Impact of Coding Style Checker on Code Review -A case study on the OpenStack projects-

Yuki Ueda*, Akinori Ihara†, Takashi Ishio* and Kenichi Matsumoto*
*Nara Institute of Science and Technology, Japan
†Wakayama University, Japan
Email: ueda.yuki.un7@is.naist.jp, ihara@sys.wakayama-u.ac.jp, {ishio, matumoto}@is.naist.jp

***Abstract*—Code review is key to ensuring the absence of potential issues in source code. Code review is changing from a costly manual check by reviewer to a cost-efficient automatic check by coding style checkers. So that patch authors can verify the changed code before submitting their patches. Although cost-efficiency, the checkers do not detect all potential issues, requiring reviewers to verify the submitted patches based on their knowledge. It would be most efficient if patch authors will learn potential issues and remove the same type of issues from patches prior to code review. This study investigates potential issues that patch authors have repeatedly introduced in their patch submissions despite receiving feedback. To understand the impact of adopting checkers to patch authors' coding style improvement, this study compares two types of potential issues: Automatically Detected Issues by checkers (ADIs) and Manually Detected Issues by reviewers (MDIs). In a case study using an OpenStack code review dataset, we found that the patch authors have repeatedly introduced the same type of MDIs, while they do not repeat ADIs. This result suggests that the introduction of code style checkers might promote the patch authors' effective potential issues learning.**

I. Introduction

Code review requires highly collaborative works between patch author and reviewers [1]. Its process involves source code verification, feedback, and modification. Reviewers provide a technical oversight for patch authors, eradicating coding issues that patch authors may not be able to self-detect. This collaborative process is key to ensuring that potential issues, i.e., potential issues are fixed.

Code review is changing from a costly manual check by reviewer to a cost-efficient automatic check by coding style checkers like Pylint[1], so that patch authors can verify the changed code before submitting their patches. The problem is that coding style checkers cannot detect all potential issues. Potential issues are almost always detected through manual code review.

We conduct an empirical study to explore if patch authors repeatedly introduce the same type of potential issues despite receiving reviewers' feedback. To this end, this study focuses on two types of potential issues: Automatically Detected Issues by coding style checkers (**ADI**s) and Manually Detected Issues by reviewers (**MDI**s).

As a case study, we analyze 228,099 submitted patches in OpenStack. In the dataset, some patch authors introduced the same type of issues more than twice. We, therefore, study how coding style checker and reviewers' feedback improve patch authors coding style or not:

RQ1: How often do the patch authors repeatedly introduce ADIs in future patch submissions? We found that the patch authors that have introduced the ADIs rarely introduce the same type of ADIs in their future patch submissions.

RQ2: How often do the patch authors repeatedly introduce MDIs in future patch submissions? We found that the patch authors repeatedly introduce the same type of MDIs more frequently than ADIs in their subsequent patch submissions. We also found that the patch authors find it difficult to detect MDIs in the submitted patches that are executable without runtime error before submitting the changed code. Hence, manual code review by the reviewers is necessary.

Our main contributions are two-fold. The first contribution is an investigation on the impact of coding style checker, which improves patch authors' coding style to avoid the same type of issues in subsequent patch submissions. The second contribution is evidence that shows difficulty patch authors face MDI with avoiding before submitting the changed code. Our study concludes that manual code review by reviewers is necessary to avoid MDIs.

The rest of the paper is organized as follows. Section II introduces the code review process and related works. Section III describes our approach in answering the two research questions. Section IV presents the results of research questions. Section V establishes the validity of our empirical study. Section VI concludes this paper and describes our future work.

II. Background and Related Work

A. Code Review Process

There are various tools for managing peer code review processes. For example, Gerrit[2] and ReviewBoard[3] are commonly used in many software projects to receive lightweight reviews. Technically, these code review tools are used for patch submission triggers, automatic tests and manual reviewing to decide whether or not the submitted patch should be integrated into a version control system.

