compute $\phi(m_{i1})$ to $\phi(m_{i7})$ using Eq.(2.7) then calculate ACC according to Eq.(2.6) as follows:

$$\begin{aligned}\phi(m_{i1}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i1})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= \int_{-\infty}^{-0.7277435} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= 0.2333853,\end{aligned}$$

$$\begin{aligned}\mathrm{ACC}(m_{i1}) &= \phi(m_{i1}) \cdot \mathrm{CC}(m_{i1}) \\ &= 0.2333853 \cdot 1 \\ &= 0.2333853.\end{aligned}$$

$$\begin{aligned}\phi(m_{i2}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i2})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= \int_{-\infty}^{1.2249329} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= 0.8896997,\end{aligned}$$

$$\begin{aligned}\mathrm{ACC}(m_{i2}) &= \phi(m_{i2}) \cdot \mathrm{CC}(m_{i2}) \\ &= 0.8896997 \cdot 2 \\ &= 1.7793995.\end{aligned}$$

$$\begin{aligned}\phi(m_{i3}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i3})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= \int_{-\infty}^{1.5357434} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\ &= 0.9376993,\end{aligned}$$

$$\begin{aligned}\mathrm{ACC}(m_{i3}) &= \phi(m_{i3}) \cdot \mathrm{CC}(m_{i3}) \\ &= 0.9376993 \cdot 10 \\ &= 9.3769934.\end{aligned}$$

$$
\begin{aligned}
\phi(m_{i4}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i4})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= \int_{-\infty}^{-0.7277435} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= 0.2333853,
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ACC}(m_{i4}) &= \phi(m_{i4}) \cdot \mathrm{CC}(m_{i4}) \\
&= 0.2333853 \cdot 1 \\
&= 0.2333853.
\end{aligned}
$$

$$
\begin{aligned}
\phi(m_{i5}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i5})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= \int_{-\infty}^{-0.7277435} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= 0.2333853,
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ACC}(m_{i5}) &= \phi(m_{i5}) \cdot \mathrm{CC}(m_{i5}) \\
&= 0.2333853 \cdot 1 \\
&= 0.2333853.
\end{aligned}
$$

$$
\begin{aligned}
\phi(m_{i6}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i6})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= \int_{-\infty}^{-0.7277435} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= 0.2333853,
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ACC}(m_{i6}) &= \phi(m_{i6}) \cdot \mathrm{CC}(m_{i6}) \\
&= 0.2333853 \cdot 1 \\
&= 0.2333853.
\end{aligned}
$$

$$
\begin{aligned}
\phi(m_{i7}) &= \int_{-\infty}^{\mathrm{zCD}(m_{i7})} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= \int_{-\infty}^{0.1502975} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx \\
&= 0.5597351,
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ACC}(m_{i7}) &= \phi(m_{i7}) \cdot \mathrm{CC}(m_{i7}) \\
&= 0.5597351 \cdot 5 \\
&= 2.7986753.
\end{aligned}
$$

The example shown that m_{i7} is having an average comment density. m_{i1}, m_{i4}, m_{i5} and m_{i6} are having a low comment density. m_{i2} and m_{i3} are affected to high abnormality of comment density of programmer p_i.

2.2 Empirical Study

2.2.1 Aim and Dataset

The aim of this empirical study is to quantitatively examine if ACC can be useful for detecting change-prone Java methods. Since change-prone methods cannot survive unscathed after their releases, they should be reviewed more carefully.

We collected methods from nine popular open source software (OSS) projects shown in Table 2.2. The main reasons why we selected those nine projects as our data source are: (a) they are popular OSS projects, (b) their source files have been maintained with the Git, and (c) they are written in Java.

For the reason (a), we believe that it is better to use popular projects; results derived from minor projects would be worthless. All of these nine projects are ranked in the top 50 popular Java projects at SourceForge.net[2]. The reason (b) is for an ease of data collection. The Git provides powerful functions to collect data including source code and their changes. The reason (c) is from our data collection tool[3].

[2]https://sourceforge.net/

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

Table 2.2: Surveyed OSS projects.

project name	#examined methods
Angry IP Scanner [30]	1,669
Eclipse Checkstyle Plugin [31]	1,511
eXo Platform [32]	1,362
FreeMind [33]	6,920
GNU ARM Eclipse Plug-ins [34]	3,813
Hibernate ORM [35]	34,395
PMD [36]	2,510
ProjectLibre [37]	30,090
SQuirreL SQL Client [38]	20,946
total	103,246

For each source file, we traced all change logs and found the initial version of each method. Then, for each method, we collected the following data: (1) CC value, (2) LCM value, (3) the author's name and e-mail address, and (4) the number of changes that occurred after the release. We compute CD value for each method with Eq.(2.4), and get the mean (μ_i) and the standard deviation (σ_i) of the CD values for each programmer. Then, we compute zCD value for each method with Eq.(2.5). Finally, we obtain ACC values with Eqs.(2.6),(2.7).

There may be an author who has two or more different names or e-mail addresses on the repository. We tried integrating duplicated author data by the following set of rules—it is a simpler version of the algorithm proposed by Bird et al. [39]: (1) if two authors have different addresses but the same name, then we regard them as the same author; (2) if two authors have the same address but different names, then we regard them as the same author.

2.2.2 Assessment Criterion

To evaluate the performance of change-prone method prediction using ACC, we leverage the receiver operating characteristic (ROC) curve [40]. When a method's ACC value is greater than an established threshold, we judge the method is change-prone. Then, we can see the true-positive rate (TPR) and the false-positive rate (FPR). The ROC curve shows the relationships between FPR and FPR for all available thresholds. The area under the curve (AUC) is a well-known criterion of the performance—a higher AUC value signifies a better performance in discriminating change-prone methods. We will compare the AUC values of ACC (the proposed metric) and the ones of CC (the conventional metric).

In order to obtain the ROC curve, it requires two classifiers: the TPR and FPR which can be derived from the 2x2 contingency matrix as illustrated in Tabel 2.3. The TPR is the ratio of the number of correctly predicted positive cases to all truly positive cases. The FPR states the ratio of the number of incorrectly predicted positive cases to all negative cases. The best prediction would resulted in coordinate $(0, 1)$ which reflects the 100% of no false negative and 100% of no false positive. The diagonal line which also called as the line of no-discrimination divides the ROC space. Any points above diagonal represent good predictors (provide better accuracy than random); any points bellow the diagonal represent poor predictors.

Table 2.3: The contingency table

<table>
<tr><th colspan="2" rowspan="2">Total Population</th><th colspan="2">Actual Condition</th></tr>
<tr><th>Condition Positive</th><th>Condition Negative</th></tr>
<tr><td rowspan="2">Predicted Condition</td><td>Predicted Condition Positive</td><td>TRUE POSITIVE</td><td>FALSE POSITIVE</td></tr>
<tr><td>Predicted Condition Negative</td><td>FALSE NEGATIVE</td><td>TRUE NEGATIVE</td></tr>
<tr><td></td><td></td><td>$\sum$ Condition Positive</td><td>$\sum$ Condition Negative</td></tr>
</table>

$$\text{True Positive Rate (TPR)} = \frac{\text{True Positive}}{\sum \text{Condition Positive}}, \quad (2.8)$$

$$\text{False Positive Rate (FPR)} = \frac{\text{False Positive}}{\sum \text{Condition Negative}}. \quad (2.9)$$

For example, let us consider 12 methods whose ACC values and change-proneness are shown in Table 2.4. Symbol T refers to a change-prone method and F does not.

Table 2.4: ACC and change-proneness methods.

ACC	10	9	9	8	7	7	5	4	3	2	2	1
Change Prone	T	T	F	F	T	F	T	F	F	F	T	F

Let τ be the threshold of ACC value for the change-prone method prediction: if a method's ACC value is greater than or equal to τ (ACC $\geq \tau$), we predict that the method is change-prone (positive); otherwise, we predict it as not change-prone (negative).

Table 2.5 shows the results when $\tau = 5$.

Table 2.5: The contingency table for $\tau = 5$

		Actual	
		T	F
Predicted	T	4	3
	F	1	4

As shown in Table 2.5, we have 4 True Positive (TP) cases, 1 False Positive (FP) one, 3 False Negative (FN) ones, and 4 True Negative (TN) ones when $\tau = 5$.

Similarly, Table 2.6 presents the contingency table when $\tau = 8$, and we have 2 TP cases, 3 FN ones, 2 FP ones, and 5 TN ones.

Table 2.6: The contingency table for $\tau = 8$

		Actual	
		T	F
Predicted	T	2	2
	F	3	5

Thus we are able to calculate the FPR and TPR which are enabled us to plot the ROC curve, using the Eqs. (2.8) and (2.9):
For $\tau = 5$:

$$\begin{aligned} \text{TPR} &= \frac{\text{True Positive}}{\sum \text{Condition Positive}} \\ &= \frac{4}{7} \\ &= 0.57, \end{aligned}$$

$$\begin{aligned} \text{FPR} &= \frac{\text{False Positive}}{\sum \text{Condition Negative}} \\ &= \frac{1}{5} \\ &= 0.20. \end{aligned}$$

For $\tau = 8$:

$$\begin{aligned} \text{TPR} &= \frac{\text{True Positive}}{\sum \text{Condition Positive}} \\ &= \frac{5}{7} \\ &= 0.71, \end{aligned}$$

$$
\begin{aligned}
\text{FPR} &= \frac{\text{False Positive}}{\sum \text{Condition Negative}} \\
&= \frac{3}{5} \\
&= 0.60.
\end{aligned}
$$

We obtain the ROC curve shown in Fig. 2.4 by changing τ from 1 to 10.

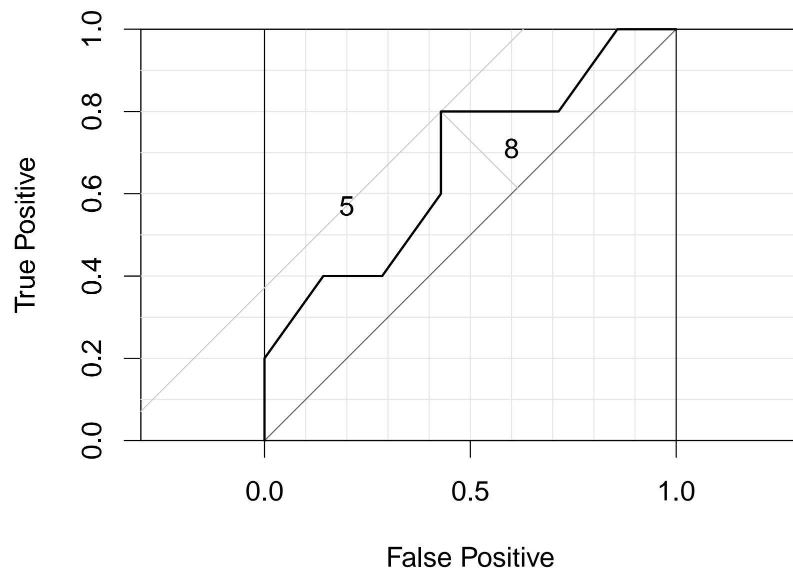


Figure 2.4: ROC curves

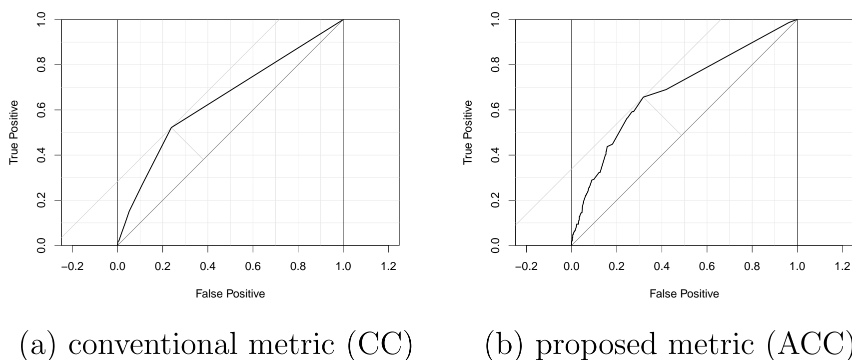
Since the number of code changes that occurred in most of methods is less than two (see Table 2.7), we consider the methods which had been modified two or more times after their release, to be change-prone methods in this chapter.

Table 2.7: Distribution of code change counts in methods.

project name	min	25%	50%	75%	max
Angry IP Scanner	0	0	0	1	45
Eclipse Checkstyle Plugin	0	0	0	1	7
eXo Platform	0	0	0	1	14
FreeMind	0	0	1	1	61
GNU ARM Eclipse Plug-ins	0	0	0	1	33
Hibernate ORM	0	0	0	1	47
PMD	0	0	0	1	13
ProjectLibre	0	0	0	0	7
SQuirreL SQL Client	0	0	0	0	21

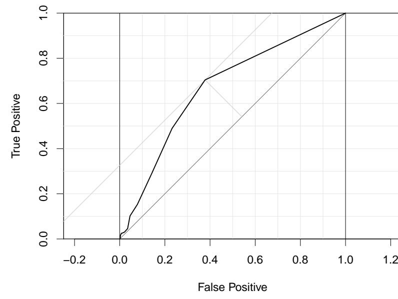
2.2.3 Results

Figure 2.5 shows ROC curves for the Angry IP Scanner: (a) and (b) are the ROC curves obtained by using the conventional metric (CC) and the proposed metric (ACC), respectively. Similarly, Figs. 2.6–2.13 presents their ROC curves.

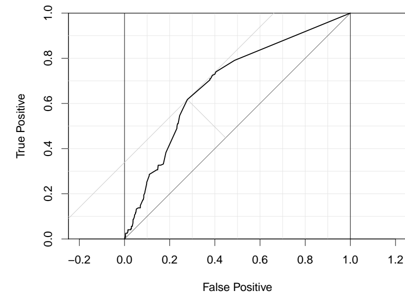


(a) conventional metric (CC)　　(b) proposed metric (ACC)

Figure 2.5: ROC curves for “Angry IP Scanner.”

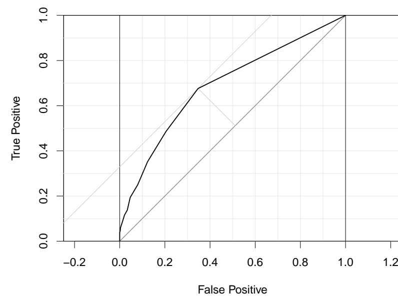


(a) conventional metric (CC)

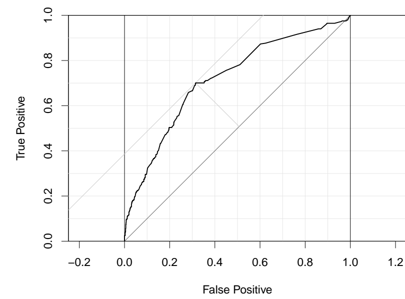


(b) proposed metric (ACC)

Figure 2.6: ROC curves for "Eclipse Checkstyle Plugin."

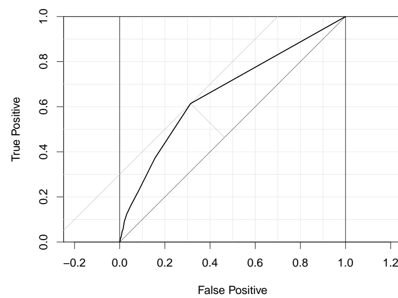


(a) conventional metric (CC)

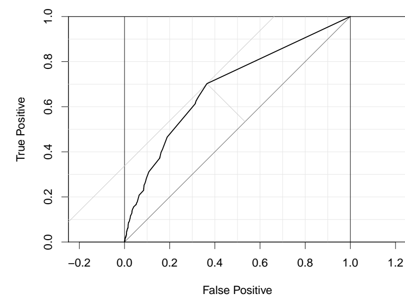


(b) proposed metric (ACC)

Figure 2.7: ROC curves for "eXo Platform."

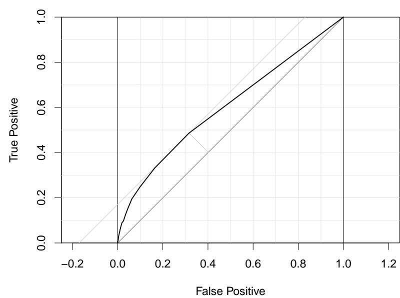


(a) conventional metric (CC)

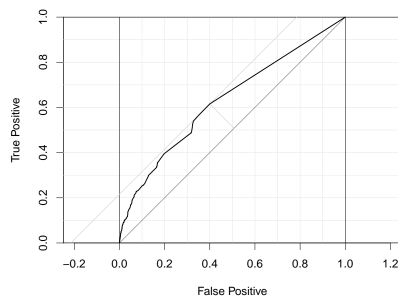


(b) proposed metric (ACC)

Figure 2.8: ROC curves for "FreeMind."

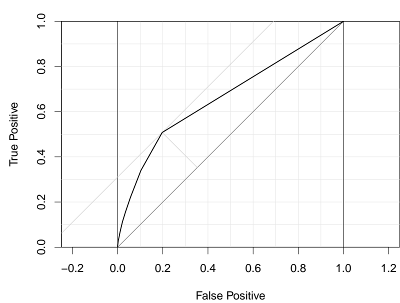


(a) conventional metric (CC)

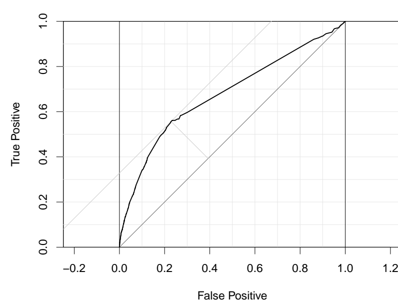


(b) proposed metric (ACC)

Figure 2.9: ROC curves for “GNU ARM Eclipse Plug-ins.”

