ATLANTIS PRESS International Journal of Networked and Distributed Computing, Vol. 5, No. 4 (October 2017) 211–220

An Evaluation of Coding Violation Focusing on Change History and Authorship of Source File*

Aji Ery Burhandenny [1,2] , Hirohisa Aman [3] , Minoru Kawahara [3]

[1] *Graduate School of Science and Engineering, Ehime University*
3, Bunkyo-cho, Matsuyama, Ehime 790–8577, Japan

[2] *Engineering Faculty, Mulawarman University*
Samarinda, East Kalimantan 75119, Indonesia

E-mail: a.burhandenny@ft.unmul.ac.id

[3] *Center for Information Technology, Ehime University*
3, Bunkyo-cho, Matsuyama, Ehime 790–8577, Japan

E-mail: {aman, kawahara}@ehime-u.ac.jp

Abstract

This paper focuses on an evaluation of coding violation warned by a static code analysis tool while considering the change history of violation and the authorship of source file. Through an empirical study with data collected from seven open source software projects, the following findings are reported: (1) the variety and the evaluation of a coding violation tend to vary accroding to the authoring type of the source file; (2) while important violations tend to vary from project to project, about 30% of violations are commonly disregarded by many programmers.

Keywords: Static code analysis, coding violation, authorship of source file, programmer's attention.

1. Introduction

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[1]. In order to reduce the risk of latent faults, it is effective to perform a careful code review[2]. 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

* An earlier version of this paper was presented at The 2nd International Conference on Big Data, Cloud Computing, and Data Science Engineering (BCD2017).

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[3]. 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. Aji et al.[4] focused on change patterns of violations over releases, and proposed a metric for evaluating violations, "Index of Programmers' Attention (IPA)." 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 literature[4], 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 paper will examine the impact of authorship on the above evaluation of violations and report the results of analysis.

The remainder of this paper is organized as follows: Section 2 presents the way of evaluating violations and our main focus. Section 3 reports our empirical study along with our discussions, and Sect.4 briefly describes the related work. Finally, Sect.5 gives the conclusions and our future plan.

2. Evaluation of Violation and Authorship of Source File

2.1. *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, Aji et al.[4] focused on change patterns of violations over releases (see Fig. 1): (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.

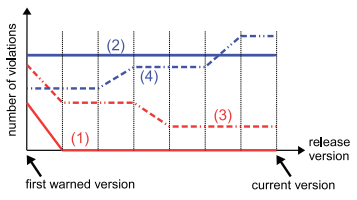

Fig. 1. Four change patterns of violations.

Aji et al. proposed the following metric, "Index of Programmers' Attention (IPA)" using these notions: For a violation (warning) v,

$$\mathrm{IPA}(v) = \frac{N_o(v) + N_d(v)}{N_s(v) + N_i(v)}\ , \tag{1}$$

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 $N_o(v) + N_d(v) + N_s(v) + N_i(v) = 0$.

2.2. *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. (1)—since IPA value can be ∞. Thus, we introduce the normalized IPA (NIPA) as follows:

$$\begin{aligned}&\mathrm{NIPA}(v)\\ &= \frac{\{N_o(v) + N_d(v)\} - \{N_s(v) + N_i(v)\}}{N_o(v) + N_d(v) + N_s(v) + N_i(v)}\ .\end{aligned} \tag{2}$$

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) $\mathrm{NIPA}(v) = -1$: $N_o(v) + N_d(v) = 0$ 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) $\mathrm{NIPA}(v) > 0$: $N_o(v) + N_d(v) > N_s(v) + N_i(v)$. 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) $\mathrm{NIPA}(v) \leqslant 0$: $N_o(v) + N_d(v) \leqslant N_s(v) + N_i(v)$. This is opposite to (iii), so v would be relatively disregarded by programmers.

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

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?

3. Empirical Study

3.1. *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 1 shows the projects surveyed in this study. These projects are randomly selected from GitHub. By comparing NIPA values of coding

Table 1. 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

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.

3.2. 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[5]:

   (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.

3.3. 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.