Following the steps illustrated below is the code review process in the Gerrit Code Review. Gerrit is used by our target OpenStack projects as a code review management tool.

[1]Pylint: https://www.pylint.org/

[2]Gerrit Code Review: https://code.google.com/p/gerrit/

[3]ReviewBoard: https://www.reviewboard.org/

1. A patch author submits a patch to Gerrit Code Review. We define the submitted patch as $Patch_1$.
2. Reviewers verify $Patch_1$. If the reviewers detect any issues, they send feedback and ask for a revision of the patch through Gerrit Code Review.
3. After the patch author revises $Patch_1$ and submits the fixed patch as $Patch_2$, the reviewer verifies $Patch_2$ again. The patch authors need to repeatedly fix the patch until the reviewers make a decision to accept or reject the patch. We define the last patch as $Patch_n$.
4. Once the patch author completely addresses the concerns of the reviewers, $Patch_n$ will be integrated into the version control system.

Code review process requires a shared coding style among its participants. When many patch authors with different technical skills contribute to software development, the project manager needs to keep a consistent coding style to keep higher maintainability. To achieve a consistent coding style, software projects often have their own coding rules. Coding rules represent a guideline for patch authors to implement software in compliance with a common coding style among patch authors.

To support implementation with common coding styles, some coding style checkers for automatically detecting potential issues are needed. Using such tools, the patch authors can automatically check whether or not the changed source code is based on the coding rule before submitting their patches. In addition, the output of the automatic coding style checkers is helpful for reviewers to study potential issues to develop coding skill [2].

B. Related Works

Code Review Many researchers have conducted empirical studies to understand code review [3], [4]. Unlike our focus, most published code review studies focus on the review process or the reviewers' communication. While code review is effective in improving the quality of software artifacts, it requires a large amount of time and many human resources [5]. Various methods are proposed to select appropriate reviewers based on the reviewer's experience [6]–[8] and complexity of code changes [9]. In our work here, we focus on code changes based on feedback from the reviewers.

Code Quality coding style checkers verify the source code automatically. In addition to Pylint, python has the tools that detect potential issues, such as pep8[4] and flake8[5]. Using these tools makes detecting python's potential issues easier. Our study aims to understand the impact of code fixing for issues detected by these coding style checkers.

Code reviews are refactoring based on coding rules [10]. Also, patch authors and reviewers often discuss and propose solutions with each other to revise patches [11]. In particular, 75% of discussions for revising a patch are about software maintenance and 15% are about functional issues [4]. These studies help us understand which issues should be solved in the code review process. We focus on the impact of code fixing suggestions from reviewers. In particular, we study whether patch authors that have introduced and fixed potential issues repeatedly introduce the same type of issues in future patch submissions.

[4]pep8: https://pypi.python.org/pypi/pep8

[5]flake8: https://pypi.python.org/pypi/flake8

III. Study Design

A. Overview

This study investigates whether patch authors who have introduced the potential issues repeatedly introduce the same type of issues in future patch submissions. Specifically, we analyze the relationship between the experience of patch authors and the potential issues included in patches.

Figure 1 describes an overview of the procedure to measure two metrics for each research question. The two metrics are:

1) *Patch Submission Experience*, which represents past submitted patches that patch authors changed and submitted to the code review tool. This study analyzes each patch p include the author's name, the time of submission, and the source file changes.
2) *potential issues* are issues which are detected by coding style checkers or by reviewers. This study identifies potential issues in the submitted version ($Patch_1$) and fixed version ($Patch_n$) and summarizes the occurrences as a metric.

We measure those metrics from a set of the submitted patches (P_s) to a code review tool.

B. Extraction of potential issues

This study identifies potential issues from a patch p if:

- Patch p is integrated into the project repository, and
- the submitted version ($Patch_1$) of p is different from the integrated version ($Patch_n$).

If a patch p is directly integrated into the project repository without any changes, the patch has no potential issue. We ignore rejected patches from our analysis because reviewers may reject those patches without fixing them completely.

