Investigation of Coding Violations Focusing on Authorships of Source Files

Aji Ery Burhandenny
Graduate School of Science and Engineering, Ehime University
Matsuyama, Ehime, Japan 790–8577

Engineering Faculty, Mulawarman University
Samarinda, East Kalimantan, Indonesia 75119
Email: a.burhandenny@ft.unmul.ac.id

Hirohisa Aman
and
Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Ehime, Japan 790–8577

***Abstract*—While static code analysis tools would be helpful in reviewing source code, they have not been actively utilized in practice. One of main reasons why they are not used by practitioners has been said that such tools output many warnings (violations to predefined rules) but most of them are false positive. Thus, there have been studies evaluating violations in the past. This paper focuses on one of such studies, which evaluates violations using their change patterns over releases. Then, the paper examines 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 paper collects violations made by a popular static code analysis tool, PMD, from seven open source software projects. The set of collected data is 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 results of data analyses show the following findings: (1) the difference in the authoring type has significant impacts on the trends of violations and their evaluations; (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.**

I. Introduction

Code review has been widely known for its usefulness in detecting latent faults as early as possible [1]. While it is ideal to perform code reviews for all source files whenever they are upgraded, it would be difficult due to their performing costs. In order to support the code review activity, automated tools, i.e., static code analysis tools have been developed in the past. While those tools can quickly and automatically detect problematic code fragments or fault-prone ones and they look to be great helps for code reviews, those tools have not been actively used by practitioners in practice. According to the report by Johnson et al. [2], one of key reasons why practitioners do not use such tools is said that too many warnings (violations to predefined rules) are outputted by tools but most of them are false-positive and worthless for many programmers. That is to say, many warnings are ones that programmers do not consider to be fixed during their programmings.

In order to address the issue of many false-positive violations, there have been studies which evaluate or prioritize violations. Shen et al. [3] proposed to leverage feedbacks from tool's users for providing rankings of violations which are more suitable for the users, and improving the true-positive rate. However, their method is not always reasonable since it requires a lot of manual evaluations (feedbacks) by user's hand. As an automated way of collecting evaluations, Aji et al. [4] focused on change patterns of violations over releases, and proposed a metric for evaluating violations, "Index of Programmer's 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 warned in the source file depend on the programmer's preference. If two or more programmers are involved in the development and maintenance of a source file, the violations appearing in the file 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 investigation.

The remainder of this paper is organized as follows: Section II presents the research background and the definition of the metric we used, and sets up our research questions. Section III reports our data collection from open source software (OSS) projects and the results of data analyses along with our discussions. Section IV briefly describes the related work of this study. Finally, Section V gives the conclusions of this paper and the future plan of our study.

II. Evaluation of Violation and Authorship of Source File

In order to evaluate violations made by a static code analysis tool, Aji et al. [4] focused on change patterns of violations over releases: (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. Aji et al. proposed the following metric, "Index of Programmers' Attention (IPA)" using these notions: For a violation (warning) v,

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

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 $\mathrm{IPA}(v) = \infty$. Notice that the study excludes violation v such that $N_o(v) + N_d(v) + N_s(v) + N_i(v) = 0$. □

Based on the above notion, we can categorize violations into the following three groups—priority "high," "middle" and "low":

1) When $N_o(v) + N_d(v) > 0$ and $N_s(v) + N_i(v) = 0$ ($\mathrm{IPA}(v) = \infty$), there were only cases that programmers paid attentions. In these cases, violation v must be related to problematic or fully-undesirable code. Such a violation would be highly important in the code quality management. We define v's priority to be "high."
2) When $N_o(v) + N_d(v) \geq N_s(v) + N_i(v) > 0$ ($1 \leq \mathrm{IPA}(v) < \infty$), there were both cases that programmers paid attentions and disregarded, and the number of former cases is greater than or equal to the number of latter cases. In these cases, violation v would be relatively important: we define v's priority to be "middle."
3) When $N_o(v) + N_d(v) < N_s(v) + N_i(v)$ ($\mathrm{IPA}(v) < 1$), there were more disregarded cases, so violation v would be insignificant: we define v's priority to be "low."