3.3.1. 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:

$$Jac(V_s, V_m) = \frac{|\, V_s \cap V_m \,|}{|\, V_s \cup V_m \,|} \,. \tag{3}$$

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

$$Dice(V_s, V_m) = \frac{2\,|\, V_s \cap V_m \,|}{|V_s| + |V_m|} \,. \tag{4}$$

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

$$Simp(V_s, V_m) = \frac{| V_s \cap V_m |}{\min\{ |V_s|, |V_m| \}} . \quad (5)$$

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 2 shows the similarities.

Table 2. 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 3).

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 3. Number of elements in V_s, V_m and $V_s \cap V_m$.

Project	$\|V_s\|$	$\|V_m\|$	$\|V_s \cap V_m\|$
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

3.3.2. 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 summarizes the numbers of violations according to their NIPA values and their authoring types. In the table, the number of violations whose NIPA values are positive—being considered to be important—is

Table 4. Number of violations according to their NIPA values and authoring types.

		Authoring Type	
Project	NIPA	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**	

Table 5. 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)

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 5 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.

In Table 5, 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 de-

pends 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 6 shows the results. In Table 6, 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 6. 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:

$$\Delta\text{NIPA}(v) = \text{NIPA}_{single}(v) - \text{NIPA}_{multi}(v)\ , \quad (6)$$

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 7 shows the distributions of $\Delta\text{NIPA}(v)$. While there are a few extreme cases ($\Delta\text{NIPA}(v) = -2$ or 2), ΔNIPA values of most violations are less than or equal to zero. Figure 2 presents boxplots of ΔNIPA values, which focus only on around zero. From Fig. 2, 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 7. Distributions of $\Delta\text{NIPA}(v)$.

		Percentile			
Project	Min.	25%	50%	75%	Max.
G	−2.000	−0.082	0	0	0.865
E	−0.670	−0.051	−0.0005	0.002	2.000
S	−0.371	−0.091	−0.008	0	0.377
R	−0.422	−0.098	0	0	2.000
JR	−0.958	−0.286	−0.151	−0.053	2.000
J4	−0.383	−0.150	−0.095	−0.036	0
H	−0.717	−0.081	−0.014	0	2.000

(G: Guava; E: Elasticsearch; S: Spring Framework; R: React Native; JR: JabRef; J4: JUnit4; H: Hibernate)

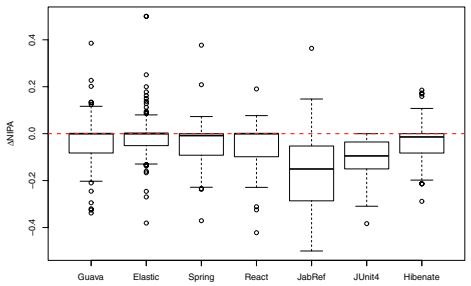

Fig. 2. Boxplots of $\Delta\text{NIPA}(v)$ (focused only on the range $[-0.5, 0.5]$).

From the all results shown in this subsection (Sect. 3.3), we can answer to RQ1 as: the difference in the authoring type has significant impacts on the trends of violations and their evaluations.

3.4. *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 8 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,

Table 8. Similarities of appearing violatins among projects.

Project	Index	(i) in single-authored files (a)	(b)	(c)	(d)	(e)	(f)	(ii) in multi-authored files (a)	(b)	(c)	(d)	(e)	(f)
(a) Guava		—	—	—	—	—	—	—	—	—	—	—	—
(b) Elasticsearch	*J*	0.606	—	—	—	—	—	0.673	—	—	—	—	—
	D	0.754	—	—	—	—	—	0.805	—	—	—	—	—
	S	0.762	—	—	—	—	—	0.829	—	—	—	—	—
(c) Spring Framework	*J*	0.514	0.638	—	—	—	—	0.657	0.783	—	—	—	—
	D	0.679	0.779	—	—	—	—	0.793	0.878	—	—	—	—
	S	0.756	0.857	—	—	—	—	0.829	0.891	—	—	—	—
(d) React Native	*J*	0.403	0.449	0.443	—	—	—	0.599	0.575	0.543	—	—	—
	D	0.574	0.62	0.614	—	—	—	0.749	0.73	0.703	—	—	—
	S	0.886	0.762	0.829	—	—	—	0.928	0.937	0.919	—	—	—
(e) JabRef	*J*	0.333	0.375	0.445	0.432	—	—	0.65	0.735	0.742	0.547	—	—
	D	0.5	0.545	0.616	0.603	—	—	0.788	0.847	0.852	0.708	—	—
	S	0.803	0.864	0.864	0.621	—	—	0.793	0.867	0.886	0.883	—	—
(f) JUnit4	*J*	0.096	0.098	0.118	0.2	0.176	—	0.533	0.529	0.524	0.517	0.536	—
	D	0.175	0.178	0.211	0.333	0.3	—	0.696	0.692	0.687	0.682	0.698	—
	S	1	1	1	1	0.857	—	0.857	0.884	0.893	0.685	0.866	—
(g) Hibernate	*J*	0.573	0.652	0.646	0.462	0.432	0.114	0.665	0.817	0.833	0.545	0.777	0.526
	D	0.729	0.789	0.785	0.632	0.603	0.204	0.799	0.899	0.909	0.705	0.874	0.689
	S	0.797	0.854	0.798	0.871	0.864	1	0.848	0.925	0.922	0.937	0.922	0.911