We investigate two groups of potential issues:

ADI: **Automatically Detected Issue by a current tool**
MDI: **Manually Detected Issue by a reviewer**

We distinguish these two groups because patch authors can identify ADIs by themselves using coding style checkers. In other words, while ADIs are directly visible to patch authors, MDIs are not detected before reviewers verify them.

1) ADI: We identified ADIs by executing coding style checker to source code files. The coding style checker outputs in which lines the target source code files include ADIs. We define the submitted patch that includes ADIs if there are ADIs in any lines of any files. Our target OpenStack project using Python employs Pylint to detect ADIs as listed in Table I. The Pylint can detect 105 ADIs. This table shows the most frequent 7 ADIs that patch authors introduced. We specifically regard three category issues with the detection messages "Convention", "Refactor", and "Warning" outputted

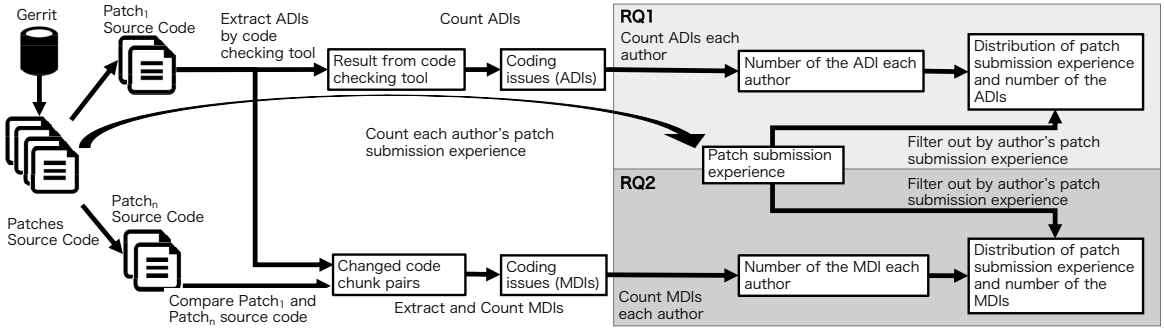

Fig. 1. Overview of our approach to answer the research questions.

TABLE I
OUR TARGET POTENTIAL ISSUES AND THE EXAMPLES.

Group	Category	Issue Type	Issue description	Submitted example	Fixed example
ADI	Convention	invalid-name	Any name not matching the PEP8 names	`lowerCase = 0`	`lower_case = 0`
		bad-continuation	Wrongly continued indentation	`if(foo > 0 and foo < 1):`	`if(foo > 0 and foo < 1):`
		wrong-import-order	Imported modules are not sorted	`import my_package import sys`	`import sys import my_package`
	Refactor	no-self-use	A method is not preceded by a decorator and does not contain any references	`def func(foo, bar): return foo * bar`	`@staticmethod def func(foo, bar): return foo * bar`
		too-few-public-method	A Class just stores data without methods	*Omitted due to the limited space**	*Omitted due to the limited space*
		too-many-arguments	A method or function takes too many arguments	*Omitted due to the limited space***	*Omitted due to the limited space*
	Warning	unused-argument	An argument is not used	`def func(foo, bar): return foo`	`def func(foo): return foo`
MDI	(None)	new_line	Line separators are deficient or excessive (including indent)	`foo = [0, 1]`	`foo = [0, 1]`
		alphabet	Any name could be correct	`foo = 0`	`bar = 0`
		space	Space or tab could be format	`foo = 0`	`foo=0`
		number	Number literal could be correct	`foo = 0`	`foo = 1`
		strings	String literal could be correct	`print("Hello")`	`print("HelloWorld")`
		comment	Comment line need to be change	`# TODO`	`# DONE`

* An example of too-few-public-method is available at: https://github.com/c-w/gutenberg/commit/2b1ff63fa1f17554bec6c9225872406d63300b72
** An example of too-many-arguments is available at: https://github.com/Bitmessage/PyBitmessage/commit/95e300d7cae5036d188ce9acc5bad6c91287dca1

from Pylint as the potential issue because they represent potential issues and potential bugs. We excluded "Error" and "Fatal" categories because they indicate functional problems.