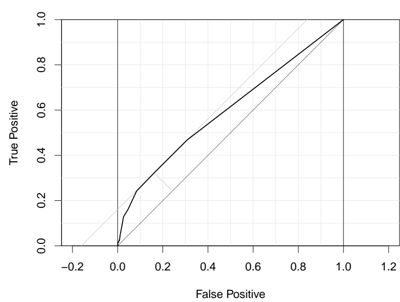


(a) conventional metric (CC)

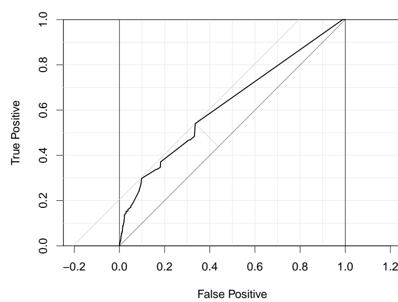


(b) proposed metric (ACC)

Figure 2.10: ROC curves for “Hibernate ORM.”

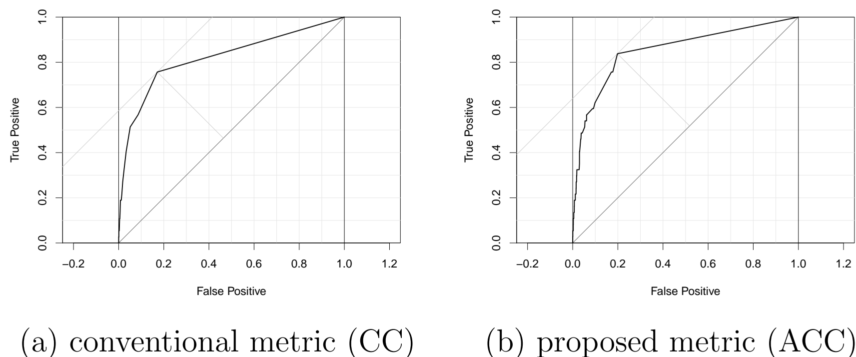


(a) conventional metric (CC)



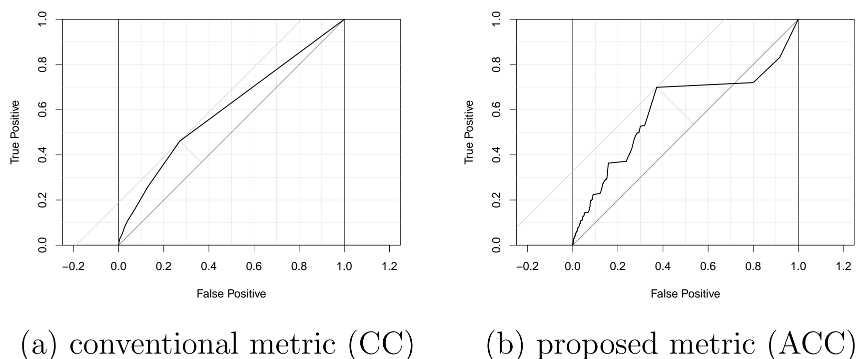
(b) proposed metric (ACC)

Figure 2.11: ROC curves for “PMD.”



(a) conventional metric (CC) (b) proposed metric (ACC)

Figure 2.12: ROC curves for “ProjectLibre.”



(a) conventional metric (CC) (b) proposed metric (ACC)

Figure 2.13: ROC curves for “SQuirreL SQL Client.”

Table 2.8 presents the AUC values for all projects. δ in the table denotes the rate of improvement by ACC instead of CC, defined as follows:

$$\delta = 100 \cdot \frac{\alpha(\text{ACC}) - \alpha(\text{CC})}{\alpha(\text{CC})} \quad (\%), \tag{2.10}$$

where $\alpha(\text{CC})$ and $\alpha(\text{ACC})$ are the AUC values of ROC curves obtained by using CC and ACC, respectively.

Table 2.8: AUC values: CC vs. ACC.

project name	AUC (α)		δ
	CC	ACC	
Angry IP Scanner	0.647	0.685	5.922
Eclipse Checkstyle Plugin	0.672	0.692	2.905
eXo Platform	0.690	0.718	4.068
FreeMind	0.665	0.691	3.940
GNU ARM Eclipse Plug-ins	0.604	0.632	4.549
Hibernate ORM	0.665	0.678	1.918
PMD	0.600	0.624	3.996
ProjectLibre	0.819	0.852	4.010
SQuirreL SQL Client	0.602	0.598	−0.700
average	—	—	3.401

2.2.4 Discussions

All projects except for the SQuirreL SQL Client show $\delta > 0$ (see Table 2.8), which means the proposed metric performs better than the conventional one in terms of change-prone method prediction. Although the SQuirreL gives a negative rate, it is almost zero (−0.7%), so the proposed metric is at almost the same level of performance as the conventional metric for the SQuirreL. In total, the average value of δ is 3.4%. While the improvement made by using ACC is not especial high, it showed better performances for eight out of nine sampled projects. Therefore, an abnormality of comment density seems to be worthy of attention.

As an ACC value is a CC value multiplied by an attention degree which is in the range between 0 and 1, it is highly unlikely that an ACC value of a method differs drastically from the CC value of the method. This would be one of the major reasons why the metrics ACC and CC show similar performances. Nonetheless, the actual results show that ACC performs better than CC for eight out of nine projects. Since the multiplication of the attention degree can change the order of methods in their metric values, the adjustment seems to work successfully in detecting change-prone methods. That is to say, the abnormality of comment density by programmer would be a useful factor for enhancing the change-prone method prediction.

2.2.5 Threats to Validity

Our results may be subject to the following concerns.

We investigated nine OSS projects and the number of subjects might be

considered small, so there is a threat of generalization. However, since the our analysis unit is a "method" and not project, the above threat would be mitigated.

We analyzed OSS projects only written in Java. While programs written in another language might show different results, the notion of comment description is common to most of the modern programming languages. Thus, we believe the limitation of programming language cannot be a serious threat to validity in this chapter.

Our data of methods are of their "initial versions" but not the ones of a certain release version. Therefore, our results might not work well for predicting code changes after the release of interest. As we mentioned in Section 2.1, there is the issue of "multi-developer" of methods. In this chapter, we proposed to evaluate an abnormality of comments by focusing on each developer. In order to take into account a method which has been developed and maintained by two or more developers, we have to make another definition of abnormality in commenting, i.e., a multi-developer version of it. Since such multi-developer cases should be addressed as well, we plan to perform further analyses in order to produce a reasonable definition in the future.

2.3 Conclusion

We proposed to adjust the code complexity metric by using the abnormality of comment densities. Since there can be a wide variety in commenting code among programmers, the abnormality of comment densities was considered in this chapter. Through our empirical study with nine OSS projects, we showed that the proposed metric works better than the conventional complexity metric in predicting change-prone Java methods which cannot survive unscathed after their releases.

Chapter 3

General Trend of Coding Violation

A successful programming is essential for producing a high quality software product, and it is always important to manage the quality of source code during the programming activity. In order to check and enhance the code quality, code review is widely known as a useful activity. However, code review is also a costly and time-consuming activity since it is done by developers, i.e., human beings. Particularly in open source software (OSS) projects, since code review mostly relies on their contributors' spare time, it is critically needed to effectively perform code review at low cost. In the context, automated tools for detecting problematic parts of programs would be helpful in streamlining code review by the programmers themselves. Those tools are referred to as "static code analyzers" or "code checkers," and they usually point the parts which violate to a predefined coding convention or the ones which match to a predefined anti-coding pattern. Since such tools can find potential poor quality code fragments and/or latent bugs in the programs being developed, they are expected to be great helps for the code review and thus for the quality management of code.

However, such code checkers have not been actively utilized by programmers in reality [41]. In many cases, a lot of violations (warnings) are reported by a code checker and many of them are not actually important points (false positives) for programmers, then programmers tend to have hesitations in using such a tool. Thus, there have been studies for enhancing the accuracy of warnings (for reducing the false positive rate) in the past [5],[6],[10]. Although those studies would be useful in finding more latent bugs, their focuses were not on whether the programmers actually paid their attention to the parts warned by such a tool or not. In other words, there may be many code fragments which did not have bugs but had been modified by the

programmers through their upgrades. Thus, in this chapter, we examine the relationship between the coding violations and the programmers' attention from the perspective of the change patterns of violations over the releases.

3.1 Static Testing and Code Checker

In a broad sense, the software testing is categorized into two types of techniques, namely the dynamic testing and the static testing. The dynamic testing refers to as the activity that executes the program and examines whether it runs successfully or not. On the other hand, the static testing means a checking of the programs without its execution. It includes the code review and inspection. While both the dynamic testing and the static testing are essential work for improving the code quality, the latter one can be performed during the programming activity, i.e., earlier than the dynamic testing.

To help the static testing, there have been software tools called "static code analyzers" or "code checkers" such as PMD[1], Checkstyle[2] and FindBugs[3]. Those tools can detect the code fragments which violate the predefined coding conventions. Although a violation is not an error, it corresponds to the part which is recommended to be improved or refactored, so such tools are expected to support an efficient improvement of the code quality.

In this chapter, we will use PMD because of its ability to detect programming mistakes directly from a source code; while Findbugs requires the code to be compiled first and then check from byte code; Checkstyle can also check a source code directly, its key focus on the coding style. Furthermore, PMD rulesets can be utilized and customized easily.

Lets us consider an example of Java code shown in Fig. 3.1.

[1] http://pmd.github.io/
[2] http://checkstyle.sourceforge.net/
[3] http://findbugs.sourceforge.net/

```
(01) import java.util.Scanner;
(02) public class JavaExample {
(03)
(04)   public static void main(String[] args) {
(05)     System.out.println("How many numbers you want to enter?");
(06)     Scanner scanner = new Scanner(System.in);
(07)     int n = scanner.nextInt();
(08)     /* Declaring array of n elements, the value
(09)      * of n is provided by the user
(10)     */
(11)     double[] arr = new double[n];
(12)     double total = 0;
(13)
(14)     for(int i=0; i<arr.length; i++){
(15)       System.out.print("Enter Element No."+(i+1)+": ");
(16)       arr[i] = scanner.nextDouble();
(17)     }
(18)     scanner.close();
(19)     for(int i=0; i<arr.length; i++){
(20)       total = total + arr[i];
(21)     }
(22)
(23)     double average = total / arr.length;
(24)
(25)     System.out.format("The average is: %.3f", average);
(26)   }
(27) }
```

Figure 3.1: Example of Java program.

PMD has 224 predefined rulesets by default. By triggering these predefined rulesets, 12 warnings are produced by PMD for the code presented in Fig. 3.1. Table 3.1 shows the plain results; detailed warnings are given in XML format shown in Fig.3.2.

Table 3.1: Warning result produced by PMD.

#	File	Line	Problem
1	JavaExample.java	2	All classes and interfaces must belong to a named package
2	JavaExample.java	2	All methods are static. Consider using a utility class instead. Alternatively, you could add a private constructor or make the class abstract to silence this warning.
3	JavaExample.java	2	headerCommentRequirement Required
4	JavaExample.java	4	Parameter ‘args’ is not assigned and could be declared final
5	JavaExample.java	4	publicMethodCommentRequirement Required
6	JavaExample.java	5	System.out.println is used
7	JavaExample.java	6	Local variable ‘scanner’ could be declared final
8	JavaExample.java	7	Avoid variables with short names like n
9	JavaExample.java	7	Local variable ‘n’ could be declared final
10	JavaExample.java	12	Found ‘DD’-anomaly for variable ‘total’ (lines ‘12’-‘20’).
11	JavaExample.java	15	System.out.print is used
12	JavaExample.java	23	Local variable ‘average’ could be declared final

We explain these warnings below. There are 3 warnings detected on line 2. The first one is as the particular class does not have package definition. This

might not be a problem when the size of project is small and the programmer has chosen to use default package. However, this warning emphasizes a good programming practice to avoid future collision of class names. The second warning enforces the use of utility class if the class has static methods only, which provide an efficiency to reuse the methods. The third one is a simple suggestion to provide comments as one of good programming practices.

Line 4 has 2 warnings. The first one states the necessity to declare "final" for all method arguments that are never been reassigned within the method. The second one is to provide comments to improve the readability.

Line 5 and 15 tell us the use of *System.out.print* that is intended for debugging purposes and can be remain in the codebase even in production code. It maybe better to use a logger rather than a print method because we can switch it on and off.

Warnings in line 6, 7 and 23 are suggesting to declare these local variables as "final" because they are assigned only once.

Line 7 also has a warning to fix the variable name since a shorter name may adversely affect the code readability.

Lines from 12 to 20 experienced "$DD - Anomaly$" that states the redefinition of defined variable. This particular warning needs to be concerned since the initially assigned values are never read and can become a potential code smell.