□

The above priorities of violations can be automatically computed with using the code change history. It is an advantage of the proposal; while Shen et al. [3] proposed to adjust priorities for improving a static code analysis tool, their approach requires feedbacks from users and it would be hard to collect a lot of data in the case of large-scale software products.

Although the notion of IPA seems to be useful in evaluating many violations automatically, it may be affected by differences in organization or style of development. If a source file has been developed and maintained by a certain programmer only, the paying-attention or disregarding would depend 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. For the sake of convenience, 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."

Toward a more sophisticated IPA-based evaluation of violations, we will tackle the following research questions:

RQ1: Does the difference in the authoring type have an impact on the trend of violations and their evaluations?
RQ2: How many violations are commonly important or worthless across projects and authoring types?

III. Empirical Study

A. Aim and Studied Projects

In order to answer the above RQs, we examine OSS products from the perspective of not only the coding violation's priority but also the file authorship. Table I shows the projects surveyed in this study, which are the same as the dataset used in the previous work [4]; Those projects had been randomly selected from GitHub. By comparing IPA-based evaluations of coding violations between the set of single-authored files and the set of multi-authored ones, we study an impact of authorship on the priority of coding violations.

B. Data Collection

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

1) We cloned the repository on our local disk in order 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

TABLE I: Surveyed OSS projects.

Project Name	Investigation Period	# of Releases
Guava	Jan. 2010 − Dec. 2015	59
Elasticsearch	Feb. 2010 − Feb. 2016	143
Spring Framework	Dec. 2008 − Apr. 2016	85
React Native	Mar. 2015 − Feb. 2016	56
JabRef	Dec. 2011 − Jan. 2016	23
JUnit4	Dec. 2004 − Dec. 2014	20
Hibernate	July. 2010 − Feb. 2016	111

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]: (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.

4) For each single-authored file f_s and each violation v warned by PMD, 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 appeared 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 $\mathrm{IPA}(v)$ and decided its priority from "high," "middle" and "low." We determined priorities of all violations appeared in multi-authored files as well.

C. 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 priorities of violations. 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 priorities. Thus, we checked these two types of differences.

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 computed the Jaccard index as the similarity: 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 similarity between them, $Jac(V_s, V_m)$, is computed with the following equation:

$$Jac(V_s, V_m) = \frac{\mid V_s \cap V_m \mid}{\mid V_s \cup V_m \mid} \ . \tag{1}$$

The Jaccard index ranges 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 II shows the computed similarities. As the results, there is a variety in the similarity. Most violations appearing in Guava or Elasticsearch are common between the sets of single-authored files and of multi-authored ones, where their similarities are around 0.8. On the other hand, JUnit4 shows a low-level similarity (0.115), so there are significant differences between the sets of violations appearing in the single-authored files and of the multi-authored ones.

TABLE II: Similarities between violation sets: single-authored files vs. multi-authored files.

Project	Jaccard Index
Guava	0.813
Elasticsearch	0.791
Spring Framework	0.637
React Native	0.547
JabRef	0.398
JUnit4	0.115
Hibernate	0.659

Next, we focused on whether there is a difference in the trends of violation priorities in accordance with the file authorship. Table III summarizes the numbers of violations according to their priority levels and their authoring types. From the table, we can see the common trend that most appearing violations have low-level priorities, and only a few violations are at the middle or high levels.

In order to examine a difference between violations according to the authorships, we singled out those middle or high-level-priority violations. Table IV shows those violations and their priority levels according to authorships. While there are 29 violations[1] having middle or high-level priorities in either the single-authored files or the multi-authored ones, 15 out of 29 violations appear only in one of two authoring types. Moreover, 12 out of the remaining 14 violations have low-level priorities in either the single-authored files or the multi-authored ones. 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 authors (programmers). Such a difference may come from the preference of programmer.

TABLE III: Number of violations according to their priorities and authoring types.

