Examination of Coding Violations Focusing on Their Change Patterns over Releases

Aji Ery Burhandenny
Graduate School of Science and Engineering
Ehime University
Matsuyama, Ehime, Japan 790–8577
Email: aji@se.cite.ehime-u.ac.jp

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*—Code review is an essential activity to ensure the quality of code being developed, and there have been static code checkers for aiding an effective code review. However, such tools have not been actively utilized in the world of programmers due to a lot of coding violations (warning) produced by tools and their false-positiveness. In order to analyze the automatically pointed violations and the actual attentions which programmers paid to those violations, this paper proposes a novel metric—the Index of Programmers' Attention (IPA)—and conducts 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.

I. Introduction

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 [1]. 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 [2]–[4]. 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 paper, we examine the relationship between the coding violations and the programmers' attention from the perspective of the change patterns of violations over the releases.

The remainder of this paper is organized as follows. Section II presents the research background and sets up two research questions. Section III reports on an empirical work examining the actual coding violations detected by a popular code checker, PMD, and analyzing the relationships of them with the programmers' attention. Section IV briefly describes related work. Finally, Section V gives the conclusion of this paper and the future work.

II. 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.

However, 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 [1]. Thus, there have been studies for enhancing the usefulness of code checkers by prioritizing warnings. Kim et al. [3] 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 paper, 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?

III. Empirical Work

A. 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 functionalities for collecting source files and their changes. We randomly selected 7 OSS projects (see Table I) 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.

TABLE I: Surveyed OSS projects.

Project Name	Investigation Period	#Releases	#Source Files
Guava	Jan. 2010 − Dec. 2015	59	1,678
Elastic Search	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

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

C. 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 I. The PMD provides predefined priorities that consist of five levels: Level 1 (Change absolutely required), level 2 (Change

[1]http://pmd.github.io/

[2]http://checkstyle.sourceforge.net/

[3]http://findbugs.sourceforge.net/

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

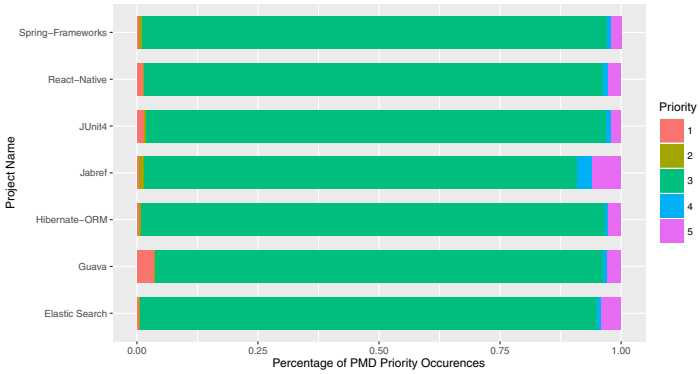


Fig. 1: Ratio of violations by their priority level predefined in the PMD (for the latest version of each OSS product).