In XML format(see Fig. 3.2), each warning(violation) provides a URL link to an external information Web page. Moreover, a priority of the corresponding violation is given, which is predefined. Although the priority information seems to be a help, it cannot be attractive in the real because a lot of "medium" leveled violations appear as we show later.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<pmd version=”5.8.1” timestamp=”2018-05-04T05:24:50.836”>
<file name=”JavaExample.java”>
<violation beginline=”2” endline=”27” begincolumn=”8” endcolumn=”1” rule=”NoPackage”
ruleset=”Naming” class=”JavaExample” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/naming.html#NoPackage” priority=”3”>All classes and interfaces must belong to
a named package</violation>
<violation beginline=”2” endline=”27” begincolumn=”26” endcolumn=”1” rule=”UseUtilityClass”
ruleset=”Design” class=”JavaExample” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/design.html#UseUtilityClass” priority=”3”>All methods are static. Consider using
a utility class instead. Alternatively, you could add a private constructor or make the class abstract
to silence this warning.</violation>
<violation beginline=”2” endline=”27” begincolumn=”8” end-
column=”1” rule=”CommentRequired” ruleset=”Comments”
class=”JavaExample” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/comments.html#CommentRequired” priority=”3”>headerCommentRequirement
Required </violation>
<violation beginline=”4” endline=”4” begincolumn=”29” endcolumn=”41”
rule=”MethodArgumentCouldBeFinal” ruleset=”Optimization” class=”JavaExample”
method=”main” variable=”args” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/optimizations.html #MethodArgumentCouldBeFinal” priority=”3”> Parameter
’args’ is not assigned and could be declared final </violation>
<violation beginline=”4” endline=”26” begincolumn=”19” endcolumn=”5”
rule=”CommentRequired” ruleset=”Comments” class=”JavaExample” method=”main”
externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/comments.html #Com-
mentRequired” priority=”3”> publicMethodCommentRequirement Required </violation>
<violation beginline=”5” endline=”5” begincolumn=”9” endcolumn=”26”
rule=”SystemPrintln” ruleset=”Java Logging” class=”JavaExample” method=”main”
externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/logging-java.html
#SystemPrintln” priority=”2”> System.out.println is used </violation>
<violation beginline=”6” endline=”6” begincolumn=”9” endcolumn=”48”
rule=”LocalVariableCouldBeFinal” ruleset=”Optimization” class=”JavaExample”
method=”main” variable=”scanner” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/optimizations.html #LocalVariableCouldBeFinal” priority=”3”> Local variable
’scanner’ could be declared final </violation>
<violation beginline=”7” endline=”7” begincolumn=”13” endcolumn=”13”
rule=”ShortVariable” ruleset=”Naming” class=”JavaExample” method=”main” variable=”n”
externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/naming.html #ShortVari-
able” priority=”3”> Avoid variables with short names like n </violation>
<violation beginline=”7” endline=”7” begincolumn=”9” endcolumn=”33”
rule=”LocalVariableCouldBeFinal” ruleset=”Optimization” class=”JavaExample”
method=”main” variable=”n” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/optimizations.html #LocalVariableCouldBeFinal” priority=”3”> Local variable ’n’
could be declared final </violation>
<violation beginline=”12” endline=”20” begincolumn=”17” endcolumn=”38”
rule=”DataflowAnomalyAnalysis” ruleset=”Controversial” class=”JavaExample”
method=”main” variable=”total” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/controversial.html #DataflowAnomalyAnalysis” priority=”5”> Found ’DD’-
anomaly for variable ’total’ (lines ’12’-’20’). </violation>
<violation beginline=”15” endline=”15” begincolumn=”17” endcolumn=”32”
rule=”SystemPrintln” ruleset=”Java Logging” class=”JavaExample” method=”main”
externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/logging-
java.html#SystemPrintln” priority=”2”> System.out.print is used </violation>
<violation beginline=”23” endline=”23” begincolumn=”9” endcolumn=”43”
rule=”LocalVariableCouldBeFinal” ruleset=”Optimization” class=”JavaExample”
method=”main” variable=”average” externalInfoUrl=”https://pmd.github.io/pmd-5.8.1/pmd-
java/rules/java/optimizations.html #LocalVariableCouldBeFinal” priority=”3”> Local variable
’average’ could be declared final </violation>
</file>
</pmd>
```


Figure 3.2: Example of PMD in XML format.

Therefore, code checkers have not been actively used in the real world. The key reasons include that a code checker displays a lot of violations (warnings) and there are many “false positive” in finding bugs [41]. Thus, there have been studies for enhancing the usefulness of code checkers by prioritizing warnings. Kim et al. [6] focused on the lifetime of violations on the repository, and they considered the shorter lifetime corresponds to the higher priority of violation. While their study provided meaningful insights, it would be more promising to consider not only the lifetime but also the change patterns of violations over the releases. For example, if there are two types of violations, A and B, and type A violations have been growing in a program but type B ones have kept a constant throughout the evolution, then type A may be less importance than type B.

If a coding violation is serious in a practical sense, programmers would improve their code fragments corresponding to the violation. Moreover, even if a code checker is not used, skilled programmers may find a part corresponding to a serious violation and improve it. Therefore, regardless of whether programmers have used a code checker or not, the changes of coding violations throughout the releases would be useful points to be focused on, and we will examine the usefulness of violations automatically detected by static code checkers in the real software projects. In order to clarify the aim of our empirical work in this chapter, we set up the following research questions (RQs):

RQ1: What kind of coding violations are related to the parts that many programmers tend to improve? and what kind of coding violations are likely to be disregarded?

RQ2: How can we reduce the meaningless violations (warnings) for programmers by omitting disregarded coding violations?

3.2 Empirical Work

3.2.1 Aim and Studied Projects

On the above RQs, we conducted empirical studies analyzing Java source files which have been developed and maintained in OSS projects. We focused on “violations” of source files warned by a code checker, and examined their distributions and trends through their releases.

We explored projects in the GitHub which is the most popular hosting service of code repository (Git). The Git offers powerful and lightweight

Table 3.2: Surveyed OSS projects.

Project Name	Investigation Period	#Releases	#Source Files
Guava	Jan. 2010 – Dec. 2015	59	1,678
Elasticsearch	Feb. 2010 – Feb. 2016	143	6,616
Spring Framework	Dec. 2008 – Apr. 2016	85	7,569
React Native	Mar. 2015 – Feb. 2016	56	2,075
JabRef	Dec. 2011 – Jan. 2016	23	1,259
JUnit4	Dec. 2004 – Dec. 2014	20	482
Hibernate	July. 2010 – Feb. 2016	111	9,993

functionalities for collecting source files and their changes. We randomly selected 7 OSS projects (see Table 3.2) satisfying the following conditions:

- They are popular projects ranked high in popularity (sorted by "stars" in the GitHub). We focused on popular projects in order to make our results as a high generality as possible.
- Their source files are written in Java. This restriction is from our data collection tools[4] we developed.
- They have been developed and maintained for at least 1 year. We excluded too young projects since they have a small number of releases and be difficult to discuss the trend of coding violations.

3.2.2 Data Collection

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

(1) In order to perform our analysis efficiently, we made a local copy (clone) of the target repository.

(2) For each release in each project, we got (checked out) all Java files from their repository. Then, we performed a code analysis of all Java files by using the PMD (ver.5.4.1) with its all rulesets and collected all violation data which appeared in those source files.

(3) For each project, we examined the change history of each violation according to the number of occurrences up to the latest version. By checking their change patterns, we observed which kind of violation had more

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

priority to be fixed and which one have been disregarded by the developers.

3.2.3 Analysis 1: Distribution of Violations Sorted by Predefined Priority

In order to capture the current status of coding violations in OSS products, we counted the violations warned by the PMD for the latest version of each product shown in Table 3.2. The PMD provides predefined priorities that consist of five levels: Level 1 (Change absolutely required), level 2 (Change highly recommended), level 3 (Change recommended), level 4 (Change optional), and level 5 (Change highly optional). Figure 3.3 shows band charts signifying ratios of the above violation-priority levels at the latest version of all the surveyed OSS products.

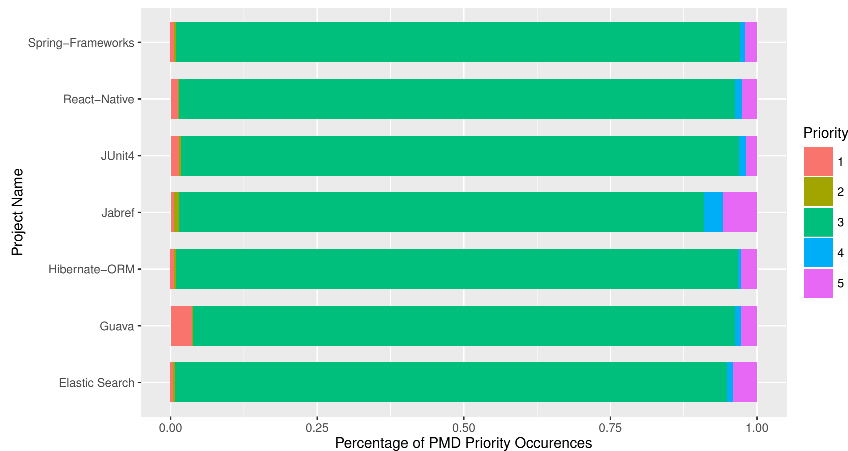


Figure 3.3: Ratio of violations by their priority level predefined in the PMD (for the latest version of each OSS product).

For all products, the violations of level 3 dominated: the percentages of level 3 were Spring-Framework: 96%, React-Native: 95%, JUnit4: 95%, Jabref: 90%, Hibernate: 96%, Guava: 92% and Elasticsearch: 94%. That is to say, there are actually many warnings which have a moderate level of recommendations to change the source code. We also iterated similar data collections for all release versions of each product. Figure 3.4 - Fig. 3.10 shows their results, where the horizontal axis signifies the number of violations and the vertical axis corresponds to their release versions; the latest version is at the head of the figure and the initial version is at the bottom of it. In these figures, green bars are corresponding to the level 3 violations.

Thus, the dominations by the level 3 violations have continued over their releases for all product. This trend may be one of the important reasons why many developers have not actively utilized static code checkers for their source files. While the level 3 violations do not seem to be trivial warnings, many programmers may have a difficulty in determining which parts should be improved preferentially because of a large number of warnings at the same priority level.

In the following section, we analyze a trend of each violation over the release version and discuss which type of violations are actually noteworthy, toward an effective utilization of code checkers.

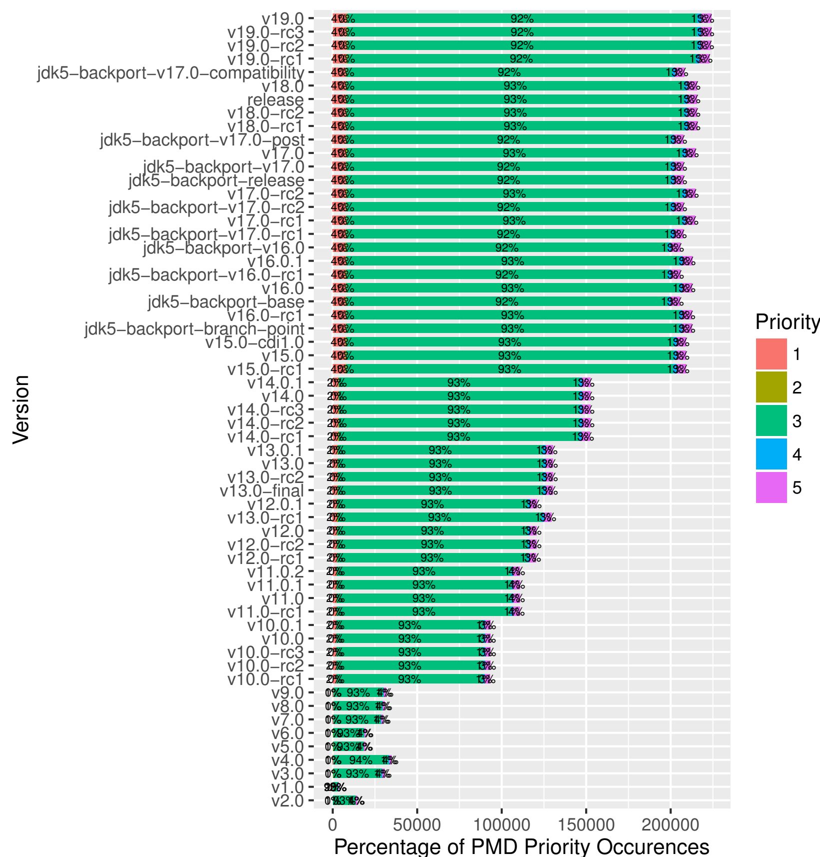


Figure 3.4: Trend of violations by the predefined priority level over all release versions: Guava.

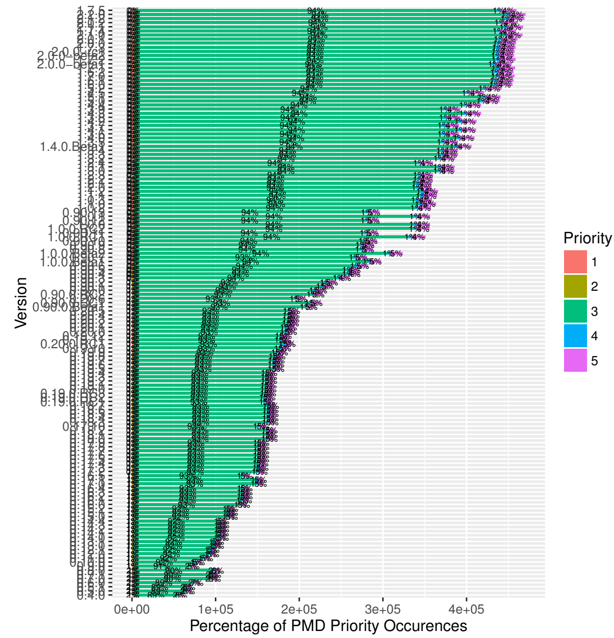


Figure 3.5: Trend of violations by the predefined priority level over all release versions: Elasticsearch.

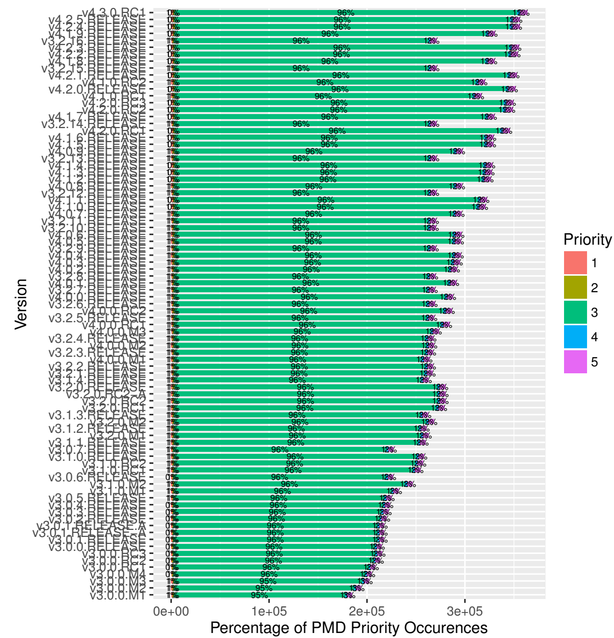


Figure 3.6: Trend of violations by the predefined priority level over all release versions: Spring-Framework.

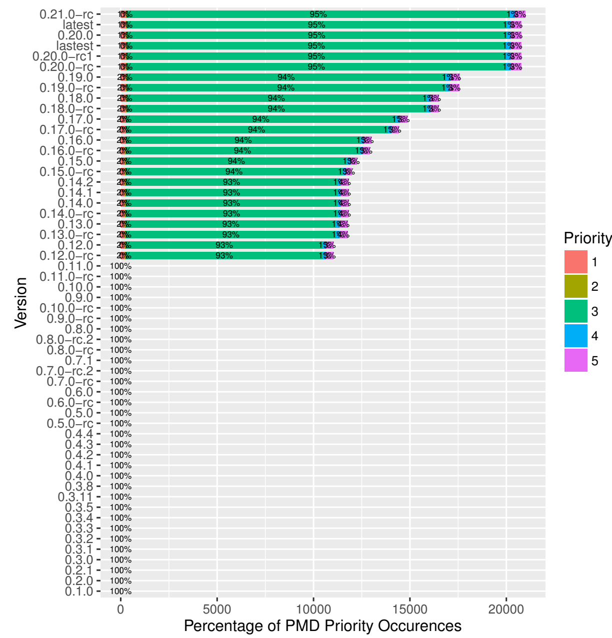


Figure 3.7: Trend of violations by the predefined priority level over all release versions: React-Native.

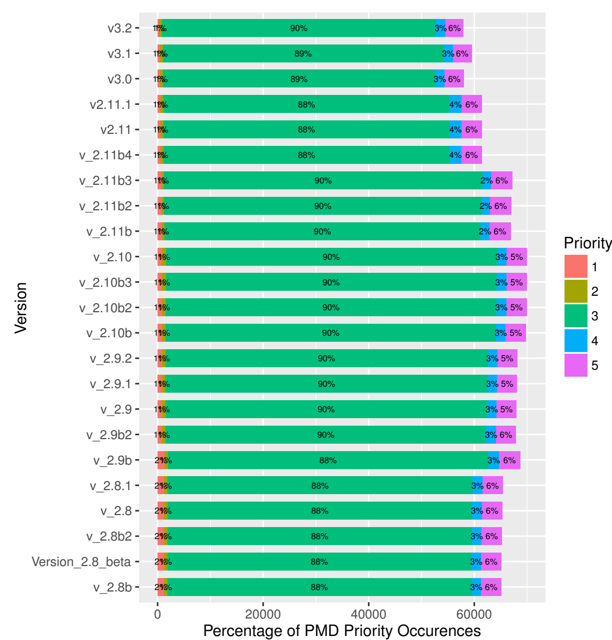


Figure 3.8: Trend of violations by the predefined priority level over all release versions: Jabref.

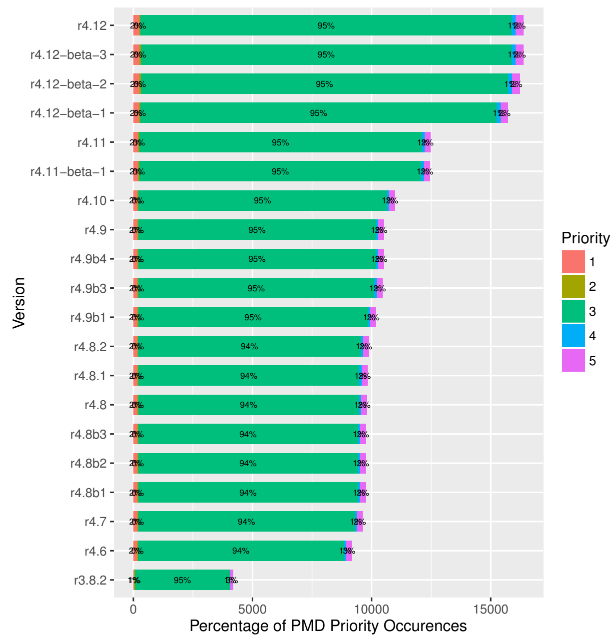


Figure 3.9: Trend of violations by the predefined priority level over all release versions: JUnit4.

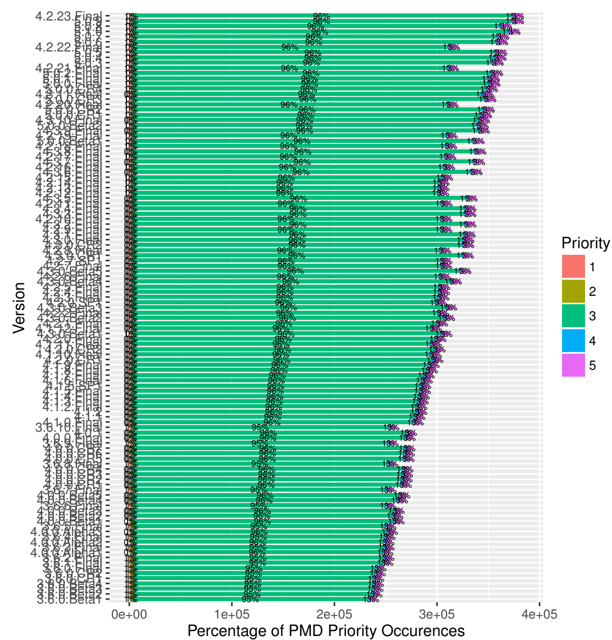


Figure 3.10: Trend of violations by the predefined priority level over all release versions: Hibernate.

3.2.4 Analysis 2: Trend of Each Violation

Trend Pattern

We examined all release versions of all source files included in the OSS projects shown in Table 3.2, by using the PMD. Then, we traced all violations warned by the PMD throughout the series of their release versions. For each source file and each coding violation, its change in the number of violations over the release is classified into the following five patterns: (1) one-shot, (2) sticky, (3) increasing, (4) decreasing and (5) fluctuating (see Fig.3.11). A “one-shot” violation appeared only once during all releases; A “sticky” violation has continued to appear since its first appearance, and their appearance count have been constant; An “increasing” violation and “decreasing” one showed growths and decays in their number of appearance over the release, respectively; A “fluctuating” violation had both growth(s) and decay(s) through the releases.

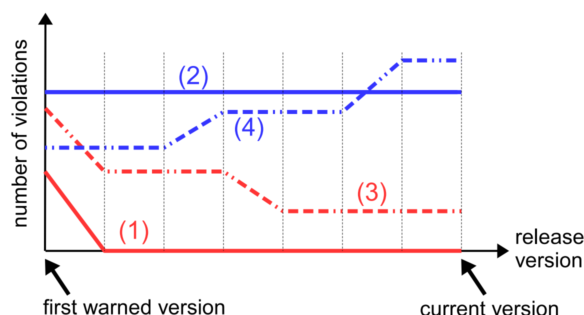
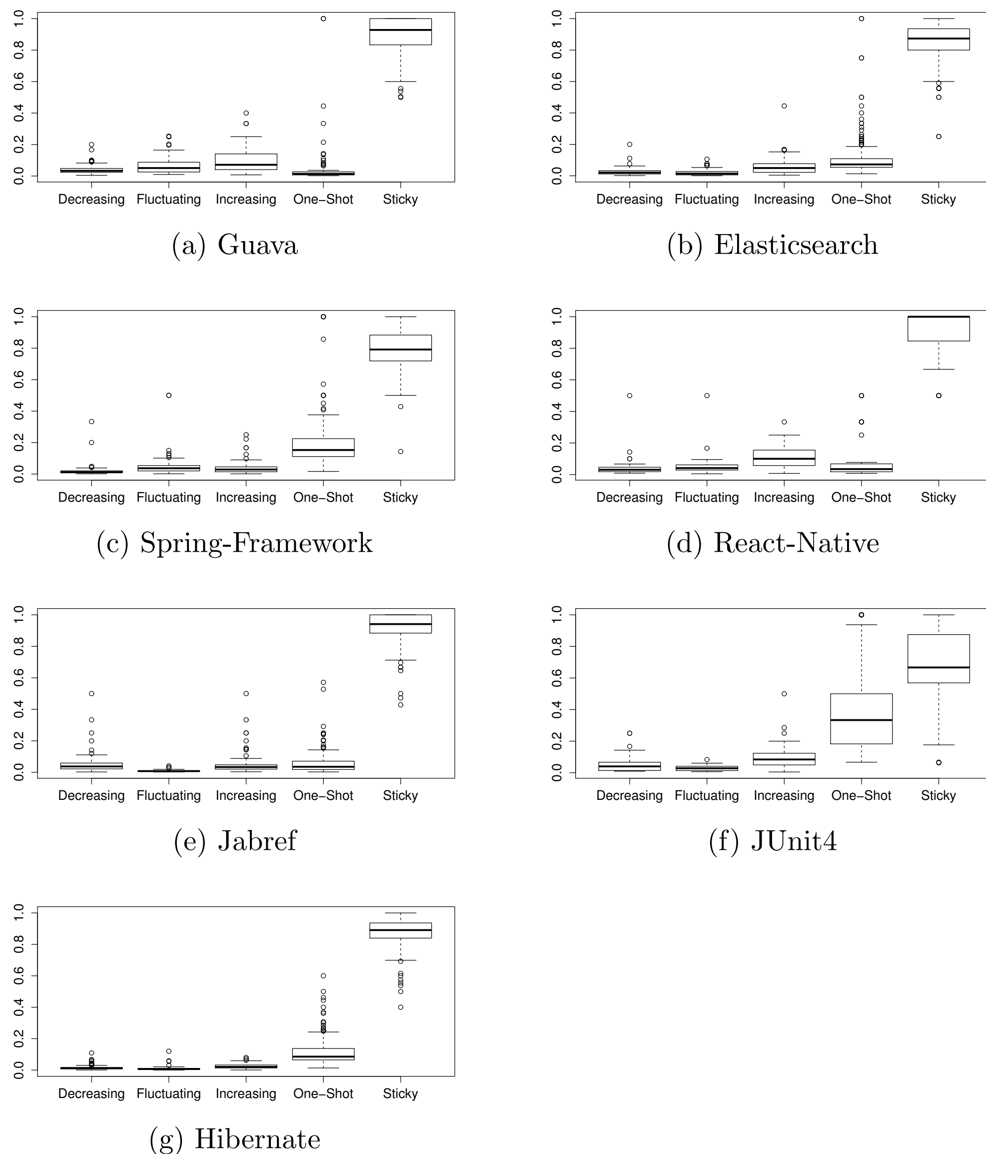


Figure 3.11: Example of change patterns.

Through our data collection and categorization, we observed different appearances of the PMD ruleset violations: 186 coding violations in (a) Guava, 221 ones in (b) Elasticsearch, 224 ones in (c) Spring-Framework, 129 ones in (d) React-Native, 206 ones in (e) JaRref, 208 ones in (f) JUnit4 and 217 ones in (g) Hibernate, respectively.

For example in (a) Guava, violation “CommentSize” was observed and had appeared in 2,336 files. Then, 0.5% of their appearances were categorized into (1) one-shot. Similarly, 77.4% were (2) sticky, 9.2% were (3) increasing, 4.1% were (4) decreasing and 8.8% were (5) fluctuating. In (c) Spring-Framework, the same “CommentSize” coding violations appeared in 12,990 files in which 13% categorized as (1) one-shot, 71.2% were (2) sticky, 4% were (3) increasing, 4.3% were (4) decreasing, and 7.5% were (5) fluctuating, respectively. Since those all results are too many to show in this

chapter, we summarize their trends in Fig. 3.12 and Table 3.3. Figure 3.12 shows the boxplots of the percentages of the above five patterns, and Table 3.3 presents the averages and medians of those percentages.



(a) Guava

(b) Elasticsearch

(c) Spring-Framework

(d) React-Native

(e) Jabref

(f) JUnit4

(g) Hibernate

Figure 3.12: Boxplot showing the rates of violation patterns in each OSS product.

Table 3.3: Averages and medians of violation trend patterns in each OSS project.

Project	Pattern	Average	Median
Guava	One-Shot	0.06	0.01
	Decreasing	0.04	0.03
	Fluctuating	0.07	0.05
	Increasing	0.09	0.07
	Sticky	0.90	0.93
Elasticsearch	One-Shot	0.12	0.07
	Decreasing	0.03	0.02
	Fluctuating	0.02	0.01
	Increasing	0.06	0.05
	Sticky	0.86	0.87
Spring Framework	One-Shot	0.20	0.15
	Decreasing	0.02	0.01
	Fluctuating	0.05	0.04
	Increasing	0.04	0.03
	Sticky	0.79	0.79
React Native	One-Shot	0.11	0.03
	Decreasing	0.06	0.03
	Fluctuating	0.07	0.04
	Increasing	0.11	0.10
	Sticky	0.91	1.00
JabRef	One-Shot	0.06	0.04
	Decreasing	0.06	0.04
	Fluctuating	0.01	0.01
	Increasing	0.05	0.03
	Sticky	0.92	0.94
JUnit4	One-Shot	0.40	0.33
	Decreasing	0.06	0.04
	Fluctuating	0.03	0.03
	Increasing	0.10	0.08
	Sticky	0.70	0.67
Hibernate	One-Shot	0.12	0.09
	Decreasing	0.02	0.01
	Fluctuating	0.01	0.01
	Increasing	0.02	0.02
	Sticky	0.88	0.89

From Fig.3.12 and Table 3.3, we observe a common trend to all OSS

projects such that a large majority of coding violations appeared in source files is categorized into "sticky" pattern. While we are not sure whether programmers checked their source files by using the PMD or not, there are many coding violations which have gone unattended. In other words, although it might be better to rewrite or redesign those parts warned as coding violations, many programmers may not have considered that they are not so wrong programs to be fixed and/or refactored. Therefore, there must be an interesting gap between the coding convention used in the code checker and the actual programming practices of many programmers. Those sticky violations would have been considered to be worthless in the real world of programmers. Similarly, the coding violations categorized as "increasing" pattern have also been disregarded through the development and maintenance of the software product. While those appearance rates are relatively low (2–10% in median), we will explore those violations along with the sticky ones later.

On the other hand, coding violations of "one-shot" pattern would be worthy of attention because they had appeared only once in the series of release versions, so those warned parts had already modified and recovered from their warned statuses soon. It is a fact that programmers had modified their programs corresponding to violations of the above "one-shot" ones sooner (by their next releases) regardless of whether programmers were used the PMD or not. Therefore, we consider the coding violations of "one-shot" pattern to be worthwhile in the quality management of source code. Moreover, the coding violations categorized as "decreasing" pattern would also be worthy to focus on. Since the number of code fragments warned by such violations has declined over the releases, programmers might be pay their attention to those parts and revised them. Indeed, violations of "one-shot" pattern and of "decreasing" pattern are not the majority of violations: the range of "one-shot" pattern ratio and that of "decreasing" pattern ratio are 1–33% and 1–4% in median, respectively. Such a minority is consistent with our interpretation that they would be important warnings for many programmers. We will explore coding violations belong to these two patterns below.

We will exclude the violations of "fluctuating" pattern from our discussion hereinafter since that pattern has aspects of both "increasing" and "decreasing" and we cannot clearly categorize them in terms of programmers' attention.

Violations Drawing Programmers' Attention

As mentioned above, while the violations of "sticky" or "increasing" pattern seem to be coding practices that are disregarded by programmers, the