2) MDI: We identified MDIs by comparing the submitted and fixed versions of a patch. However, it should be noted that we found it difficult to detect semantic differences and potential issues fixed between the versions because the patch authors or reviewers do not always explain why code fixing is needed. Hence, this study assumes that a potential issue is fixed by only one chunk change. As the future work, we will focus on changes that have multiple types of changes such as "number" and "strings". We approximate potential issues by syntactic differences using a heuristic approach. We regard 6 types of syntactic elements as MDIs listed in Table I. Some MDIs are not independent of ADIs (e.g. "bad-continuation" and "space"). These MDIs were found frequently through our previous study [12] by manual surveys in the code changes between $Patch_1$ and $Patch_n$. The process of MDI detection comprises four steps as follows. Figure 2 illustrates the process for an example patch.

1. We extract pairs of code chunks (consecutive lines of code) from the submitted version and fixed version of a patch. The pairs are simply recognized by applying `diff` utility to the two versions.
2. For each chunk pair, we extract the differences between characters. If the chunk pair is exactly the same, the chunk has no potential issues.
3. We classify the differences of a chunk pair into six types of syntactic elements. We employed the lexical analyzer of ANTLR with a Python grammar file in implementation.
4. If the differences between a chunk pair include only one syntactic element type, we consider the chunk as an instance of a potential issue related to the type. If the differences include two or more types, we regard the differences as a problem other than potential issues. Finally, we extract all MDIs included in the submitted version of the patch.

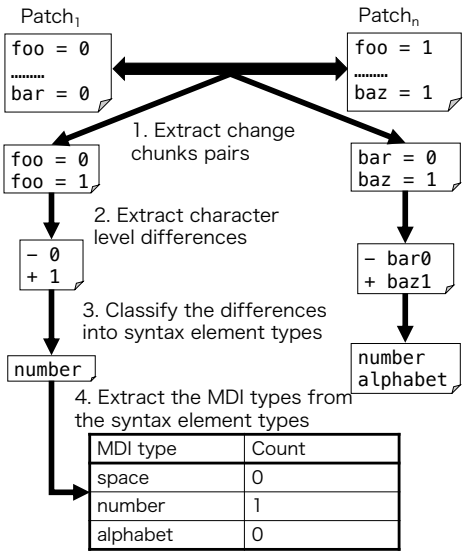

Fig. 2. Overview of our approach to count MDIs.

The detection accuracy of the above steps has been validated by the manual surveys using samples of code changes based on confidence level (95%). The individual detection accuracy for each MDI is: 100.0% for "comment", 97.5% for "strings", 96.8% for "new_line" and "space", 86.4% for "alphabet", and 57.1% for "number". The detection of five MDIs except for "number" is sufficiently accurate for statistical analysis. We also analyze "number" because it appears frequently.

C. Potential Issue Count

This study measures the occurrence of an individual issue type in patches. For an issue i, we represent its occurrence in a patch as a Boolean value $I_i(p) \in \{0, 1\}$. We define the number $A_i(a, n)$ for author a's n-th patch as follows.

$$A_i(a, n) \quad = \quad \sum_{p \in P(a,n)} I_i(p) \qquad (1)$$

If an author a repeatedly introduces an issue type i, $A_i(a, n)$ increases with the number of patch submissions n. Using this approach, we analyze whether patch authors that have introduced the potential issues repeatedly introduce the same type of issues in future patch submissions.

IV. Case Study

A. Dataset

This study targets a large and successful OSS project: the OpenStack systems, which have recorded a large amount of reviewing activity using the code review tool. OpenStack is a collection of open source components that make up the enterprise-scale cloud computing platform[6]. The OpenStack project has the coding rules based on PEP8 and employs a Pylint to automatically detect violations of coding rules.