Project	Priority	Single-Authored	Multi-Authored
Guava	Low	146	161
	Middle	0	1
	High	0	2
Elasticsearch	Low	140	169
	Middle	0	3
	High	3	2
Spring Framework	Low	119	177
	Middle	0	1
	High	0	1
React Native	Low	66	110
	Middle	0	1
	High	4	0
JabRef	Low	63	161
	Middle	0	2
	High	3	3
JUnit4	Low	14	111
	Middle	0	0
	High	0	1
Hibernate	Low	119	184
	Middle	0	0
	High	4	0

[1] Violations "AddEmptyString," "PositionLiteralsFirstInCaseInsensitiveComparisons" and "ImportFromSapePackage" appears across projects.

TABLE IV: Appearing violations with middle or high level priorities in single-authored files or multi-authored files.

Project	Violation	Priority	
		Single-Authored	Multi-Authored
Guava	AddEmptyString	Low	Middle
	EmptyStatementNotInLoop	—	High
	ForLoopsMustUseBraces	Low	High
Elasticsearch	JUnit4TestShouldUseTestAnnotation	High	Low
	PositionLiteralsFirstInCaseInsensitiveComparisons	High	Low
	CloneMethodMustBePublic	—	Middle
	DoNotCallSystemExit	—	Middle
	UseProperClassLoader	High	Middle
	BadComparison	—	High
	DontImportJavaLang	—	High
Spring Framework	AddEmptyString	—	Middle
	UnusedLocalVariable	—	High
React Native	UnnecessaryLocalBeforeReturn	—	Middle
	ConfusingTernary	High	Low
	IfStmtsMustUseBraces	High	—
	SimplifyBooleanReturns	High	Low
	UseLocaleWithCaseConversions	High	Low
JabRef	MisleadingVariableName	—	Middle
	UnusedImports	—	Middle
	AvoidUsingShortType	—	High
	DuplicateImports	—	High
	AppendCharacterWithChar	High	Low
	ImportFromSamePackage	High	High
	SingletonClassReturningNewInstance	High	Low
JUnit4	DoNotThrowExceptionInFinally	—	High
Hibernate	AvoidCatchingNPE	High	Low
	ImportFromSamePackage	High	—
	PositionLiteralsFirstInCaseInsensitiveComparisons	High	Low
	TooManyStaticImports	High	Low

In Table IV, only two violations—emphasized in boldface—have middle or high-level priorities 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 is a warning that there is no need to import classes within the same package. Since the former violation seems to be related to a potential fault and not to a programmer's preference, 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 a little or no relation with a fault. Hence, making the violation totally depends on who writes/maintains the code: While "ImportFromSamePackage" has also a high-level priority 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 IPA values which are original data to decide the priority levels of violations—low, middle and high. For each project, we computed the Spearman rank correlation coefficient between the sets of IPA values corresponding to violations warned in the single-authored files and in the multi-authored ones. Table V shows the results; Notice that those correlation coefficients are computed by using only IPA values of violations which are common to both the set of single-authored files and that of multi-authored ones. In Table V, no strong correlation between IPA values is observed in our data. That is to say, the priority—the degree of attention paid by programmer—of a violation appearing in single-authored files tends to be independent of the priority in multi-authored ones, even when we focus only on the common violations.

TABLE V: Spearman rank correlation coefficients between the set of single-authored files and that of multi-authored ones in terms of IPA 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

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.

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

In the previous subsection, we have analyzed violations by focusing on the difference in the authoring style. Now we introduce another perspective of analysis: the comparison across the projects.

For violations appearing in single-authored files, we computed similarities among projects as well. Table VI shows the

TABLE VI: Similarities among projects in terms of the set of violations appearing in single-authored files.

Project	(a)	(b)	(c)	(d)	(e)	(f)	(g)
(a) Guava	—	0.606	0.514	0.403	0.333	0.096	0.573
(b) Elasticsearch	0.606	—	0.638	0.449	0.375	0.098	0.652
(c) Spring Framework	0.514	0.638	—	0.443	0.445	0.118	0.646
(d) React Native	0.403	0.449	0.443	—	0.432	0.200	0.462
(e) JabRef	0.333	0.375	0.445	0.432	—	0.176	0.432
(f) JUnit4	0.096	0.098	0.118	0.200	0.176	—	0.114
(g) Hibernate	0.573	0.652	0.646	0.462	0.432	0.114	—

TABLE VII: Similarities among projects in terms of the set of violations appearing in multi-authored files.