violations of "one-shot" or "decreasing" pattern would draw programmer's attention. In simple terms, many programmers would consider the parts warned as the latter kind of violation to be problematic, and the ones corresponding to the former type would be worthless for enhancing the quality of their programs.

However, the trend of violations is not always the same—while a certain violation was of "sticky" pattern at a program, the violation was of "one-shot" pattern at another program. For instance, let us take a part of violation data in the Guava project, shown in Table 3.4.

Table 3.4: The number of violations by the trend pattern in the Guava (samples).

Violation	One-shot	Sticky	Increasing	Decreasing
...	...	...	...	...
CommentSize	12	51,528	7,433	4,242
...	...	...	...	...
ShortMethodName	1	2,311	131	450
...	...	...	...	...
TooManyFields	0	283	0	0
...	...	...	...	...

Violation "CommentSize" appeared at many parts of source programs, and there are all four patterns of interest. While violation "ShortMethodName" also has all four patterns, the ratio is different from the ones of "CommentSize." Violation "TooManyFields" has a totally different ratio of them: there is only one pattern—"sticky." In order to compare those violations from the perspective of programmers' attention, we now introduce a novel criterion, "the index of programmers' attention (IPA)" as follows:
Given a violation v,

$$\mathrm{IPA}(v) = \frac{N_o(v) + N_d(v)}{N_s(v) + N_i(v)} \ ,$$

where $N_o(v)$, $N_d(v)$, $N_s(v)$ and $N_i(v)$ are the numbers of parts warned as violation "v" belonging to "one-shot," "decreasing," "sticky" and "increasing," respectively. When $N_s(v) + N_i(v) = 0$, we define $\mathrm{IPA}(v) = \infty$. □

The numerator of $\mathrm{IPA}(v)$ corresponds to the number of parts that are warned as the violation v and have programmers' attention, and the denominator is also the number of parts that are warned as the violation v but they are disregarded by programmers. Thus, the higher IPA value of a violation v, the violation v tends to be regarded as more worthy in the quality management. For example of violations in Table 3.4, we get the following IPA

values:

$$\text{IPA(CommentSize)} = \frac{12 + 4242}{51528 + 7433} \simeq 0.072 \ ,$$

$$\text{IPA(ShortMethodName)} = \frac{1 + 450}{2311 + 131} \simeq 0.185$$

and

$$\text{IPA(TooManyFields)} = \frac{0 + 0}{283 + 0} = 0 \ .$$

Therefore, we can consider that violation "ShortMethodName" has more programmers' attention than "CommentSize," and "TooManyFields" tends to be disregarded by programmers.

Due to many violations appeared in the products, we will show the violations whose IPA value is ranked in the top 10 IPA values in Tables 3.5–3.11. In these tables, violations which appeared across projects are marked with an asterisk(*).

Table 3.5: Violations of top 10 IPA (Guava).

Violation	IPA
EmptyStatementNotInLoop	∞
UnnecessaryParentheses	∞
ForLoopsMustUseBraces	0.342
ShortMethodName	0.185
MissingBreakInSwitch	0.181
AssignmentInOperand	0.172
SimplifyBooleanReturns	0.167
LawOfDemeter	0.150
PreserveStackTrace	0.150
JUnitAssertionsShouldIncludeMessage	0.129

Table 3.6: Violations of top 10 IPA (Elasticsearch).

Violation	IPA
CloneMethodMustBePublic	1.000
UnnecessaryBooleanAssertion	0.557
BadComparison	0.500
JUnit4TestShouldUseAfterAnnotation	0.333
UnnecessaryCaseChange	0.225
NonStaticInitializer	0.178
GuardDebugLogging	0.167
AvoidCatchingGenericException	0.140
InefficientStringBuffering	0.130
UseObjectForClearerAPI	0.128

Table 3.7: Violations of top 10 IPA (Spring Framework).

Violation	IPA
EmptyFinallyBlock	∞
NonCaseLabelInSwitchStatement	∞
OptimizableToArrayCall	∞
SwitchDensity	∞
UseEqualsToCompareStrings	∞
CheckResultSet	0.930
FinalizeDoesNotCallSuperFinalize	0.500
InstantiationToGetClass	0.194
DuplicateImports*	0.117
GenericsNaming	0.111

Table 3.8: Violations of top 10 IPA (React Native).

Violation	IPA
UnnecessaryLocalBeforeReturn	1.000
DoNotUseThreads*	0.210
EmptyMethodInAbstractClassShouldBeAbstract	0.151
LooseCoupling	0.150
JUnit4TestShouldUseBeforeAnnotation	0.125
AvoidDuplicateLiterals	0.113
IfStmtsMustUseBraces	0.111
AvoidThrowingRawExceptionTypes	0.100
AvoidInstantiatingObjectsInLoops	0.078
DataflowAnomalyAnalysis*	0.078

Table 3.9: Violations of top 10 IPA (JabRef).

Violation	IPA
MisleadingVariableName	0.889
DuplicateImports*	0.250
CloseResource	0.228
UnsynchronizedStaticDateFormatter	0.214
ImportFromSamePackage	0.190
CommentDefaultAccessModifier	0.140
UselessQualifiedThis*	0.133
DoNotUseThreads*	0.122
BeanMembersShouldSerialize	0.110
DataflowAnomalyAnalysis*	0.108

Table 3.10: Violations of top 10 IPA (JUnit4).

Violation	IPA
AvoidDeeplyNestedIfStmts	∞
DoNotCallGarbageCollectionExplicitly	∞
DoNotExtendJavaLangError	∞
DoNotThrowExceptionInFinally	∞
ExcessiveMethodLength	∞
ExcessiveParameterList	∞
PrematureDeclaration	∞
ReplaceHashtableWithMap	∞
ReturnEmptyArrayRatherThanNull	∞
SimplifyStartsWith	∞
SwitchStmtsShouldHaveDefault	∞
TooFewBranchesForASwitchStatement	∞
TooManyFields	∞
UselessQualifiedThis*	∞

Table 3.11: Violations of top 10 IPA (Hibernate).

Violation	IPA
UnnecessaryFinalModifier	0.297
ExceptionAsFlowControl	0.167
IfElseStmtsMustUseBraces	0.149
UseIndexOfChar	0.131
NcssConstructorCount	0.097
UseAssertTrueInsteadOfAssertEquals	0.092
SwitchStmtsShouldHaveDefault	0.085
UseAssertNullInsteadOfAssertTrue	0.083
LocalVariableCouldBeFinal	0.080
AvoidUsingVolatile	0.074

While there are a few violations common to different projects (marked with an asterisk), a majority of the top 10 violations appeared in one project only. Thus, we can say that the important coding violations may vary from project to project. Indeed, there would be project-specific factors influencing the programming activity, such as the preferences of main contributors (developers) on the coding style. For example, while violation "ShortMethodName" is ranked as number 4 in the Guava's IPA list (see Table 3.5), it does not appear in the top 10 lists of projects other than the Guava, i.e., Tables 3.6–3.11. Although programmers in the projects other than the Guava did

not pay attention to whether a method's name is too short or not[5], it does not seem to be a popular feature to be checked and improved in any Java projects. This tendency seems to support the work by Boogerd et al. [42] in which they examined the MISRA-C coding violations and mentioned the importance of an appropriate ruleset selection for finding problematic parts in their programs.

Violations being disregarded

Next, we focus on the violations whose IPA value is zero. Since none of them has either "one-shot" or "decreasing" pattern, they are considered to be the ones being disregarded by the programmers. Table 3.12 shows the number of violations whose IPA value is zero. As shown in Table 3.12, the number of violations which have been disregarded by programmers is not little, and they would be worthless in checking programs by using tools in a practical sense.

Table 3.12: Number of violations whose IPA value is zero.

Project	#(IPA = 0)
Guava	90
Elasticsearch	42
Sprint Framework	57
React Native	85
JabRef	80
JUnit4	34
Hibernate	67

In total, 209 unique violations are the ones whose IPA value is zero, and 31 out of 209 violations ($\simeq 15\%$) are common to a majority of projects (four or more projects). Table 3.13 shows those 31 violations in the decreasing order of the number of projects in which the violation appeared with IPA = 0. Since those violations' IPA values are zero in a majority of surveyed projects, they tend to be disregarded by programmers. For instance, "AvoidArrayLoops" (see No.1 in Table 3.13) points out a code fragment in which there is a data copy between two arrays by using a loop (`for` statement for example); the code fragment is recommended using `System.arraycopy` method instead of using a loop. Many programmers do not seem to take care of whether they

[5] The rankings were No.126 in Elasticsearch, No.49 in Spring Framework, No.18 in React Native, No.127 (IPA= 0) in JabRef, No.110 in JUnit4 and No.131 in Hibernate-ORM, respectively.

Table 3.13: Disregarded violations (IPA= 0) common to four more projects.

No.	Violation	#projects
1	AvoidArrayLoops	6
2	MissingStaticMethodInNonInstantiatableClass	6
3	AvoidStringBufferField	5
4	AvoidUsingHardCodedIP	5
5	LoggerIsNotStaticFinal	5
6	ShortInstantiation	5
7	StringToString	5
8	SuspiciousEqualsMethodName	5
9	AvoidFieldNameMatchingTypeName	4
10	AvoidPrefixingMethodParameters	4
11	AvoidThreadGroup	4
12	BooleanGetMethodName	4
13	ByteInstantiation	4
14	ConsecutiveAppendsShouldReuse	4
15	ConsecutiveLiteralAppends	4
16	DoNotCallGarbageCollectionExplicitly†	4
17	DoNotThrowExceptionInFinally†	4
18	DontImportJavaLang	4
19	EmptyWhileStmt	4
20	FinalFieldCouldBeStatic	4
21	InefficientStringBuffering†	4
22	JUnit4TestShouldUseAfterAnnotation†	4
23	NcssTypeCount	4
24	NonStaticInitializer†	4
25	OptimizableToArrayCall	4
26	TooFewBranchesForASwitchStatement	4
27	UnnecessaryWrapperObjectCreation	4
28	UnusedNullCheckInEquals	4
29	UseAssertEqualsInsteadOfAssertTrue	4
30	UseAssertTrueInsteadOfAssertEquals†	4
31	UselessStringValueOf	4

should use `System.arraycopy` or write a code with using a loop. Therefore, such a point tends to be worthless in a practical sense.

However, all of them are not always disregarded by programmers; there are 6 violations which are also appeared in the top 10 IPA rankings (Tables 3.5–3.11)—they are marked with a dagger (†) in Table 3.13. For example,

"InefficientStringBuffering" (see No.21 in Table 3.13) appeared in the IPA top 10 ranking of Elasticsearch (see Table 3.6) but its IPA value is zero in 4 out of the remaining 5 projects. That violation corresponds to the case that non-literals are concatenated by using "`+`" operator in a `StringBuffer` constructor or an `append` method. Many programmers might write such a program because of a convenience of operator "`+`." However, since such a case is inefficient in terms of the memory allocation, there are also programmers in the development of the Elasticsearch, who considered those parts should be improved.

While there are some exceptional instances like those six violations (marked with a dagger) shown in Table 3.13, most of them are likely to be considered less serious.

IPA vs. Number of Violations

Finally, we analyze a relationship between the IPA value and the number of violations. Table 3.14 shows the distribution of IPA values across all projects, where "x%" signifies the x percentile of IPA values (for $x = 10, 20, \ldots, 90$). From Table 3.14, we divide the set of violations into eight subsets by their IPA values: (1) IPA $= 0$, (2) $0 <$ IPA $\leq$ the 40 percentile (40%), (3) 40% $<$ IPA $\leq$ 50%, (4) 50% $<$ IPA $\leq$ 60%, $\cdots$, (7) 80% $<$ IPA $\leq$ 90%, and (8) IPA $>$ 90%. Figure 3.13 presents the number of violations by the IPA category. From Fig. 3.13, we can say most of the violations seem to be the ones whose IPA values are relatively high. Therefore, even though there were many violations in the real programs, they had also been related to the parts which had been improved by programmers in reality. In other words, unimportant violations whose IPA values are zero or nearly zero are a minority of violations automatically detected by the PMD: (1) 1%, (2) 4%, (3) 7%, (4) 10%, (5) 17%, (6) 19%, (7) 28% and (8) 14%. For instance, the violations whose IPA values are less than or equal to the median of all IPA values form only 12%, which is obtained by (1) + (2) + (3).

Table 3.14: Distribution of IPA values across all projects.

Min.	10%	20%	30%	40%	50%
0	0	0	0	0.00274	0.00786

60%	70%	80%	90%	Max.
0.0181	0.0303	0.0476	0.0800	∞

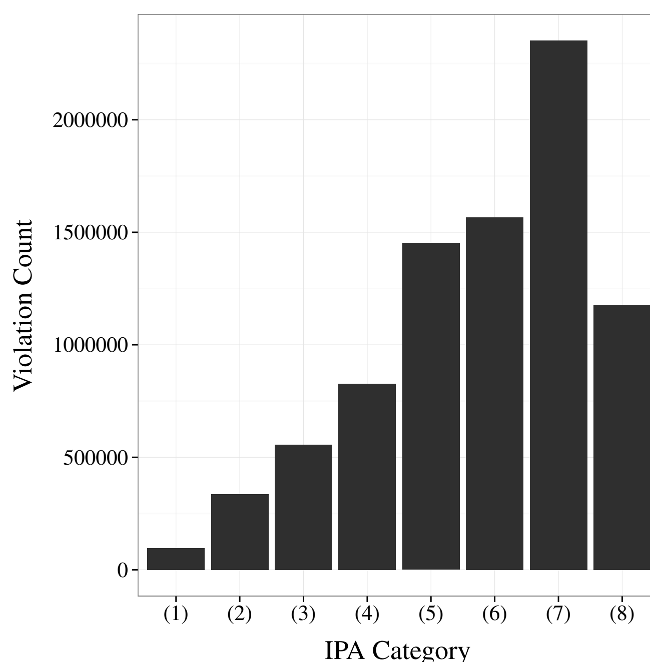


Figure 3.13: Number of violations by IPA category.

3.2.5 Threats to Validity

Our data is limited to Java products because our tool supports only Java. Thus, although we randomly selected popular Java OSS projects from the GitHub, we cannot say that we have no selection bias of data in our empirical work. We have to analyze OSS products developed in a language other than Java as well.

In our data set, there might be code fragments which are removed from their source files or moved into other source files. Although the number of such code fragments would be small, they can be a noise in our analysis based on IPA value.

3.3 Related Work

There have been several studies that specifically targeted OSS projects to develop actionable alert identification techniques based on automated static code analysis warning results[11]. Among those studies, Spacco et al.[5], Kim et al.[6] and Lee et al.[10] utilized a static code analyzer and identified influential coding violations or warnings in finding bugs: Spacco et al.

exploited a fuzzy algorithm to determine commonalities among warnings; Kim et al. focused on the lifetime of warning (violation) and used it for their prioritization; Lee et al. analyzed the effects of coding violations on the readability of programs. Although our work is also based on the coding violations detected by a code checker and evaluates the importance of violation, our evaluation approach utilized the real code modifications made by programmers. Moreover, our attempt to make the connection between human factors (programmers' attentions) and violation trend offers reasonable perspective to explain how developers have contributed the quality managements with software evolution.

Hanam et al.[12] stated that a utilization of the source code history on the repository can be useful in projecting trends of alerts (warnings) on the source code, and thus be beneficial for predicting the source code evolution. While our approach is partially similar to their work from the perspective of focusing on both the code evolution on the repository and the trends of coding violations over the releases, our main focus is on whether the programmer had fixed the warned parts or not in reality.

Avgustinov et al.[15] proposed a "violation matching approach" and "developer fingerprinting" to reveal coding habits of individual developers using revision history. They tracked developer's activity chronologically using the repository together with the analysis results from their code analysis tool. While their approach is beneficial, their focus was not on the trend of violations. However, a further analysis focusing on the developer's coding practice like the work by Avgustinov et al. is our significant future work.

3.4 Conclusion

In this chapter, we focused on Java coding violations detected by the PMD, a popular static code analysis tool. A coding violation may be related to a code fragment which should be improved or refactored, so such a violation might be a useful information in regard to the quality of code.

We statistically investigated the trend of violations over the releases of seven popular OSS products, and categorized them by their trend patterns. Then, we introduced the index of programmers' attention (IPA) in order to quantify the degree to which programmers paid attention to the part warned by the violation. Through analyses using IPA, we obtained the following findings regardless of whether a programmer is using the PMD or not: (1) important violations (having high IPA values) may vary from project to project; (2) there are some unimportant violations common to different projects, but they are a minority of violations detected by the PMD (about 12%). There-

fore, while many violations may be made by the PMD, most of them are likely to be worthy in improving the code quality, and it is ineffective to reduce the violations by eliminating such unimportant violations. In reality, however, we cannot deny that there are many "false positive" in their warnings because most IPA values are low (less than a few percent).

Chapter 4

Impact of Individual Difference on Coding Violation

The programming has been widely known as the most significant activity for producing high-quality software systems. The programming activity is usually performed by a programmer, i.e., a human being. Thus, it is hard to avoid any human errors during the programming, so there is always a risk of a latent fault in a source code. Moreover, such a risk usually gets higher through the system's upgrade since a code change may create another fault [43]. In order to reduce the risk of latent faults, it is effective to perform a careful code review [44]. A code review can find potentially problematic parts of source programs, which are fault-prone parts or hard-to-understand ones. While a code review is a useful activity for enhancing the quality of source code, it is also costly activity. For a large-scale system, it is not easy to perform a thorough review for all source programs. Furthermore, whenever a large-scale system is upgraded, it is impractical to review not only modified code but also all possible parts which may be affected by the modifications. Thus, a manually-performed code review is limited by lacks of time, effort and manpower. To support code reviews by programmers themselves, there have been automated support tools referred to as "static code analysis tools." Those tools point the parts which violate to a predefined coding convention or the ones which match to a predefined anti-coding pattern. Since such tools can find potential poor quality parts or latent faults, they can be great helps for the code review and thus for the quality management of code.

However, such analysis tools have not been actively utilized in reality [45]. In many cases, a lot of violations (warnings) are reported by a tool and many of them are not actually important (false positives), then programmers tend to have hesitations in using such a tool. Thus, there have been studies for prioritizing (evaluating) violations to reduce the false-positive rate in the

