Empirical Analysis of Coding Standard Violation Focusing on Its Coverage and Importance

Aji Ery Burhandenny[*][†], Hirohisa Aman[‡] and Minoru Kawahara[‡]
[*]Graduate School of Science and Engineering, Ehime University, Matsuyama, Ehime 790-8577, Japan
[†]Engineering Faculty, Mulawarman University, Samarinda 75119, East Kalimantan, Indonesia
[‡]Center for Information Technology, Ehime University, Matsuyama, Ehime 790-8577, Japan

***Abstract*—Toward an effective utilization of static code analysis tools, this paper investigates which violation is familiar with more programmers and widely appears in source files (having a high coverage), and which violation is really related to bugfixes (having a high importance), for six popular OSS projects. The results show: 1) the familiar violations tend to differ among projects, and only 25 violations are common to all surveyed projects; 2) the trends of their importance vary from project to project.**

I. Introduction

An efficient performance of code review has become a great challenge for developers. To support code review, static code analysis tools have been developed. However, in many cases, these tools have not been widely used by developers: a static analysis tool tends to produce a huge number of warnings but most of them are false positive ones, i.e., they are not serious points to be revised [1], [2]. It is one of key reasons why developers do not actively use such a tool. Therefore, toward a better utilization of such tools, there have been studies in the past (e.g, [3], [4]). While previous studies provide notable findings and proposals, they have not well argued the real trends of violations (warnings) made by a static analysis tool. In this paper, we examine the coding violations from the perspectives of how widely they appear and how important they are in terms of bugfixes.

II. Coding Standard Violation

A. Change of Coding Standard Violations through Commits

By using both the code repository and a static code analysis tool, we can get the change history of coding standard violations in source files. Concretely, for each source file, we obtain all versions of the file from the repository and check them by a static code analysis tool. Then, for each violation which appeared in a version of the file, we can see who made it, who cleared it and when they made/cleared it.

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

B. Research Questions and Metrics

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

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

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

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

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

The following metric FC(v_i) is the file coverage of v_i, which expresses how widely v_i appears in source files: FC$(v_i) = n_s(v_i)/s$, where $n_s(v_i)$ is the number of source files which have experiences with being warned as violation v_i.

The following metric DC(v_i) is the developer coverage of v_i, which signifies how widely v_i is linked to developers: DC$(v_i) = n_p(v_i)/p$, where $n_p(v_i)$ is the number of developers who have made or increased v_i.

Both FC(v_i) and DC(v_i) range $[0, 1]$.

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

2) Violation Importance: As mentioned above, if the warning count of violation v_i decreased through a bugfix, v_i seems to be related to the fixed bug, and it would be an important violation for preventing a bug inducing. On the other hand, if the warning count of v_i increased through a bugfix, v_i seems to be less important. By focusing on the difference between the decreased count and the increased one, we define the following metric IMP(v_i) which presents an importance of v_i: $\mathrm{IMP}(v_i) = \frac{n_{dec}(v_i)}{\sum_{j=1}^{w} n_{dec}(v_j)} - \frac{n_{inc}(v_i)}{\sum_{j=1}^{w} n_{inc}(v_j)}$, where $n_{dec}(v_i)$ and $n_{inc}(v_i)$ signify the numbers of bugfix commits in which the warning count of v_i decreased and increased, respectively.

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

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

III. Empirical Analysis

To answer the above RQs, we collect a lot of actual data from OSS development projects and analyze the coding violations which had appeared in the source files. In order to ensure the generality and usefulness of our empirical results, we randomly selected six popular OSS projects from the GitHub: (1) Elasticsearch, (2) Guava, (3) JabRef, (4) JUnit4, (5) React Native and (6) Spring Framework.

We computed FC, DC and IMP values of violations and focused on violations having relatively high coverage, whose FC and CD values are greater than their medians. As a result, they differ among projects; Table I shows the number of violations categorized by "IMP > 0" and "IMP ≤ 0."

As a consequence, only 25 violations are common to all six projects (see Table II). There is only one violation whose IMP value is always positive among projects, "UnusedImports." On the other hand, there are only two violations whose IMP values are always non-positive, "CallSuperInConstructor" and "CommentRequired." That is to say, even if we focus only on commonly familiar (higher coverage) violations, their trends of importance differ from project to project.

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

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

TABLE I
Number of violations having high coverage in each project.

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

TABLE II
Common violations having higher FC values and DC values.

# of projects where IMP > 0	violations
6	UnusedImports
5	DataflowAnomalyAnalysis
4	LawOfDemeter, AccessorMethodGeneration
3	LocalVariableCouldBeFinal, AvoidInstantiatingObjectsInLoops, AvoidCatchingGenericException, AtLeastOneConstructor, AvoidLiteralsInIfCondition, BeanMembersShouldSerialize, UseConcurrentHashMap, OnlyOneReturn, CommentSize
2	ConfusingTernary, EmptyCatchBlock, GodClass, UselessParentheses, DefaultPackage
1	TooManyMethods, CommentDefaultAccessModifier, ShortVariable, LongVariable, MethodArgumentCouldBeFinal
0	CallSuperInConstructor, CommentRequired

IV. Conclusion

We focused on coding standard violations warned by a static code analysis tool, and proposed to evaluate them from the perspective of the familiarity—a file coverage and a developer coverage—and the importance—a strength of relationships with bugfixes. We conducted an empirical analysis of coding violations appearing in six popular OSS projects. As a result, familiar violations seemed to differ among projects, and only 25 violations were common to all surveyed projects. Moreover, the trends of their importance varied from project to project. Therefore, we found that it is better to tune the assessments of violations for each project, rather than to build a general guideline of coding violations.

Acknowledgment

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

References

[1] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proc. Int'l Conf. Softw. Testing Verification & Validation*, Apr. 2009, pp. 141–150.

[2] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *Proc. Int'l Conf. Softw. Analysis, Evolution, & Reeng.* , Mar. 2015, pp. 161–170.

[3] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proc. 11th Working Conf. Mining Softw. Repositories*, May 2014, pp. 152–161.

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