[6]OpenStack: http://www.openstack.org/

TABLE II
Number of the introduced ADIs from 228,099 patches (sort by ADIs and MDIs in $Patch_1$)

Group	Issue Type	ADI in $Patch_1$	ADIs in $Patch_n$	Authors
ADI	invalid-name	2,517	1,406	1,116
	bad-continuation	1,544	797	809
	unused-argument	1,472	779	784
	no-self-use	1,432	745	779
	too-few-public-methods	1,372	701	736
	wrong-import-order	1,330	652	715
	too-many-arguments	1,209	613	662
		MDIs in $Patch_1$		
MDI	alphabet	22,950		3,130
	strings	21,094		3,066
	space	19,565		3,008
	number	16,512		2,760
	new_line	8,328		2,163
	comment	5,381		1,679

We target patches which the OpenStack project received from 2011 to 2015. During this period, 6,575 patch authors submitted 228,099 patches to the project. We obtain $Patch_1$ and $Patch_n$ using Gerrit REST API[7].

We obtained the patch submission experience, ADIs, and MDIs from the patch set. Table II shows the number of patches including each of the ADIs and MDIs, and the number of patch authors who submitted the patches. Top 7 potential issues frequently reported by the Pylint from the paper due to the space limitations. From Table II, we can recognize many ADIs in $Patch_1$ and the MDIs in $Patch_1$ which are fixed before $Patch_n$ for each patch author. For example, 1,116 patch authors have introduced 2,517 "invalid-name" issues as ADIs in total. It appears that each patch author has introduced the same "invalid-name" issue type 2.26 times on average. Also, 44.1% "invalid-name" issues are removed between $Patch_1$ which are fixed before $Patch_n$.

B. Analysis

This study investigates how often patch authors repeatedly introduce two groups of potential issues: ADIs (RQ1) and MDIs (RQ2). For each group of potential issues, we perform two analyses. First, this study analyzes how often patch authors have introduced each type of potential issue in our target dataset. We study whether ADIs or MDIs are more likely to be repeatedly introduced. Second, this study analyzes whether patch authors that have introduced the potential issues are likely to introduce the same type of issues repeatedly in future patch submissions. Since some patch authors have not submitted a sufficient number of patches to analyze their experience, we select patch authors who have submitted patches greater than or equal to 20 times. This threshold includes the most active 1,599 patch authors who have created 80% of patches in total according to the Pareto principle [13].

RQ1: How often do the patch authors repeatedly introduce ADIs in future patch submissions? Table III shows the distributions of 3 ADI issue counts for each author and

[7]https://gerrit-review.googlesource.com/Documentation/rest-api.html

TABLE III
THE RATIO (%) OF PATCH AUTHORS WHO INTRODUCED EACH OF ADIS AND MDIS n TIMES

n	invalid-name	bad-continuation	unused-argument	alphabet	strings	space
1	36.6	44.9	45.6	8.7	9.5	10.3
2	21.0	22.2	21.7	9.3	10.1	10.4
3	11.9	11.4	10.7	7.3	7.7	8.1
4	7.2	5.7	5.6	5.9	6.3	6.5
5	5.0	4.4	4.2	5.1	5.4	5.5
6	3.6	3.2	2.9	4.5	4.7	4.8
7	2.7	2.2	2.4	3.9	4.1	4.1
8	2.2	1.6	1.7	3.6	3.7	3.6
9	1.6	1.1	1.2	3.2	3.3	3.3
$\geq$10	8.1	3.3	4.0	48.5	45.3	43.4

TABLE IV
DISTRIBUTION OF EACH ADI INTRODUCED WHEN THE SUBMISSION EXPERIENCE IS 20

ADI Type	1st Qu.	Median	3rd Qu.
invalid-name	1	1	2
bad-continuation	1	1	2
unused-argument	1	1	2
no-self-use	1	1	2
too-few-public-methods	1	1	2
wrong-import-order	1	2	2
too-many-arguments	1	1	2

issue type. The plots include 1,117 patch authors who have introduced any ADIs at least once. Only 36%–45% of patch authors do not have introduced the same type of issue multiple times, even though the project has coding rules. However, 69%–78% of patch authors only introduced the same type of issues up to 3 times.