past. We focused on change patterns of violations over releases, and proposed a metric for evaluating violations, "Index of Programmers' Attention (IPA)" in the previous chapter. When some parts of a source file were warned as a violation and the number of those warned parts have been decreased through their upgrades, such a decreasing trend is a proof that the programmer paid an attention to the violation and fixed them. On the other hand, if the number of warned parts have been constant or increased through their upgrades, the programmer would disregard the corresponding violation. IPA is a ratio of the former cases to the latter cases as an index of violation's importance. While an empirical study using IPA was reported in the previous chapter, the study missed a consideration for the authorship of source files. When a source file has been developed and maintained by a single programmer, the violations depend on his/her preference. If two or more programmers are involved in the source file, the violations would be influenced by common sense of those programmers. Thus, this chapter will examine the impact of authorship on the above evaluation of violations and report the results of analysis.

4.1 Evaluation of Violation and Authorship of Source File

4.1.1 Evaluation of Violation

A static code analysis tool can automatically check all source files included in a release version of a large-scaled software product. While we can easily perform such a thorough checking thanks to an automated tool, it may be hard to utilize the results of checking because of a large number of violations (warnings) made by the tool—we may face too many violations to examine whether we should modify the warned parts or not. Hence, a proper prioritization of violation has been required for a successful utilization of static code analysis tools.

In order to automatically evaluate the priority of violation from the perspective of code change history, we focused on change patterns of violations over releases in the previous chapter: (1) one-shot, (2) sticky, (3) decreasing, (4) increasing, and (5) other. The one-shot pattern of a violation means that it appeared only at one version through all releases. The sticky pattern of a violation refers to the case that the number of appearances is constant from its first warned version to the latest one. The decreasing pattern of a violation corresponds to the case that the number of appearances monotonically decreased after its first appearance version, and the increasing pattern

means a monotonically increasing case. The other pattern is a mixed case of the above four patterns. Violations belong to the one-shot pattern or the decreasing one seem to be paid attentions by programmers since one or more violations had been eliminated through a upgrade. On the other hand, violations of the sticky pattern or the increasing one would be disregarded by programmers because those ones were not cleared throughout upgrades.

Then, we proposed the following metric, "Index of Programmers' Attention (IPA)" using these notions: For a violation (warning) v,

$$\mathrm{IPA}(v) = \frac{N_o(v) + N_d(v)}{N_s(v) + N_i(v)} \ , \tag{4.1}$$

where $N_o(v)$, $N_d(v)$, $N_s(v)$ and $N_i(v)$ are the numbers of parts warned as violation v belonging to "one-shot," "decreasing," "sticky" and "increasing," respectively. When $N_s(v) + N_i(v) = 0$, it is defined as $\mathrm{IPA}(v) = \infty$. Notice that the study excludes violation v such that $N_o(v)+N_d(v)+N_s(v)+N_i(v) = 0$.

4.1.2 Impact of Authorship on Evaluation

Although IPA can be useful in evaluating many violations automatically, it may be affected by differences in style of development. If a source file has been developed and maintained by a certain programmer only, the paying-attention depends on the programmer's preference. Thus, we will focus on whether a source file has been developed and maintained by a single programmer or not, and examine its impact on the violation evaluation in this paper.

We will call a source file which has been developed and maintained by a single developer, as a "single-authored file"; we will refer to a source file which has been developed or maintained by two or more developers, as a "multi-authored file."

To statistically compare evaluations of a violation between single-authored file and multi-authored one, we cannot use the original IPA—Eq. (4.1)—since IPA value can be ∞. Thus, we introduce the normalized IPA (NIPA) as follows:

$$\mathrm{NIPA}(v) = \frac{\{N_o(v) + N_d(v)\} - \{N_s(v) + N_i(v)\}}{N_o(v) + N_d(v) + N_s(v) + N_i(v)} \ . \tag{4.2}$$

The range of NIPA value is $[-1, 1]$. A violation v is evaluated in accordance with its $\mathrm{NIPA}(v)$ value as follows:

(i) $\mathrm{NIPA}(v) = 1$: $N_o(v)+N_d(v) > 0$ and $N_s(v)+N_i(v) = 0$. There are only cases that programmers paid attentions to violation v. In these cases,

violation v must be related to problematic or fully-undesirable code. Such a violation would be important in the code quality management.

(ii) NIPA$(v) = -1$: $N_o(v) + N_d(v) = 0$ and $N_s(v) + N_i(v) > 0$. This is totally opposite to (i), and there are only cases that programmers disregarded v. Hence, v must be insignificant in the maintenance of the software product.

(iii) NIPA$(v) > 0$: $N_o(v) + N_d(v) > N_s(v) + N_i(v)$. There are both cases that programmers paid attentions and disregarded, but the number of former cases is greater than that of latter cases. Thus, v would be relatively important.

(iv) NIPA$(v) \leq 0$: $N_o(v) + N_d(v) \leq N_s(v) + N_i(v)$. This is opposite to (iii), so v would be relatively disregarded by programmers.

In simple words, if violation v has a positive and greater NIPA(v) value, v is considered to be more important for more programmers, and vice versa.

For example, suppose we have a set of trend data for violation v, shown in Table 4.1. The table gives numbers of cases observed in single-authored files and in multi-authored ones, respectively: 8 one-shot cases in single-authored files, 12 decreasing cases in multi-authored files, etc.

Table 4.1: Example of violation v's trend.

	important		disregarded	
	one-shot	decreasing	sticky	increasing
authorship	$N_o(v)$	$N_d(v)$	$N_s(v)$	$N_i(v)$
single	8	24	10	18
multi	14	12	20	18

Using Eq. (4.1), we able to compare NIPA for single-authored files and multi-authored files as follow:

$$\text{NIPA single-authored}(v) = \frac{\{8 + 24\} - \{10 + 18\}}{8 + 24 + 10 + 18} = 0.067.$$

$$\text{NIPA multi-authored}(v) = \frac{\{14 + 12\} - \{20 + 18\}}{14 + 12 + 20 + 18} = -0.1875\ .$$

In this example, while v is considered to be relatively important by some programmers within their single-authored files, it seems to be disregarded by

other programmers since its multi-authored NIPA value is negative. Therefore, we can say the violation v is more important to the programmer with single authorship compare to the multi authorship one.

Using the above metric, NIPA, we will analyze the impact of authoring type—single-authored files vs. multi-authored ones—on the evaluation of violations in the following section. To clarify our goals of empirical study, we set our research questions (RQs) as follows:

RQ1: Is the authoring type noteworthy in evaluating coding violations?

RQ2: How many violations are commonly important or worthless across projects and authoring types?

4.2 Empirical Study

4.2.1 Aim and Dataset

In order to answer the above RQs, we examine open source software (OSS) products from the perspective of not only the coding violation importance (NIPA value) but also the file authorship. Table 4.2 shows the projects surveyed in this study. These projects are randomly selected from GitHub. By comparing NIPA values of coding violations between the sets of single-authored files and of multi-authored ones, we study an impact of authorship on the priority of coding violations.

Table 4.2: Surveyed OSS projects.

Project Name	Investigation Period
Guava	Jan 2010 – Dec 2015
Elasticsearch	Feb 2010 – Feb 2016
Spring Framework	Dec 2008 – Apr 2016
React Native	Mar 2015 – Feb 2016
JabRef	Dec 2011 – Jan 2016
JUnit4	Dec 2004 – Dec 2014
Hibernate	Jul 2010 – Feb 2016

4.2.2 Data Collection

For each project, we conducted our data collection in the following procedure:

(1) We cloned the repository on our local disk to analyze the trends of coding violations smoothly.

(2) For each release version, we checked out all files and performed a static code checking by using the PMD (ver. 5.4.1) with its all rule sets.

(3) For each file f, we examined who created f or made changes to f by checking commit logs which f has been involved in, i.e., author(s). Then, we counted the unique number of authors associated with f, and decided if f is a single-authored file or a multi-authored one. Since there may be an author who has two or more different names or e-mail addresses, we integrated duplicated authors by the following rules [39]:

   (i) if two authors have different addresses but the same name, then we regard them as the same author.

   (ii) if two authors have the same address but different names, then we regard them as the same author.

(4) For each single-authored file f_s and each violation v, we traced the change of its occurrences over releases and decided its change pattern from 1) one-shot, 2) sticky, 3) decreasing, 4) increasing and 5) other. Similarly, we decided change patterns of all violations by checking all multi-authored files as well.

(5) For each violation v appearing in single-authored files, we obtained $N_o(v)$, $N_s(v)$, $N_d(v)$ and $N_i(v)$ by counting the decided patterns in all files, then computed NIPA(v). Similarly, we computed NIPA(v) of all violations appearing in multi-authored files as well.

4.2.3 Analysis 1 (for RQ1): Comparison of Violations Appearing in Single-Authored Files vs. Multi-Authored Files

If the difference in the authoring type—single author vs. multi authors—has an impact on the coding violations, there are differences in the sets of violations or in the NIPA values. That is to say, the former type is a distinct difference such that the set of violations appearing in the single-authored files differs from the set of ones appearing in the multi-authored files. On the other hand, the latter type of difference is more complicated: although appearing violations are common regardless of the authoring type, there is a discrepancy in their evaluations. Thus, we checked these two types of differences.

Differences in the Sets of Violations

At first, for each OSS project, we examined the similarity between the sets of violations which appear in the single-authored files and in the multi-authored ones, respectively. We compute similarities between them with using three popular indexes—the Jaccard index, the Dice index and the Simpson index: Let V_s and V_m be the sets of violations appearing in the single-authored files and in the multi-authored ones, respectively. The Jaccard index, $Jac(V_s, V_m)$, is computed with the following equation:

$$Jac(V_s, V_m) = \frac{\mid V_s \cap V_m \mid}{\mid V_s \cup V_m \mid} \ . \tag{4.3}$$

The Dice index, $Dice(V_s, V_m)$, is obtained with the following equation:

$$Dice(V_s, V_m) = \frac{2 \mid V_s \cap V_m \mid}{|V_s| + |V_m|} \ . \tag{4.4}$$

The Simpson index, $Simp(V_s, V_m)$, is computed with the following equation:

$$Simp(V_s, V_m) = \frac{\mid V_s \cap V_m \mid}{\min\{\ |V_s|, |V_m|\ \}} \ . \tag{4.5}$$

The above three indexes range from 0 to 1: a higher value corresponds to a pair of sets which are more similar, i.e., there are more common elements in those sets. Table 4.3 shows the similarities.

Table 4.3: Similarities between violation sets: single-authored files vs. multi-authored files.

Project	*Jac*	*Dice*	*Simp*
Guava	0.812	0.897	0.952
Elasticsearch	0.791	0.883	0.979
Spring Framework	0.637	0.779	0.975
React Native	0.547	0.707	0.914
JabRef	0.397	0.569	1.000
JUnit4	0.115	0.206	0.929
Hibernate	0.659	0.795	0.992

As the results, there are varieties in the similarity across projects. Most violations appearing in Guava and Elasticsearch are common between the sets of single-authored files and of multi-authored ones, where their similarities are around or higher than 0.8 in terms of all similarity indexes. That is to say, about 80% or more violations are common to both single-authored files and

multi-authored ones. On the other hand, JUnit4 shows low-level similarities in terms of Jaccard index and Dice index (0.115 and 0.206). However, its Simpson index is high (0.929). JabRef shows relatively similar results—low values in Jaccard and Dice indexes, but the highest value in Simpson index. If $V_s \subseteq V_m$, the Simpson index becomes 1.0; JabRef is in that case. Indeed, all projects show high Simpson index values (> 0.9). Thus, their dissimilarities in Jaccard and Dice indexes seem to be caused because the relationship between V_s and V_m is close to a case that $V_s \subseteq V_m$, i.e., $|V_s| \simeq |V_s \cap V_m|$ (see Table 4.4).

Therefore, there are likely to be differences between the sets of violations appearing in the single-authored files and of the multi-authored ones, and the variety of violations in the single-authored files tend to be more limited than that in the multi-authored files.

Table 4.4: Number of elements in V_s, V_m and $V_s \cap V_m$.

Project	$\lvert V_s \rvert$	$\lvert V_m \rvert$	$\lvert V_s \cap V_m \rvert$
Guava	146	164	140
Elasticsearch	143	174	139
Spring Framework	119	179	116
React Native	70	111	64
JabRef	66	166	66
JUnit4	14	112	13
Hibernate	123	184	122

Differences in NIPA Values

Next, we will examine whether there is a difference in the trends of violation priorities (NIPA values) in accordance with the file authorship. Table 4.5 summarizes the numbers of violations according to their NIPA values and their authoring types.

Table 4.5: Number of violations according to their NIPA values and authoring types.

Project	NIPA	Authoring Type	
		Single	Multi
Guava	$= -1$	105	94
	$\in (-1, 0]$	41	67
	$\in (0, 1)$		**1**
	$= 1$		**2**
Elasticsearch	$= -1$	66	61
	$\in (-1, 0]$	74	110
	$\in (0, 1)$		**1**
	$= 1$	**3**	**2**
Spring Framework	$= -1$	100	86
	$\in (-1, 0]$	19	91
	$\in (0, 1)$		**1**
	$= 1$		**1**
React Native	$= -1$	46	78
	$\in (-1, 0]$	20	33
	$\in (0, 1)$		
	$= 1$	**4**	
JabRef	$= -1$	51	64
	$\in (-1, 0]$	12	97
	$\in (0, 1)$		**2**
	$= 1$	**3**	**3**
JUnit4	$= -1$	14	77
	$\in (-1, 0]$		34
	$\in (0, 1)$		
	$= 1$		**1**
Hibernate	$= -1$	55	67
	$\in (-1, 0]$	64	117
	$\in (0, 1)$		
	$= 1$	**4**	

In the table, the number of violations whose NIPA values are positive—being considered to be important—is expressed in boldface. From the table, we can see the common trend that most appearing violations are disregarded (NIPA < 0), and only a few violations are paid attention by programmer(s) (NIPA> 0).

Table 4.6 shows violations whose NIPA values are greater than 0, according to authorships. While there are 23 violations being considered important (NIPA > 0) in either the single-authored files or the multi-authored ones, 21 out of 23 violations appear only in one of two authoring types. That is to say, important violations getting programmers' attention tend to differ in accordance with the authorship of source file; Even if a violations is considered to be important at a single-authored file, it may be disregarded by many other programmers. Such a difference may come from the preference of programmer.

Table 4.6: Appearing violations whose NIPA values are greater than zero in single-authored files or multi-authored files.

	Guava		Elasticsearch		Spring Framework		React Native		JabRef		JUnit4		Hibernate	
Violation	S	M	S	M	S	M	S	M	S	M	S	M	S	M
AddEmptyString		0.587				0.396								
AppendCharacterWithChar									1					
AvoidCatchingNPE													1	
AvoidUsingShortType										1				
BadComparison				1										
ConfusingTernary							1							
DoNotThrowException InFinally												1		
DontImportJavaLang				1										
DuplicateImports										1				
EmptyStatementNotInLoop		1												
ForLoopsMustUseBraces		1												
IfStmtsMustUseBraces							1							
ImportFromSamePackage									1	1			1	
JUnit4TestShouldUse TestAnnotation			1											
MisleadingVariableName										0.143				
PositionLiteralsFirstInCase InsensitiveComparisons			1										1	
SimplifyBooleanReturns							1							
SingletonClassReturning NewInstance									1					
UnusedImports										0.118				
UnusedLocalVariable						1								
UseLocaleWith CaseConversions							1							
UseProperClassLoader			1	0.500										
TooManyStaticImports													1	

(S: Single-authored files; M: Multi-authored files)

In Table 4.6, only two violations (emphasized in boldface) have positive NIPA values in both of the authoring types: "UseProperClassLoader" in Elasticsearch and "ImportFromSamePackage" in JabRef. The former violation is a recommendation to replace the invocation of getClassLoarder() with Thread.currentThread().getContextClassLoader() because the original code might not work properly in the J2EE environment. The latter violation says that that there is no need to import classes within the same package. Since the former violation seems to be related to a potential fault, it is natural that the violation was fixed regardless of the authorship. On the other hand, the latter violation would be on the way of coding and have no relation with a fault. Hence, making the violation totally depends on who writes the code: While "ImportFromSamePackage" has also the highest NIPA value in the single-authored files of Hibernate, it does not appear in the multi-authored ones of the same project.