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

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.

Next, we explore which violations are common across projects. Table 9 presents the numbers of common violations across projects. While 5 violations are shown in the "NIPA > 0" row of Table 9, 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.

As shown in Table 9, 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 $\leqslant 0$). The remaining 5 violations are shown in Fig. 3; The figure also presents the project name in which the violation's NIPA > 0.

Table 9. Number of violations common to all projects.

	Single-Authored	Multi-Authored
NIPA> 0	0	5*
NIPA$\leqslant 0$	12	66
Total	12	71

*Number of violations whose NIPA values are positive at least 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 presents the remaining $66 (= 71 - 5)$ violations which commonly appear in all projects and are always disregarded (NIPA $\leqslant 0$). The all of 12 disregarded violations

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

Fig. 3. Commonly-appearing violations which have NIPA > 0 in one project.

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

Fig. 4. Commonly-disregarded violations in all projects.

in the single-authored are also included in the list shown in Fig. 4, and those violations are emphasized in boldface. The commonly-disregarded violations shown in Fig. 4 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.† These results would mean that critical violations vary from project to project.

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[6,7].

3.5. 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. Related Work

Shen et al.[6] 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

† Although violations presented in Fig. 3 showed positive NIPA value in a certain project, it just means that these violations are "not commonly disregarded" ones.

of large-scale software products. On the other hand, NIPA uses automatically-collected changes of violations instead of users' feedbacks.

Lee et al.[8] 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.[9] 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.

5. 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[4]. 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.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 16K00099 and DIKTI Scholarships, Directorate Generale of higher Education of Indonesia.

References

1. C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, (McGraw-Hill, New York, 2008).
2. P. C. Rigby and C. Bird, Convergent Contemporary Software Peer Review Practices, in *Proc. 9th Joint Meeting European Softw. Eng. Conf. & ACM SIGSOFT Symp. Foundations Softw. Eng.*, (St. Petersburg, Russia, 2013), pp.202–212.
3. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, in *Proc. 35th Int'l Conf. Softw. Eng.*, (San Francisco, 2013), pp. 672–681.
4. A. E. Burhandenny, H. Aman, and M. Kawahara, Examination of coding violations focusing on their change patterns over releases, in *Proc. 23rd Asia-Pacific Softw. Eng. Conf.*, (Hamilton, New Zealand, 2016), pp. 121–128.
5. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, Mining email social networks, in *Proc. Int'l Workshop Mining Softw. Repositories*, (Shanghai, China, 2006), pp. 137–143.
6. H. Shen, J. Fang, and J. Zhao, EFindBugs: Effective error ranking for FindBugs, in *Proc. 4th Int'l Conf. Softw. Testing, Verification and Validation*, (Berlin, 2011), pp.299–308.
7. C. Boogerd and L. Moonen, Assessing the value of coding standards: An empirical study, in *Proc. 24th Int'l Conf. Softw. Maintenance*, (Beijing, China, 2008), pp. 277–286.
8. T. Lee, J. B. Lee, and H. P. In, A study of different coding styles affecting code readability, *Int'l J. Softw. Eng. & Its App.*, **7**(5) (2013) 413–422.
9. S. Kim and M. D. Ernst, Which warnings should I fix first? in *Proc. 6th Joint Meeting European Softw. Eng. Conf. & ACM SIGSOFT Symp. Foundations Softw. Eng.*, (Dubrovnik, Croatia, 2007), pp. 45–54.