Also, Table IV shows the number of ADIs introduced by the most active patch authors. The table shows the median, the first quartile, and the third quartile of the ADIs in the first 20 patch submissions. The numbers are rounded to the nearest decimals to indicate the number of introductions. While some patch authors have more experience than 20 patches, we omitted the distributions of the later patches because they are very similar to the first 20 patches.

While the table includes the most frequent 7 types of ADIs, the median numbers of each issue are only introduced once except for "bad-continuation" and "wrong-import-order" issues. The result indicates that most patch authors only introduced the same type of issues twice. If the code review tools show experience of patch authors who introduced ADIs more than 3 times or not, this information can help to consider the reviewers should check the issues for experienced patch authors. Those issues are easy to understand, and the coding style checker is available for both patch authors and reviewers. When patch authors have received feedback once, they will understand how to check and avoid those potential issues in future patch submissions.

Answer of RQ1: While new patch authors may introduce ADIs, they will not repeatedly introduce the same type of ADIs more than three times.

RQ2: How often do the patch authors repeatedly introduce MDIs in future patch submissions? Table III shows the distributions of 3 MDI issue counts of patch authors. 3,488 patch authors who have introduced any MDI at least once. Only 8%–10% of patch authors do not have introduced the same type of issues multiple times. The numbers are greater than ADIs. We analyze the distributions of the numbers of MDIs introduced by the most active patch authors. Table V shows the median, the first quartile, and the third quartile of MDI issue counts in the first 20 patch submissions. While some patch authors have more experience than 20 patches, we omitted the distributions of the later patches because they are very similar to the first 20 patches. The median numbers of each issue for the patch authors are 3–6 times. The result means that reviewers have to verify those issues in patches irrespective of the experience of patch authors.

TABLE V
DISTRIBUTION OF EACH MDI INTRODUCED WHEN THE SUBMISSION EXPERIENCE IS 20

MDI Type	1st Qu.	Median	3rd Qu.
alphabet	4	6	8
strings	4	5	8
space	4	5	8
number	4	5	7
new_line	2	3	5
comment	2	3	4

Listings 1 and Listings 2 shows example patche whose "alphabet" issues are fixed through code review. The patches are written by the same author; Listings 1 shows the first instance of the "alphabet" issue introduced by the author. The submitted version imports "_" function to handle a log message, while he OpenStack project has a rule to use `_LW` ("Log warning") to record a message[8]. Since this rule is a project-specific rule, the correct implementation is difficult for a novice patch author. In Listing 2, the patch author uses the wrong object as a method receiver; so that the submitted patch read data from an object and write it to the same object stored in `common_policy` variable. That source code is also executable without a runtime error.

Answer of RQ2: Patch authors repeatedly introduce the same type of MDIs. Reviewers have to review patches carefully to identify MDIs.

V. THREATS TO VALIDITY

External validity: We analyzed 611 subprojects in the OpenStack project but only Python source code files. Surely, when we target other projects which use other programming languages, we may find differences. The OpenStack project mainly uses common coding rules that use the coding style checker (Pylint). We can then adapt our approach to analyze other software with other programming languages. We categorized MDIs by their change contents rather than their purposes in order to automate our analysis. Hence, a single category of MDI may include modifications for different purposes like Listings 1 and Listings 2. It has a risk to overestimate the

[8]https://docs.openstack.org/oslo.i18n/latest/user/usage.html

Listing 1
EXAMPLE OF THE FIXED "ALPHABET" ISSUE FOR AN IMPORT CORRECT MODULE: THE FIRST TIME FOR AN AUTHOR

```
Patch 1
1 from nova.i18n import _
-----------------------------------------------------------------
Patch n
1 from nova.i18n import _LW
```