In order to examine further correspondence relationships between the sets of violations warned in single-authored files and in multi-authored ones, we compared their NIPA values. For each project, we computed the Spearman rank correlation coefficient between the sets of NIPA values corresponding to violations warned in both the single-authored files and the multi-authored ones. Table 4.7 shows the results. In Table 4.7, no strong correlation between NIPA values is observed in our data. That is to say, the evaluation of a violation appearing in single-authored files tends to be independent of that in multi-authored ones, even when we focus only on the common violations.

Table 4.7: Spearman rank correlation coefficients between the set of single-authored files and that of multi-authored ones in terms of NIPA value.

Project	Correlation Coefficient
Guava	0.492
Elasticsearch	0.339
Spring Framework	0.433
React Native	0.127
JabRef	0.160
JUnit4	0.000
Hibernate	0.344

To understand how two sets of source files differ in terms of NIPA value, we computed the following $\Delta\text{NIPA}(v)$ for each violation v in each project:

$$\Delta\text{NIPA}(v) = \text{NIPA}_{single}(v) - \text{NIPA}_{multi}(v) \ , \quad (4.6)$$

where $\text{NIPA}_{single}(v)$ and $\text{NIPA}_{multi}(v)$ are the NIPA value of violation v in the set of single-authored files and in that of multi-authored one, respectively.

Table 4.8 shows the distributions of ΔNIPA(v). While there are a few extreme cases (ΔNIPA(v) $= -2$ or 2), ΔNIPA values of most violations are less than or equal to zero. Figure 4.1 presents boxplots of ΔNIPA values, which focus only on around zero. From Fig. 4.1, we can see the trend that ΔNIPA values tend to be negative in many cases. Thus, the degree of importance of a violation v—NIPA(v)—would increase from a single-authored file to a multi-authored file. In other words, even if a violation in a source file is disregarded by a programmer, the violation (warning) may be resolved by another programmer when the source file becomes a multi-authored type from a single-authored one.

Table 4.8: Distributions of ΔNIPA(v).

		Percentile			
Project	Min.	25%	50%	75%	Max.
Guava	−2.000	−0.082	0	0	0.865
Elasticsearch	−0.670	−0.051	−0.0005	0.002	2.000
Spring Framework	−0.371	−0.091	−0.008	0	0.377
React Native	−0.422	−0.098	0	0	2.000
JabRef	−0.958	−0.286	−0.151	−0.053	2.000
JUnit4	−0.383	−0.150	−0.095	−0.036	0
Hibernate	−0.717	−0.081	−0.014	0	2.000

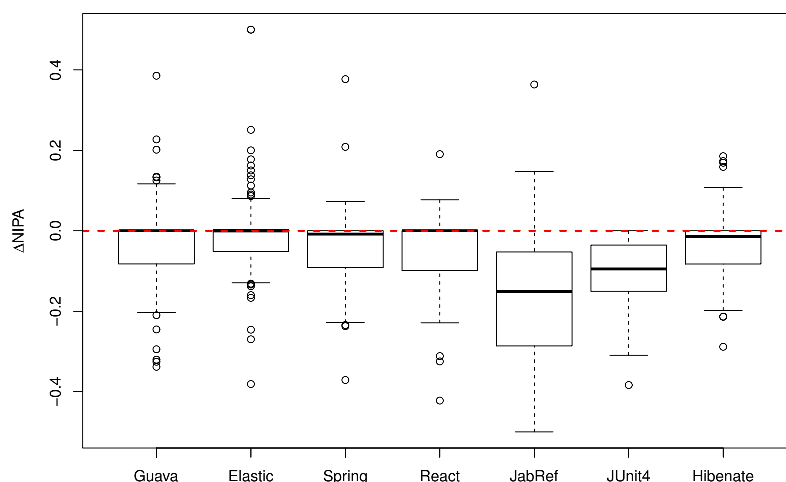


Figure 4.1: Boxplots of ΔNIPA(v) (focused only on the range $[-0.5, 0.5]$).

From the all results shown in this subsection, we can answer to RQ1 as: the difference in the authoring type has significant impacts on the trends of violations and their evaluations.

4.2.4 Analysis 2 (for RQ2): Comparison of Violations across Projects

Now we introduce another perspective of analysis: the comparison across the projects.

For violations appearing in single-authored files and in multi-authored files, we computed similarities among projects with using three indexes—Jaccard index, Dice index and Simpson index. Table 4.9 shows the results. For violations appearing in single-authored files, the averages of similarities (excluding the ones with themselves) in terms of Jaccard index, Dice index and Simpson index are 0.391, 0.534 and 0.859, respectively; The averages of similarities in multi-authored files are 0.634, 0.771 and 0.877, respectively. Thus, in all three indexes, the commonalities of violations appearing in the multi-authored files are higher than those in the single-authored ones. Indeed, for all pairs of projects, Jaccard indexes and Dice indexes in multi-authored files are higher than the ones in single-authored ones—for example, pair (b)-(c): $0.638 < 0.7834$ (in Jaccard index) and $0.779 < 0.878$ (in Dice index). While Simpson indexes show some opposite relationships, the reason would be that Simpson index tends to have a higher value when there is a big gap between compared sets in terms of size (number of elements)—Since JUnit4 has only 14 elements (violations) in its set of violations appearing in the single-authored files.

Hence, the multi-authored files are more likely to have a higher commonality of violations than the single-authored ones across projects. Trends of appearing violations are possibly generalized through maintenances by two or more programmers.

Table 4.9: Similarities of appearing violatins among projects.

Project	Index	(i) in single-authored files (a)	(b)	(c)	(d)	(e)	(f)
(a) Guava		—	—	—	—	—	—
(b) Elasticsearch	J	0.606	—	—	—	—	—
	D	0.754	—	—	—	—	—
	S	0.762	—	—	—	—	—
(c) Spring Framework	J	0.514	0.638	—	—	—	—
	D	0.679	0.779	—	—	—	—
	S	0.756	0.857	—	—	—	—
(d) React Native	J	0.403	0.449	0.443	—	—	—
	D	0.574	0.62	0.614	—	—	—
	S	0.886	0.762	0.829	—	—	—
(e) JabRef	J	0.333	0.375	0.445	0.432	—	—
	D	0.5	0.545	0.616	0.603	—	—
	S	0.803	0.864	0.864	0.621	—	—
(f) JUnit4	J	0.096	0.098	0.118	0.2	0.176	—
	D	0.175	0.178	0.211	0.333	0.3	—
	S	1	1	1	1	0.857	—
(g) Hibernate	J	0.573	0.652	0.646	0.462	0.432	0.114
	D	0.729	0.789	0.785	0.632	0.603	0.204
	S	0.797	0.854	0.798	0.871	0.864	1

Project	Index	(ii) in multi-authored files (a)	(b)	(c)	(d)	(e)	(f)
(a) Guava		—	—	—	—	—	—
(b) Elasticsearch	J	0.673	—	—	—	—	—
	D	0.805	—	—	—	—	—
	S	0.829	—	—	—	—	—
(c) Spring Framework	J	0.657	0.783	—	—	—	—
	D	0.793	0.878	—	—	—	—
	S	0.829	0.891	—	—	—	—
(d) React Native	J	0.599	0.575	0.543	—	—	—
	D	0.749	0.73	0.703	—	—	—
	S	0.928	0.937	0.919	—	—	—
(e) JabRef	J	0.65	0.735	0.742	0.547	—	—
	D	0.788	0.847	0.852	0.708	—	—
	S	0.793	0.867	0.886	0.883	—	—
(f) JUnit4	J	0.533	0.529	0.524	0.517	0.536	—
	D	0.696	0.692	0.687	0.682	0.698	—
	S	0.857	0.884	0.893	0.685	0.866	—
(g) Hibernate	J	0.665	0.817	0.833	0.545	0.777	0.526
	D	0.799	0.899	0.909	0.705	0.874	0.689
	S	0.848	0.925	0.922	0.937	0.922	0.911

(J: Jaccard; D: Dice; S: Simpson)

Next, we explore which violations are common across projects. Table 4.10 presents the numbers of common violations across projects. While 5 violations are shown in the "NIPA > 0" row of Table 4.10, it does not mean that those violations have always positive NIPA values in all projects; they are the ones having NIPA > 0 *at least* one project.

Table 4.10: Number of violations common to all projects.

	Single-Authored	Multi-Authored
NIPA>0	0	5*
NIPA≤ 0	12	66
Total	12	71

*Number of violations whose NIPA values are positive at least one project.

As shown in Table 4.10, 71 violations commonly appear in the multi-authored files of all projects, and 66 out of 71 violations are always disregarded by the programmers (NIPA ≤ 0). The remaining 5 violations are shown in Fig. 4.2; The figure also presents the project name in which the violation's NIPA > 0.

AppendCharacterWithChar (JabRef),
AvoidUsingShortType (JabRef),
ConfusingTernary (React Native),
SimplifyBooleanReturns (React Native),
UnusedImports (JabRef)

Figure 4.2: Commonly-appearing violations which have NIPA > 0 in one project.

While five violations are considered to be important, each of them has a positive NIPA value in only one project (React Native or JabRef), and these violations are disregarded in the remaining 6 projects. That is to say, there are no commonly-important violation across projects. Figure 4.3 presents the remaining $66(= 71-5)$ violations which commonly appear in all projects and are always disregarded (NIPA ≤ 0). The all of 12 disregarded violations in the single-authored are also included in the list shown in Fig. 4.3, and those violations are emphasized in boldface. The commonly-disregarded violations shown in Fig. 4.3 correspond to about 30% of all violations. In other words, about 30% of automatically-warned violations might be worthless for many programmers. On the other hand, we did not find any violation having positive NIPA value in all projects[1]. These results would mean that critical violations vary from project to project.

[1] Although violations presented in Fig. 4.2 showed positive NIPA value in a certain project, it just means that these violations are "not commonly disregarded" ones.

AbstractClassWithoutAnyMethod, AbstractNaming, AccessorClassGeneration, AddEmptyString, ArrayIsStoredDirectly, AssignmentInOperand, **AtLeastOneConstructor**, AvoidCatchingGenericException, AvoidCatchingThrowable, AvoidDuplicateLiterals, AvoidFieldNameMatchingMethodName, AvoidInstantiatingObjectsInLoops, AvoidLiteralsInIfCondition, AvoidReassigningParameters, AvoidSynchronizedAtMethodLevel, AvoidThrowingRawExceptionTypes, AvoidUsingVolatile, **BeanMembersShouldSerialize**, BooleanGetMethodName, **CallSuperInConstructor**, ClassWithOnlyPrivateConstructorsShouldBeFinal, CommentDefaultAccessModifier, **CommentRequired**, **CommentSize**, CompareObjectsWithEquals, ConsecutiveLiteralAppends, ConstructorCallsOverridableMethod, CyclomaticComplexity, **DataflowAnomalyAnalysis**, DefaultPackage, DoNotUseThreads, EmptyCatchBlock, EmptyMethodInAbstractClassShouldBeAbstract, ExcessiveImports, ExcessivePublicCount, FieldDeclarationsShouldBeAtStartOfClass, GodClass, **ImmutableField**, InefficientStringBuffering, InsufficientStringBufferDeclaration, **LawOfDemeter**, **LocalVariableCouldBeFinal**, **LongVariable**, LooseCoupling, **MethodArgumentCouldBeFinal**, ModifiedCyclomaticComplexity, NullAssignment, **OnlyOneReturn**, PositionLiteralsFirstInComparisons, PreserveStackTrace, RedundantFieldInitializer, ShortMethodName, ShortVariable, SignatureDeclareThrowsException, StdCyclomaticComplexity, TooManyMethods, UncommentedEmptyConstructor, UncommentedEmptyMethodBody, UnnecessaryFullyQualifiedName, UnnecessaryLocalBeforeReturn, UnusedModifier, UseCollectionIsEmpty, UseConcurrentHashMap, UseUtilityClass, UseVarargs, UselessParentheses, VariableNamingConventions

Figure 4.3: Commonly-disregarded violations in all projects.

Therefore, we can answer to RQ2 as: while important violations tend to vary from project to project and from person to person, about 30% of violations would be commonly worthless across projects for many programmers. Thus, we should prepare a proper rule set of violations in accordance with the domain and organization of the project. It seems to dovetail with the previous work saying the importance of customization (flexibility) in static code analysis tools [42],[8].

4.2.5 Threats to Validity

Since the change pattern of a coding violation is decided using the number of warned parts in a source file, there might be a change of the violation but the change was masked as “sticky.” In such a case, both the increase and the decrease of the same violation momentarily occurred at that time. However, those changes were made by the same programmer and it means that he/she did not paid special attention to the violation. Hence, such a

"sticky" pattern would not have a serious impact on our results.

While we examined code changes, we are not sure if the programmers used a static code analysis tool or not during their programming activities. Thus, our results might not be well-matched with the programmers' real trends of regarding/disregarding violations. There might be latent highly-important violations or totally-trivial violations. Nonetheless, no appearance of a violation means that the violation would be rare, so our method is one of available ways to observe programmers' practices. We plan to enhance our accuracy of evaluation by analyzing more and more projects in the future.

Because of our tool (PMD) limitation, we explored Java projects only. Thus, there might be language-specific trends in our results. While many of basic coding violations and rules are common to modern procedural/object-oriented languages, we will perform similar analyses for other projects whose development languages are other than Java in the future, and prove the generality of our results.

4.3 Related Work

Shen et al. [8] proposed to leverage feedbacks from static code analysis tool's users for providing rankings of violations which are more suitable for the users, and improving the true-positive rate. Since their approach requires feedbacks from tool users, it would be hard to collect a lot of data in the case of large-scale software products. On the other hand, NIPA uses automatically-collected changes of violations instead of users' feedbacks.

Lee et al. [10] analyzed how the readability of code is affected by coding violations. While their study is useful in evaluating violations from the perspective of code readability, the work missed change history of violations over time. Kim et al. [6] proposed to prioritize violations using their lifetimes, and their focus is similar to our study. However, Kim et al. did not consider the change patterns of violations over releases. The importance of violation having the decreasing pattern would be significantly higher than the one having the increasing pattern even if those violations have the same lifetime.

4.4 Conclusion and Future Work

We examined coding violations based on code changes while considering the authorship of source file—single author or multi authors. As our criterion of violation's priority, we introduced the normalized index of programmers'

attention (NIPA) which is based on the previous work mentioned in the previous chapter. Through analyses of data collected from seven OSS projects, we proved that the difference in the authoring type has significant impacts on evaluations of violations: The variety of violations appearing in single-authored files may have a big gap with that in multi-authored files. Moreover, priorities of violations appearing in single-authored files tend to be lower than the ones in a multi-authored file. Violations caused by one programmer may be resolved by another programmer through the evolution of product.

We also investigated commonalities of violations across projects, and showed that about 30% of violations are commonly disregarded by many programmers across projects. On the other hand, we did not find commonly-important violations across projects, so important violations tend to vary from project to project and from person to person.

Our evaluation method, i.e., computing NIPA values and checking authors, can be automatically performed on a version control system like Git. By prioritizing violations based on the proposed method and the results, static code analysis tools would become more useful helps for more programmers.

Since the difference in programmers' preferences may also cause the diversity of violations, we need to focus on not only the number of developers but also individual developers in the future. Our future work includes: (1) a further analysis with data of not only Java but also other language; (2) a more detailed analysis focusing on each programmer and his/her trend of making violations.

Chapter 5

Coverage and Importance of Coding Violation

A code review is one of essential activities to ensure the quality of software. By reviewing source programs, problematic parts and/or hard-to-understand parts are improved and the quality of code would be enhanced. The quality of code review is crucial to deliver high quality software, so it is ideal to perform a careful review whenever a program is developed or modified. However, a thorough code review requires much effort and time, i.e., it is also an expensive activity. Thus, an efficient performance of code review has become a great challenge for developers [16]. To support code review activities, static code analysis tools (automated tools) have been developed. A static code analysis tool analyzes source programs and warns problematic parts or risky parts (in terms of potential faults) in accordance with predefined rulesets. For example, some open source software (OSS) development projects using GitHub have leveraged static analysis tools in checking programs (patches) submitted to those projects as pull requests [21].

However, in many cases, static analysis tools have not been widely used by developers. Indeed, a static analysis tool tends to produce a huge number of warnings but most of them are false positive ones, i.e., they are not serious points to be revised [7],[14]. It is one of key reasons why developers do not want to actively use such a tool. Therefore, toward a better utilization of static analysis tools, there have been studies in the past [5],[6],[12],[15],[9],[17],[20],[18],[19]. Spaco et al. [5], Kim et al. [6], Lee et al. [9] and Hanam et al. [12] highlighted the importance to prioritize violations through the code evolution histories in order to enhance the precision of defect alerts. They have layed down the foundation of the mechanism to utilize code evolution history for a better code review. Moreover in recent years, there has been a growth in studies to unearth the human factors through the

utilization of static analysis tools. Thongtanunam et al. [17] focused on the code authorship and the review contribution to relate the developer's contribution with defect-proneness of the code. Mello et al. [20] highlighted the importance of developer collaboration and reviewer's previous knowledges regarding faulty modules to achieve a better code review from the perspective of precise code smell detection. Avgustinov et al. [15] introduced the notion of violation fingerprint which is based on the coding behaviour of individual developer to identify which violations the developer tends to fix or disregard in his/her actual programming activities. The usefulness of their work was partially supported by Ostberg et al. [18]: they concluded that developer's personality has a big impact on the usage of static analysis tool. Yang et al. [19] established a methodology and a dataset architecture for mining code review activities in the Gerrit code review system.

While the above studies provide notable findings and proposals, they have not well argued the real trends of violations warned or cleared through upgrades of programs, so we are motivated to get a deeper insight of what have been happened in an actual programming activity. We believe that a developer's behaviour in his/her source code maintenance can be meaningful to capture critical aspects of software quality. Fortunately, a huge amount of the historical record of software development and maintenance are freely available from open repositories such as GitHub. Therefore, this chapter examines the coding violations from the perspectives of who made them and if they are related to bugfixes.