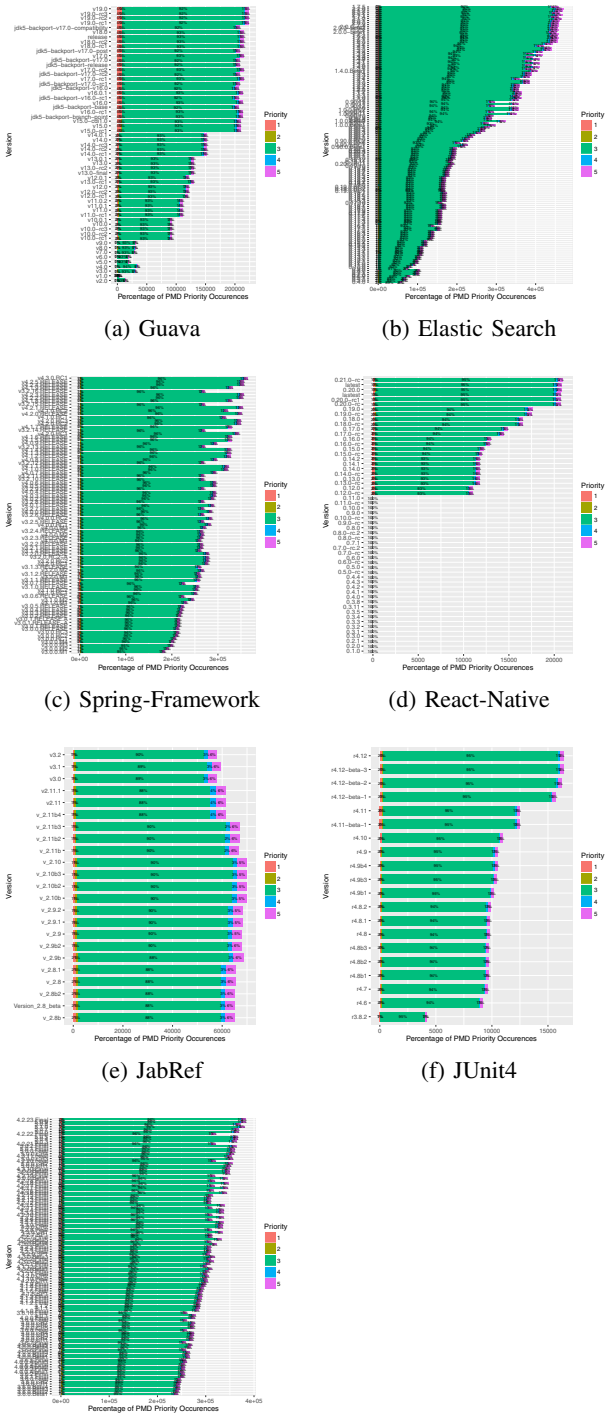
highly recommended), level 3 (Change recommended), level 4 (Change optional), and level 5 (Change highly optional). Figure 1 shows band charts signifying ratios of the above violation-priority levels at the latest version of all the surveyed OSS products.

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 Elastic Search: 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 2 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 Fig.2, 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.

D. Analysis 2: Trend of Each Violation

1) Trend Pattern: We examined all release versions of all source files included in the OSS projects shown in Table I, 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. A "one-shot" violation appeared only once during all releases; A "sticky"



(a) Guava (b) Elastic Search

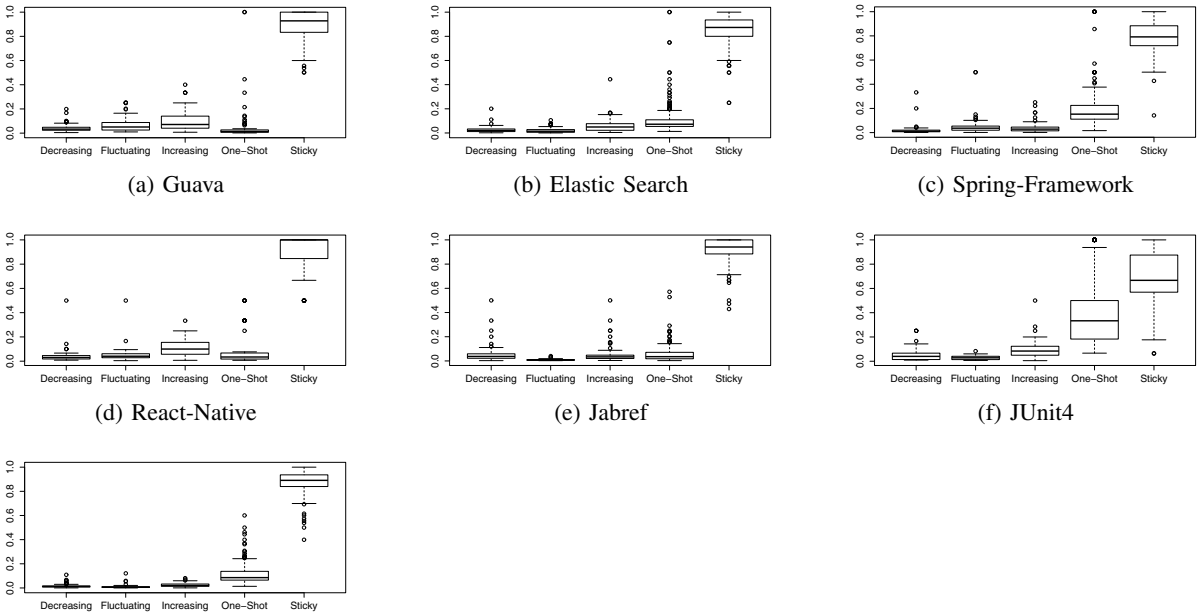
(c) Spring-Framework (d) React-Native

(e) JabRef (f) JUnit4

(g) Hibernate

Fig. 2: Trend of violations by the predefined priority level over all release versions.