Project	(a)	(b)	(c)	(d)	(e)	(f)	(g)
(a) Guava	—	0.673	0.657	0.599	0.650	0.533	0.665
(b) Elasticsearch	0.673	—	0.783	0.575	0.735	0.529	0.817
(c) Spring Framework	0.657	0.783	—	0.543	0.742	0.524	0.833
(d) React Native	0.599	0.575	0.543	—	0.547	0.517	0.545
(e) JabRef	0.650	0.735	0.742	0.547	—	0.536	0.777
(f) JUnit4	0.533	0.529	0.523	0.517	0.536	—	0.526
(g) Hibernate	0.665	0.817	0.833	0.545	0.777	0.526	—

results. The average of similarities (excluding the ones with themselves) is 0.342. Thus, about 30–40% of violations seem to be common with other projects. Since (f) JUnit4 has lower similarities with the others, it might have a special trend of violations, caused by a particular author.

In a similar fashion, we computed similarities among projects for violations appearing in multi-authored files. Table VII presents the results. The average of similarities is 0.634: about 60–70% of violations are common to other projects.

By comparing Tables VI and VII, the commonality of violations appearing in the multi-authored files is about 1.85 times higher (on average) than the single-authored ones. Indeed, for all pairs of projects, similarities in multi-authored files are higher than the ones in single-authored ones—"Table VI $<$ Table VII"; for example, pair (a)-(b): $0.606 < 0.673$, pair (b)-(c): $0.638 < 0.783$, pair (c)-(d): $0.443 < 0.543$, and so on. Trends of appearing violations are possibly generalized through maintenances by two or more programmers.

Next, we examined which violations are common across projects. Table VIII and Fig. 1 presents the numbers of common violations across projects; Table VIII also shows the numbers of violations corresponding to middle or high-level priorities as the numbers enclosed in parentheses: for example, 25 violations appearing in single-authored files are common to 6 projects, and 2 out of 25 violations have middle or high-level priorities[2]. In Table VIII, the maximum count of violations warned in single-authored files is at column "1," thus, most violations appeared in a particular project only. On the other hand, the maximum count of violations warned in multi-authored files is at opposite side "7," which mean that most violations are common to all projects. From these results, we can say that: more project-specific violations tend to appear in single-authored files, and more common violations do in multi-authored files.

TABLE VIII: Number of common violations across projects and of ones corresponding to middle or high-level priorities.

Author ship	# of Appearing Projects						
	1	2	3	4	5	6	7
Single	53(0)	24(3)	28(2)	33(1)	26(2)	25(2)	12(0)
Multi	33(2)	19(1)	24(2)	20(0)	32(4)	35(3)	71(4)

[2]It corresponds to "Single" row, "6" column.

In multi-authored files, 67 $(= 71-4)$ violations are common to all projects but they have low-level priorities (see Table VIII). That is to say, while those 67 violations must be popular, all of them have been disregarded by many programmers. Figure 2 presents the list of those 67 violations. They 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 violations having middle or high-level priorities, which are common to three or more projects[3]. 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 [3], [6].

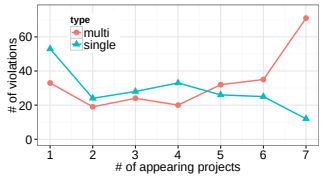

Fig. 1: Number of common violations across projects.

[3]Although Table VIII shows that 4 violations correspond to middle or high-level priorities, it just means that these violations are "not common low-level (disregarded)" ones. In other words, these 4 violations are not commonly-regarded ones for all projects while one of them has a middle or high-level priority at only one or two out of seven projects.