5.1 Coding Standard Violation

5.1.1 Change of Coding Standard Violations through Commits

In an OSS project, developers make or modify source files, then put them into the project's repository. The file upload (putting source files into the repository) is called "commit." Whenever a developer commits source files, the repository automatically records the commit data including the developer who made it, the committed time, the code changes and the developer's message (change log).

We can easily obtain the development history of source files and any version of source files from the repository. By using both the code repository and a static code analysis tool, we can also get the change history of coding standard violations in source files. Concretely, for each source file, we obtain all versions of the file from the repository and check them by a static code

analysis tool. Then, for each violation which appeared in a version of the file, we can see who made it, who cleared it and when they made/cleared it.

Table 5.1 presents an example of development history where 3 developers d_1, d_2 and d_3 have developed and maintained 4 source files f_1, f_2, f_3 and f_4 through 7 commits.

Suppose these source files have been warned by a static code analysis tool, and 5 violations v_1, v_2, v_3, v_4 and v_5 are observed as shown in Table 5.2. The numbers shown in each violation column of the table signify the numbers of the warned parts corresponding to the violation. For example, at commit #1, two parts in file f_1 are warned as violation v_1, and four parts are done as v_2. At commit #4, f_1 is changed then v_1 is decreased (partially cleared) as $3 \to 1$, v_2 is not changed as $4 \to 4$, and v_3 is newly made as $0 \to 2$.

Table 5.1: Example of Development History.

commit #	developer	committed file(s)	description
1	d_1	f_1, f_2	added new files
2	d_1	f_2	changed some code in f_2
3	d_2	f_2, f_3	changed some code in f_2 added new file f_3
4	d_1	f_1	fixed a bug in f_1
5	d_2	f_2	fixed a bug in f_2
6	d_3	f_3, f_4	changed some code in f_3 added new file f_4
7	d_3	f_2, f_3	fixed bugs in f_2, f_3

Table 5.2: Example of Violation Changes.

			violation				
file	commit #	developer	v_1	v_2	v_3	v_4	v_5
f_1	1	d_1	3	4	0	0	0
	4	d_1	1	4	2	0	0
f_2	1	d_1	2	0	0	4	3
	2	d_1	3	2	0	4	3
	3	d_2	3	2	0	4	3
	5	d_2	2	2	1	6	1
	7	d_3	2	2	1	6	0
f_3	3	d_2	4	0	2	0	0
	6	d_3	4	0	2	1	2
	7	d_3	4	0	1	0	3
f_4	6	d_3	2	0	1	0	1

Who Made the Violation?

In the example shown in Tables 5.1 and 5.2, all commits involving file f_1 are made by developer d_1 only. Thus, all violations which have appeared in f_1 depend on d_1's coding practice and preference. On the other hand, commits involving f_3 are made by d_2 and d_3, and appearing violations may depend on d_2 or d_3 or both.

For each commit, each developer has been related to violations as follows.

- Commit #1: d_1 made v_1, v_2, v_4 and v_5 in f_1 or f_2.
- Commit #2: d_1 made v_2 in f_2.
- Commit #3: d_2 changed f_2 but the coding violations stayed; d_2 made v_1 and v_3 in f_3.
- Commit #4: d_1 decreased (partially cleared) v_1 as $3 \rightarrow 1$, but made another violation, v_3, in f_1.
- Commit #5: d_2 decreased v_1 and v_5, but made v_3 and increased v_4, in f_2.
- Commit #6: d_3 made v_4 and v_5 in f_3, and did v_1, v_3 and v_5 in f_4.
- Commit #7: d_3 decreased (cleared) v_5 as $1 \rightarrow 0$ in f_2; d_3 decreased v_3, but increased v_5, in f_3

Through the above 7 commits, d_1 newly made or increased all 5 violations, d_2 did 3 violations (v_1, v_3 and v_4), and d_3 did 4 violations (v_1, v_3, v_4 and v_5), respectively.

As presented above, we can observe who made the violation through commits.

Is the Violation Related to Bugs?

Next, we consider the relationship of violations with latent bugs. It is not always true that all coding standard violations are related to the code quality, especially, the bug-proneness. The strength of the relationship can be seen from the perspective of the violation change history. If the warning count of a violation in a source file is decreased when a bugfix was made for the source file, the violation seems to be related to the fixed bug. On the other hand, if the warning count is increased through a bugfix, the corresponding violation would not be related to the fixed bug.

In the example shown in Table 5.1, commits #4, #5 and #7 are aimed at bugfixes. We can see the follows from Table 5.2.

- In f_1: v_1 is decreased at commit #4, but v_3 is increased at that commit.
- In f_2: while v_1 and v_5 are decreased, v_3 and v_4 are increased at commit #5; v_5 is decreased (cleared) at commit #7.
- In f_3: v_3 and v_4 are decreased at commit #7, but v_5 is increased at that commit.
- In f_4: there is no bugfix commit.

Through 3 bugfix commits (#4, #5 and #7), v_1 experiences 2 decrease events, v_3 does 1 decrease event and 2 increase ones, v_4 does 1 decrease event and 1 increase one, and v_5 does 2 decrease events and 1 increase one, respectively; v_2 did not change through these bugfix commits.

By checking the above increases and decreases of violations through commits, we can see which violation is stronger related to bugs.

5.1.2 Research Questions

By observing changes of coding violations through commits, we can see the following two things: 1) who makes which violations, and 2) which violations are related to bugs.

When a violation has been made by a certain developer, the violation seems to depend only on the developer. On the other hand, when a violation has been made by more developers, it would be a familiar violation which may appear more frequently. Since the former type of violation is developer-specific, the data analysis and discussion about such a violation would not be attractive to many researchers and practitioners. Hence, we will focus on the latter (more frequently-appearing) violations in this chapter, and analyze their importance from the perspective of the bug-proneness. Now we set up the following research questions (RQs) to clarify the aims of our empirical analysis:

- RQ1: Which coding violations are familiar with more developers and more important in preventing bugs?
- RQ2: Does the difference of project cause differences in the familiarity and the importance of a violation?

By analyzing violation data for RQ1, we will be able to discriminate noteworthy violations from worthless ones in the set of familiar violations. It would be a practical support of an effective code review by using a static code analysis tool.

RQ2 is a question about the generality of findings which would be derived from RQ1. If trends of violations have a high commonality among different projects, we can prioritize the coding violations as a general guideline. Otherwise, we would have to tune the assessments of violations to each project, i.e., it is better to build a project-specific evaluation model of violations. The aim of RQ2 is to see if we can produce a general guideline or not; if a project-specific way is better.

5.1.3 Metrics

In order to collect quantitative data for answering the RQ1 and RQ2, we define metrics for measuring the familiarity and the importance of a violation.

Violation Coverages

Suppose s source files have been developed and maintained by p developers, and w violations v_i (for $i = 1, \ldots, w$) have appeared in those source files. We quantify the familiarity of violation v_i from two different perspectives—the file coverage and the developer coverage.

The following metric $\mathrm{FC}(v_i)$ is the file coverage of v_i, which expresses how widely v_i appears in source files:

$$\mathrm{FC}(v_i) = \frac{n_s(v_i)}{s}, \tag{5.1}$$

where $n_s(v_i)$ is the number of source files which have experiences with being warned as violation v_i.

The following metric $\mathrm{DC}(v_i)$ is the developer coverage of v_i, which signifies how widely v_i is linked to developers:

$$\mathrm{DC}(v_i) = \frac{n_p(v_i)}{p}, \tag{5.2}$$

where $n_p(v_i)$ is the number of developers who have experiences with making or increasing warnings of v_i.

Both $\mathrm{FC}(v_i)$ and $\mathrm{DC}(v_i)$ range $[0, 1]$.

In the example of Table 5.2, we have $s = 4$ and $p = 3$. Let us take an example of v_5: Since v_5 had appeared in 3 files (f_2, f_3 and f_4), $\mathrm{FC}(v_5) = 3/4$; Because v_5 had been made or increased by 2 developers (d_1 and d_3), $\mathrm{DC}(v_5)$ $= 2/3$. Consequently, these metrics are computed shown as Table 5.3.

When a violation has both a high FC value and a high DC value, the violation widely appears in source files and is familiar with more developers.

Table 5.3: FC(v_i) and DC(v_i) for Table 5.2.

v_i	files	FC(v_i)	developers	DC(v_i)
v_1	f_1, f_2, f_3, f_4	4/4	d_1, d_2, d_3	3/3
v_2	f_1, f_2	2/4	d_1	1/3
v_3	f_1, f_2, f_3, f_4	4/4	d_1, d_2, d_3	3/3
v_4	f_2, f_3	2/4	d_1, d_2, d_3	3/3
v_5	f_2, f_3, f_4	3/4	d_1, d_3	2/3

Violation Importance

As mentioned above, if the warning count of violation v_i decreased through a bugfix, v_i seems to be related to the fixed bug, and it would be an important violation for preventing a bug inducing. On the other hand, if the warning count of v_i increased through a bugfix, v_i seems to be less important. By focusing on the difference between the decreased count and the increased one, we define the following metric IMP(v_i) which presents an importance of v_i:

$$\mathrm{IMP}(v_i) = \frac{n_{dec}(v_i)}{\sum_{j=1}^{w} n_{dec}(v_j)} - \frac{n_{inc}(v_i)}{\sum_{j=1}^{w} n_{inc}(v_j)}, \tag{5.3}$$

where $n_{dec}(v_i)$ and $n_{inc}(v_i)$ signify the numbers of bugfix commits in which the warning count of v_i decreased and increased, respectively.

Since the total number of the decrease commits would differ from the total number of the increase commits, we define the above metric by using the normalized values instead of raw counts. Intuitively, it is like an evaluation of reactions to a news by using the number of “thumbs up” and that of “thumbs down” which we often see at a news Website such as Yahoo.com.

In the example of Tables 5.1 and 5.2, the bugfix commits are commit #4, #5 and #7. According to Eq.(5.3), we get the importance values of violations shown in Table 5.4: for example, the data corresponding to v_5 is obtained as follows.

- Commit #4: no change.
- Commit #5: the warning counts in f_2 decreased as $3 \rightarrow 1$.
- Commit #7: while the warning counts in f_2 decreased as $1 \rightarrow 0$, the counts in f_3 increased as $2 \rightarrow 3$.

A higher value of IMP(v_i) represents that v_i has a stronger link to a latent bug. Such link data can be an empirical basis toward an effective code review with using a static code analysis tool.

Table 5.4: IMP(v_i) for Table 5.2.

v_i	bugfix commits #4	#5	#7	$n_{dec}(v_i)$	$n_{inc}(v_i)$	IMP(v_i)
v_1	−	−		2	0	1/3
v_2				0	0	0
v_3	+	+	−	1	2	−1/3
v_4		+	−	1	1	−1/12
v_5		−	+,−	2	1	1/12
total				6	4	

(−: decrease of the warning; +: increase of the warning)

5.2 Empirical Analysis

5.2.1 Aim and Data Collection

To answer the above RQs, we collect a lot of actual data from OSS development projects and analyze the coding violations which had appeared in the source files.

In order to ensure the generality and usefulness of our empirical results, we randomly selected six popular OSS projects from the GitHub, shown in Table 5.5. The development language in all of them is Java. This language limitation is from the static code analysis tool, PMD, we used. Singh et al. [21] emphasizes that PMD provides a time benefit to the reviewer if it has been used prior to pull request submission, since violations detected by PMD overlapped with nearly 16% of the reviewer comments. This is the reason why we used PMD in this study.

Table 5.5: Investigated OSS projects.

project	data collection period	number of developers	number of files
Elasticsearch	Feb. 2010 – May. 2017	178	3,403
Guava	Jan. 2010 – May. 2017	174	1,933
JabRef	Dec. 2011 – Apr. 2017	37	1,029
JUnit4	Dec. 2004 – Dec. 2014	124	208
React Native	Mar. 2015 – May. 2017	299	650
Spring Framework	Dec. 2008 – May. 2017	170	3,958

For each project, we collect data in the following procedure.

1. Make a local copy (clone) of the Git repository.

2. For each source file[1] stored in the repository, investigate its change history through the commit log. Then, for each version of each source file, get (check out) the content and check it by PMD (ver.5.4.1). Use all rule sets for the PMD checking.

3. Classify all commits into bugfix commits and non-bugfix ones by checking whether their commit messages include a bugfix-related keyword or not [46].

4. For each commit, determine the developer who made it: referred to as “author” in Git repository. Since some developer use two or more different names (IDs) or email addresses, identify them according to the following rules [39]: 1) if two authors have different names but the same email address, they are the same author; 2) if two authors have different email addresses but the same name, they are the same author. Moreover, we also performed manual checks because there may also be an abbreviated form of name.

5. Store data obtained in the above steps 2), 3) and 4) into a database, and organize them in forms like Tables 5.1 and 5.2. Then, compute metric data by Eqs. (5.1),(5.2) and (5.3).

5.2.2 Results

Since we have no absolute threshold of a high coverage, we will decide it in accordance with the data distribution. Now we consider the median of coverage to be a threshold, and define that a violation has a relatively-high coverage if its FC value and DC value are greater than their medians, respectively. (see the partition “C_{HH}” in Fig.5.1)

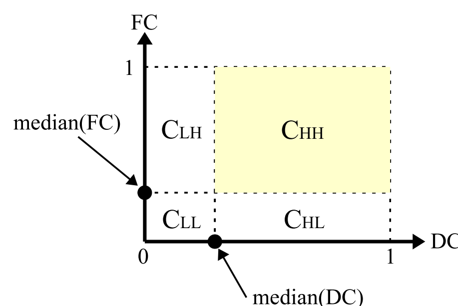


Figure 5.1: Partitions of violations according to their coverage metrics.

[1]Programs for testing are excluded. We regard a source file as a test code if its file path includes “junit” or “test” (case insensitive).

As a part of our results, the computed IMP values of violations which belong to C_{HH} of Elasticsearch are shown in Table 5.6, in decreasing order of IMP value. Similarly, we computed IMP values of violations belong to C_{HH} of each project, and got many violations which differ among projects. Table 5.7 shows the number of violations classified by "IMP > 0" and "IMP ≤ 0."

Table 5.6: The most important violations and the least important ones which belong to C_{HH} in Elasticsearch.

rank	violation
1	UnusedImports
2	LocalVariableCouldBeFinal
3	AvoidFieldNameMatchingMethodName
4	DataflowAnomalyAnalysis
5	LawOfDemeter
6	PrematureDeclaration
7	AvoidPrefixingMethodParameters
8	AvoidFinalLocalVariable
9	ImmutableField
10	ConfusingTernary
⋮	⋮
83	AvoidReassigningParameters
84	OnlyOneReturn
85	DefaultPackage
86	CommentRequired
87	CommentDefaultAccessModifier
88	ShortVariable
89	SimplifyBooleanExpressions
90	LongVariable
91	MethodArgumentCouldBeFinal
92	CommentSize

Table 5.7: Number of violations which belong to C_{HH} in each project.

	number of violations	
project	IMP > 0	IMP ≤ 0
Elasticsearch	50	42
Guava	33	47
JabRef	21	26
JUnit4	23	25
React Native	34	32
Spring Framework	26	52

As a consequence, only 25 violations are common to all six projects. Table 5.8 presents these violations. In the table, symbol "✓" signifies that the corresponding violation has positive IMP value (IMP > 0) in the corresponding project. The order of violations is the decreasing order of "✓" count.

Table 5.8: Common violations which belong to C_{HH} in all projects.

	project					
violation	E	G	Ja	JU	R	S
UnusedImports	✓	✓	✓	✓	✓	✓
DataflowAnomalyAnalysis	✓	✓	✓	✓	✓	
LawOfDemeter	✓			✓	✓	✓
AccessorMethodGeneration		✓	✓		✓	✓
LocalVariableCouldBeFinal	✓	✓			✓	
AvoidInstantiatingObjectsInLoops	✓	✓	✓			
AvoidCatchingGenericException	✓			✓	✓	
AtLeastOneConstructor	✓			✓	✓	
AvoidLiteralsInIfCondition	✓	✓			✓	
BeanMembersShouldSerialize	✓	✓				✓
UseConcurrentHashMap		✓	✓		✓	
OnlyOneReturn		✓		✓	✓	
CommentSize		✓			✓	✓
ConfusingTernary	✓			✓		
EmptyCatchBlock	✓					✓
GodClass		✓	✓			
UselessParentheses			✓	✓		
DefaultPackage		✓	✓			
TooManyMethods	✓					
CommentDefaultAccessModifier				✓		
ShortVariable					✓	
LongVariable						✓
MethodArgumentCouldBeFinal		✓				
CallSuperInConstructor						
CommentRequired						

There is only one violation whose IMP value is always positive among projects, "UnusedImports." On the other hand, there are only two violations whose IMP values are always non-positive, "CallSuperInConstructor" and "CommentRequired." That is to say, even if we focus only on commonly familiar (higher coverage) violations, their trends of importance differ from project to project.

5.2.3 Discussions

In this study, we investigated the changes of coding violations through commits in six popular OSS projects, and evaluated their coverage and importance by using metrics FC, DC and IMP. As a result, higher-coverage (familiar) coding violations tend to differ from project to project, and only 25 violations are common. Moreover, their trends of importance—if IMP value is positive or not—also vary among projects: only one violation always shows IMP > 0 and only two violations always have IMP ≤ 0.

The violation corresponding to always positive IMP value is "UnusedImports." Since this is based on the notion "avoid unused import statements," it does not seem to be a direct cause of bug. This violation might be cleared by reorganizing code when a bug is fixed.