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.



(a) Guava (b) Elastic Search (c) Spring-Framework

(d) React-Native (e) Jabref (f) JUnit4

(g) Hibernate

Fig. 3: Boxplot showing the rates of violation patterns in each OSS product.

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) Elastic Search, 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 paper, we summarize their trends in Fig. 3 and Table II. Figure 3 shows the boxplots of the percentages of the above five patterns, and Table II presents the averages and medians of those percentages.

From Fig.3 and Table II, 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. In Section III-D2, we will explore coding violations belong to these two patterns.

We will exclude the violations of "fluctuating" pattern from

TABLE II: 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
Elastic Search	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

our discussion hereinafter since that pattern has aspects of both "increasing" and "decreasing" and we cannot clearly categorize them in terms of programmers' attention.

2) 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 III. 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,

TABLE III: 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
...	...	...	...	...

"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 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 III, we get the following IPA values:

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

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

and

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

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 IV–X. In these tables, violations which appeared across projects are marked with an asterisk(*).

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 IV), it does not appear in the top 10 lists of projects other than the Guava, i.e., Tables V–X. Although programmers in the projects other than the Guava did not pay attention to whether a method's

TABLE IV: 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 V: Violations of top 10 IPA (Elastic Search).

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 VI: 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 VII: 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 VIII: 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 IX: Violations of top 10 IPA (JUnit4).

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

TABLE X: 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

TABLE XI: Number of violations whose IPA value is zero.

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

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. [5] 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.

3) 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 XI shows the number of violations whose IPA value is zero. As shown in Table XI, 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.

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).

[5]The rankings were No.126 in Elastic Search, 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 XII: 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

Table XII 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 XII) 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 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 IV–X)—they are marked with a dagger (†) in Table XII. For example, "InefficientStringBuffering" (see No.21 in Table XII) appeared in the IPA top 10 ranking of Elastic Search (see Table V) 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 Elastic Search, who considered those parts should be improved.

TABLE XIII: 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	∞

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

4) IPA vs. Number of Violations: Finally, we analyze a relationship between the IPA value and the number of violations. Table XIII 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 XIII, 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 4 presents the number of violations by the IPA category. From Fig. 4, 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).

E. 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

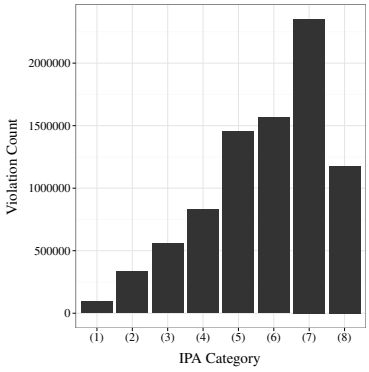


Fig. 4: Number of violations by IPA category.

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.

IV. 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 [6]. Among those studies, Spacco et al. [2], Kim et al. [3] and Lee et al. [4] 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. [7] 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. [8] 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.

V. Conclusion and Future Work

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%). Therefore, 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). Our future work is to enhance the accuracy of pointing the part to be improved, by using not only the PMD but also other related factors including the process and product metrics. For example, the data of who developed the code would be useful in predicting fault-prone parts by using it along with coding violations. Moreover, to evaluate our results using some benchmarks such as FaultBench v0.3 [9] proposed by Heckman and Williams [10] from the perspective of whether a coding violation is actionable or not.

Acknowledgment

This work was supported by JSPS KAKENSHI Grant Number 16K00099 and DIKTI Scholarships, Directorate General of Higher Education of Indonesia.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proceedings of the 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

[8] 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 *Proceedings of the 37th International Conference on Software Engineering*, May 2015, pp. 437–447.

[9] S. Heckman and L. Williams, "Faultbench," 2014. [Online]. Available: http://www.realsearchgroup.org/faultbench/

[10] ——, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct. 2008, pp. 41–50.