AbstractNaming, AccessorClassGeneration, AppendCharacterWithChar, ArrayIsStoredDirectly, AssignmentInOperand, AtLeastOneConstructor, AvoidCatchingGenericException, AvoidCatchingThrowable, AvoidDuplicateLiterals, AvoidFieldNameMatchingMethodName, AvoidInstantiatingObjectsInLoops, AvoidLiteralsInIfCondition, AvoidReassigningParameters, AvoidSynchronizedAtMethodLevel, AvoidThrowingRawExceptionTypes, AvoidUsingVolatile, BeanMembersShouldSerialize, BooleanGetMethodName, CallSuperInConstructor, ClassWithOnlyPrivateConstructorsShouldBeFinal, CommentDefaultAccessModifier, CommentRequired, CommentSize, CompareObjectsWithEquals, ConfusingTernary, 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, SimplifyBooleanReturns, StdCyclomaticComplexity, TooManyMethods, UncommentedEmptyConstructor, UncommentedEmptyMethodBody, UnnecessaryFullyQualifiedName, UnusedModifier, UseCollectionIsEmpty, UseConcurrentHashMap, UseUtilityClass, UseVarargs, UselessParentheses, VariableNamingConventions

Fig. 2: Commonly-disregarded violations.

E. Threats to Validity

We heavily rely our data collection on Github repositories. It may be a threat to validity in regard to a bias of data. Although we mitigated the threat by selecting projects randomly, we should perform a wider data collection and an analysis in order to show a high-level generality of our results.

We examined Java source code only due to our tool (PMD) limitation. There might be language-specific trends in our results. Since we have just categorized violations and analyzed their trends from a statistical perspective, we would like to examine qualitative aspects of them in the future.

While we examined code changes in repositories, we are not sure whether 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. Although our method is one of available ways to observe programmers' practices, we would need to validate our data and results in the future.

IV. Related Work

Spacco et al. [7] exploited a fuzzy algorithm to determine commonalities among violations. Lee et al. [8] analyzed how the readability of code is affected by coding violations. While their studies are useful in detecting important or disregarded violations, they missed change history of violations over time.

Kim et al. [9] focused on the lifetime of warning (violation) and used it for their prioritization. Their focus are similar to our study. However, they did not consider the trends of violation changes: change patterns. Even if a violation has a long lifetime, there is a significant difference between a decreasing trend and an increasing one in terms of its importance. Our work utilizes the change patterns of violations through real code modifications and used metric "IPA" as a connection between human factors (programmers' attentions) and violation trends. Then, we have discussed an influence of difference in the style of authoring source files.

V. Conclusion and Future Work

In this paper, we focused on the authorship of source files, and considered two authoring types: single-authored files and multi-authored ones. The trends of coding violations warned in single-authored files might be influenced by the programmer's preference, then there may be a significant difference in evaluating violations. Thus, we examined the impacts of the authoring type on evaluations of coding violations.

We collected violations warned by a popular analysis tool, PMD, from seven OSS projects. Then, we analyzed the trends of violations and their evaluations, by comparing the sets of data from single-authored files and from the multi-authored ones. From the results of analyses, we obtained the following findings: (1) the difference in the authoring type has significant impacts on the trends of violations and their evaluations; (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.

Hence, it is important to prepare an appropriate rule set for the domain and organization of the target project while considering the authorship of code, in order to utilize static code analysis tools. 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 more 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.

Acknowledgment

This work was supported by JSPS KAKENSHI Grant Number 16K00099 and DIKTI Scholarships, Directorate Generale of higher Education of Indonesia. The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] M. E. Fagan, "Advances in software inspections," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, pp. 744–751, July 1986.

[2] 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 Int'l Conf. Softw. Eng.*, May 2013, pp. 672–681.

[3] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *Proc. the 2011 Fourth IEEE Int'l Conf. Softw. Testing, Verification and Validation*, Mar. 2011, pp. 299–308.

[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.*, Dec. 2016, pp. 121–128.

[5] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proc. 2006 Int'l Workshop on Mining Softw. Repositories*, May 2006, pp. 137–143.

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

[7] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proc. 2006 Int'l Workshop on Mining Softw. Repositories*, May 2006, pp. 133–136.

[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.*, vol. 7, no. 5, pp. 413–422, 2013.

[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.*, 2007, pp. 45–54.