Next, we consider commonly-disregarded violations "CallSuperInConstructor" and "CommentRequired." CallSuperInConstructor warns a class if its all constructors do not call super(). While the PMD official site says "it is a good practice to call super() in a constructor," the need to call super() depends on the class design, so it may be less related to bugfixes. CommentRequired warns a class, a protected method, a public method or a filed which has no comment. Because this violation is in C_{HH} of all projects, it seems to be a real trend that many programmers may skip describing comments for one of the above elements in their programs. While well-commented programs are easier to understand, it is not always true that non-commented programs are harmful. Moreover, there are studies saying that well-commented programs may be more complicated ones and consequently more fault-prone[26]. That is to say, it might be reasonable that these two types of violations are not linked to bugfixes.

Let us answer to our RQs. While we found commonly-familiar 25 violations, their importance vary from project to project as shown in Table 5.7. Since there are only a few violations which are commonly-important or commonly-worthless, it would be better to tune the priorities of violations to the project, and build a project-specific evaluation model by using the feedback from its change history of violations.

5.2.4 Threats to Validity

In our data collection, we linked the code changes to fixed bugs if the commit is aimed at a bugfix. However, other types of code changes might also be mixed as we discussed in regard to "UnusedImports." They would be noise in our data analysis. While it is not easy to examine the detailed reasons of all code changes, a further analysis of code changes' aims would be essential

to accurately analyze the data. It is our significant future work.

While we used PMD as our static code analysis tool, other tools may warn different violations. However, for the same programming language, the programming practices and styles would not differ widely. Moreover, since we used all rule sets prepared in PMD, many of the same or similar violations would be covered. Thus, the difference of tool is not a serious threat to validity in this study.

5.3 Related Work

Hanam et al. [12] stated that a utilization of the source code history on the repository can be useful in projecting trends of alerts (warnings) on the source code, and thus be beneficial for predicting the source code evolution. We are also aware of the importance of the history data stored in the repository. While Hanam et al. focused on the code evolution, we leveraged such data for analyzing the coding violations.

Avgustinov et al. [15] proposed a developer fingerprint to reveal an individual developer's coding habit using the code change history and their coding violations. Although their approach is useful to know developer-specific trends of coding violations, it was not linked to a bugfix. By combining their approach with our method, we would be able to produce a developer-specific bug prediction mechanism based on their coding violation trend. We would like to study such an approach as well.

Kim and Ernst [6] prioritized warnings based on their lifetime and revealed that around 10% of warnings that related to bug fixing activities. While their results are partially supported by our results, we considered not only the relationships with bugfixes but also the familiarity of developers to warnings.

5.4 Conclusion

We focused on coding standard violations warned by a static code analysis tool, and proposed to evaluate them from the perspective of the familiarity—a file coverage and a developer coverage—and the importance—a strength of relationships with bugfixes. Since violations having high coverage are familiar with more programmers and frequently appear in many source files, it is useful to understand which ones of those familiar violations are important to prevent a bug inducing.

We conducted an empirical analysis of coding violations appearing in

six popular OSS projects. As a result, familiar violations seemed to differ among projects, and only 25 violations were common to all surveyed projects. Moreover, the trends of their importance varied from project to project. Therefore, we found that it is better to tune the assessments of violations for each project, rather than to build a general guideline of coding violations. Toward an effective code review, it would be useful to collect the change history of violations and to produce a project-specific evaluation model of violations. Since our method for evaluating violations can be automatically performed, we plan to apply our method to a just-in-time bug prediction by computing importance of violation from their change history in the future. A further analysis of detailed link between a bugfix and a violation change is also our significant future work.

Chapter 6

Conclusion

Since programming activity is an intelligent task by human beings, human factors would play important roles in the development of quality software systems. This dissertation explores the influence of human factors on the programming activity. As measurable features in programming activities, we focus on "comments" and "coding violations."

Comments are artifacts that programmers freely describe in their programs. Since it can vary from programmer to programmer how many comments they write, we analyze the impact of individual difference in writing comments.

Coding violations present various ways of observing programmers' preferences and habits. So we performed several analyses focusing on coding violations.

The contributions are summarized as follows.

The first research stage (Chapter 2): this study focused on differences in comment densities among individual programmers, and proposed to adjust the conventional code complexity metric (the cyclomatic complexity) by using the abnormality of the comment density. This empirical study with nine popular open source Java products including 103,246 methods showed that the proposed metric performs better than the conventional one in predicting change-prone methods; the proposed metric improved the area under the ROC curve (AUC) by about 3.4% on average. Thus it can be beneficial in a change-prone method prediction.

The second stage of research (Chapter 3): this study focused on coding violation trend across all version of project. In order to analyze the automatically pointed violations and the actual attentions which programmers paid to those violations, this study proposed a novel metric—the Index of Programmers Attention (IPA)—and conducted an empirical study focusing on the change patterns of violations over the releases of popular seven open source

software products, under two research questions (RQs): (RQ1) What kind of coding violations are related to the parts that many programmers tend to improve? and what kind of coding violations are likely to be disregarded?; (RQ2) How can we reduce the meaningless violations for programmers by omitting disregarded coding violations? The empirical results showed the following findings: (1) important violations (having high IPA values) may vary from project to project; (2) there are some unimportant violations common to different projects, but they are a minority of automatically detected violations (about 12%). Therefore, while many violations may be made by a code checker, most of them are likely to be worthy in improving the code quality, and it is ineffective to reduce the violations by eliminating such unimportant violations

The third stage (Chapter 4): this study examined an impact of authorship on those violation evaluations because a preference of a certain programmer may have an affect on a creation or modification of violation. This study collected violations made by a popular static code analysis tool, PMD, from seven open source software projects. The set of collected data was divided into two subsets according to the authorship of source file: the set of violations appearing in source files which have been developed and maintained by a single programmer (single-authored files) vs. the set of ones appearing in source files which have been done by two or more programmers (multi-authored files). The normalized index of programmers' attention (NIPA) was introduced as an assessment criterion for violation prioritization. The results of data analyses showed the following findings: (1) the difference in the authoring type has significant impacts on evaluations of violations which is shown as the variety of violations appearing in single-authored files may have a big gap with that in multi-authored files. Moreover, priorities of violations appearing in single-authored files tend to be lower than the ones in a multi-authored file. Violations caused by one programmer may be resolved by another programmer through the evolution of product; (2) while important violations tend to vary from project to project and from person to person, about 30% of violations would be commonly worthless across projects for many programmers.

The forth stage of research (Chapter 5) was an in-depth investigation at commit level of the repository, which identify individual programmer who made code changes. This study conducted an empirical analysis focusing on the coverage and importance of coding violation, using six popular OSS projects. That is to say, this paper investigated which violation is familiar with more programmers and frequently appears in many source files (having a high coverage), and which violation is really related to bugfixes (having a high importance). The empirical results showed the following findings: (1)

the familiar violations tend to differ among projects, and only 25 violations are common to all surveyed projects; (2) the trends of their importance vary from project to project. Therefore, this study concluded that it is better to tune the assessments of violations for each project by collecting its change history of violations in order to accurately understand bug-related warnings, rather than to build a general guideline.

The above findings emphasizes the importance of focusing on human factors in the successful software development. While there have been many studies providing useful guidelines and metrics for software developers in the software engineering world, our research results showed that we should focused on individual differences as well and utilized them to improve the conventional guidelines and metrics. By combining well-defined general guidelines with a mechanism of automatic tuning based on the individual programmer's data, such as "comments" and "coding preferences/habits" we presented, we would obtain a more sophisticated software development environment.

We plan to implement support systems based on our empirical results in the future.

Bibliography

[1] B. Crawford, R. Soto, C. L. de la Barra, K. Crawford, and E. Olguín, "The influence of emotions on productivity in software engineering," in *HCI International 2014 - Posters' Extended Abstracts*, C. Stephanidis, Ed. Cham: Springer International Publishing, 2014, pp. 307–310.

[2] D. Graziotin, X. Wang, and P. Abrahamsson, "Happy software developers solve problems better: psychological measurements in empirical software engineering," *PeerJ*, vol. 2, p. e289, Feb. 2014. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3961150/

[3] L. F. Capretz and F. Ahmed, "Making sense of software development and personality types," *IT Professional*, vol. 12, no. 1, pp. 6–13, Jan 2010.

[4] D. Viana, T. Conte, D. Vilela, C. R. B. de Souza, G. Santos, and R. Prikladnicki, "The influence of human aspects on software process improvement: Qualitative research findings and comparison to previous studies," in *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*, May 2012, pp. 121–125.

[5] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proc. 2006 International Workshop on Mining Software Repositories*, May 2006, pp. 133–136.

[6] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 45–54.

[7] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proc. International Conference on Software Testing Verification and Validation*, April 2009, pp. 141–150.

[8] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *Proc. 2011 4th IEEE International Conference on Software Testing, Verification and Validation*, Mar. 2011, pp. 299–308.

[9] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study of the effect of file editing patterns on software quality," in *Proc. 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 456–465.

[10] T. Lee, J. B. Lee, and H. P. In, "A study of different coding styles affecting code readability," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 5, pp. 413–422, 2013.

[11] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[12] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proc. 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 152–161. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597100

[13] Y. Qiu, W. Zhang, W. Zou, J. Liu, and Q. Liu, "An empirical study of developer quality," in *Proc. 2015 IEEE International Conference on Software Quality, Reliability and Security – Companion*, Aug 2015, pp. 202–209.

[14] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Proc. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, March 2015, pp. 161–170.

[15] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *Proc. 37th International Conference on Software Engineering*, May 2015, pp. 437–447.

[16] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *Proc. 2015 IEEE International Conference on Software Maintenance and Evolution*, Sept 2015, pp. 111–120.

[17] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Revisiting code ownership and its relationship with software quality in the scope of modern code review,” in *Proc. IEEE/ACM 38th International Conference on Software Engineering*, May 2016, pp. 1039–1050.

[18] J. P. Ostberg, S. Wagner, and E. Weilemann, “Does personality influence the usage of static analysis tools? an explorative experiment,” in *Proc. IEEE/ACM Cooperative and Human Aspects of Software Engineering*, May 2016, pp. 75–81.

[19] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, “Mining the modern code review repositories: A dataset of people, process and product,” in *Proc. IEEE/ACM 13th Working Conference on Mining Software Repositories*, May 2016, pp. 460–463.

[20] R. M. d. Mello, R. F. Oliveira, and A. F. Garcia, “On the influence of human factors for identifying code smells: A multi-trial empirical study,” in *Proc. 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Nov. 2017, pp. 68–77.

[21] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, “Evaluating how static analysis tools can reduce code review effort,” in *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, Oct 2017, pp. 101–105.

[22] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, “A study of the documentation essential to software maintenance,” in *Proc. 23rd International Conf. Design of Communication*, Sept. 2005, pp. 68–75.

[23] D. Steidl, B. Hummel, and E. Juergens, “Quality analysis of source code comments,” in *Proc. 21st International Conf. Program Comprehension*, May 2013, pp. 83–92.

[24] R. P. Buse and W. R. Weimer, “A metric for software readability,” in *Proc. 2008 International Symposium on Software Testing and Analysis*, July 2008, pp. 121–130.

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

[26] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, “Lines of comments as a noteworthy metric for analyzing fault-proneness in methods,” *IEICE Transactions on Information and Systems*, vol. E98-D, no. 12, pp. 2218–2228, Dec. 2015.

[27] ——, “Empirical analysis of fault-proneness in methods by focusing on their comment lines,” in *Proc. 21st Asia-Pacific Software Engineering Conference*, vol. 2, Dec. 2014, pp. 51–56, (presented at the 2nd International Workshop on Quantitative Approaches to Software Quality (QuASoQ2014) co-located with APSEC2014).

[28] ——, “Empirical analysis of change-proneness in methods having local variables with long names and comments,” in *Proc. 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2015, pp. 50–53.

[29] T. J. McCabe, “A complexity measure,” *IEEE Trans. Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[30] “Angry IP Scanner,” http://angryip.org/, accessed Aug.26. 2016.

[31] “Eclipse Checkstyle Plugin,” http://eclipse-cs.sourceforge.net/, accessed Aug.26. 2016.

[32] “eXo Platform,” https://www.exoplatform.com/, accessed Aug.26. 2016.

[33] “FreeMind,” http://freemind.sourceforge.net/, accessed Aug.26. 2016.

[34] “GNU ARM Eclipse Plug-ins,” http://gnuarmeclipse.github.io/, accessed Aug.26. 2016.

[35] “Hibernate ORM,” http://hibernate.org/orm/, accessed Aug.26. 2016.

[36] “PMD,” https://pmd.github.io/, accessed Aug.26. 2016.

[37] “ProjectLibre,” http://www.projectlibre.org/, accessed Aug.26. 2016.

[38] “SQuirreL SQL Client,” http://squirrel-sql.sourceforge.net/, accessed Aug.26. 2016.

[39] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, “Mining email social networks,” in *Proc. 2006 International Workshop on Mining Software Repositories*, ser. MSR ’06. New York, NY, USA: ACM, May 2006, pp. 137–143.

[40] A. Agresti, *Categorical Data Analysis*. N.J.: Wiley Interscience, 2002.

[41] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 2013 International Conference on Software Engineering*, May 2013, pp. 672–681.

[42] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Proc. IEEE International Conf. Softw. Maintenance*, Sept. 2008, pp. 277–286.

[43] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[44] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proc. 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 202–212.

[45] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 2013 35th International Conference on Software Engineering*, May 2013, pp. 672–681.

[46] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. 2005 International Workshop on Mining Software Repositories*, May 2005, pp. 1–5.