Listing 2
EXAMPLE OF THE FIXED "ALPHABET" ISSUE FOR THE CORRECT VALUE: THE 17 TH TIME FOR THE AUTHOR OF LISTING 1

```
Patch 1
1 common_policy.set_rules(dict((k, common_policy.parse_rule
    (v))
2                              for k, v in rules.items()))
-----------------------------------------------------------------
Patch n
1 ironic_policy.set_rules(dict((k, common_policy.parse_rule
    (v))
2                              for k, v in rules.items()))
```

number of repeatedly introduced MDIs. In this research, we compared developers' experiences across multiple sub-projects with the number of potential issues, to measure the coding style improvement of the entire developer. Because MDIs are general category than ADIs. It can be future work to verify the improvement of coding styles of each developer. The number of patch submissions represents the experience of discussions with reviewers. In other words, patch authors had opportunities to learn potential issues in source code. In our future work, we plan to develop a method to detect MDIs automatically based on past code review feedback and patch's size/complexity.

Internal validity: First, we target potential issues when the patch authors change a couple of lines. Hence, we do not collect "God Class" or "Spaghetti Code" in the potential issues because large-scale changes will not become potential issues. Secondly, our analysis of ADIs may be dependent on pylint. Python has other coding style checkers like pep8. We choose pylint because it can detect more issues such as "unused-argument" that are not detected by pep8. Finally, our analysis compares only $Patch_2$ and $Patch_{n-1}$; it may misrecognize potential issues fixed through multiple revisions as another type of change. However, each revision also may include redundant changes that are not reflected in the final patch. Our analysis tries to investigate the effect of code review ignoring such changes.

VI. CONCLUSION

This paper conducted an empirical study to understand whether or not patch authors repeatedly introduce potential issues during code review process. As a case study using an OpenStack code review dataset, we specifically focused on two types of potential issues: automatically detected issues by coding (ADI) and manually detected issues by coding rules (MDI). We found those patch authors introduce the same type of ADIs less than three times. However, patch authors are likely to re-introduce the same type of MDIs. These results suggest that coding style checker are effective for improving patch authors' coding style. However, code reviewers should carefully verify issues that can not be detected automatically, no matter how many reviews they undergo. To save the cost associated with manual reviews, software projects need coding style checkers, and the scope of coding style checkers needs to be enhanced so that MDIs could also be automatically detected.

ACKNOWLEDGMENT

We would like to thank the Support Center for Advanced Telecommunications (SCAT) Technology Research, Foundation. This work was supported by JSPS KAKENHI Grant Numbers 18H03222, 17H00731, 15H02683 and 18KT0013.

REFERENCES

[1] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proc. ICSE*, 2015, pp. 403–414.

[2] R. Lobb and J. Harlow, "Coderunner: A tool for assessing computer programming skills," *ACM Inroads*, vol. 7, no. 1, pp. 47–51, 2016.

[3] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *Proc. SANER*, 2015, pp. 171–180.

[4] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs: How the current code review best practice slows us down," in *Proc. ICSE*, 2015, pp. 27–28.

[5] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proc. MSR*, 2014, pp. 192–201.

[6] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *Proc. SANER*, 2015, pp. 141–150.

[7] M. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review." *Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2015.

[8] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: Code reviewer recommendation in github based on cross-project and technology experience," in *Proc. ICSE*, 2016, pp. 222–231.

[9] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proc. ICSE*, 2011, pp. 541–550.

[10] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *Proc. ICSME*, 2014, pp. 271–280.

[11] J. Tsay, L. Dabbish, and J. Herbsleb, "Let ' s talk about it: Evaluating contributions through discussion in github." in *Proc. FSE*, 2014, pp. 144–154.

[12] Y. Ueda, A. Ihara, T. Hirao, T. Ishio, and K. Matsumoto, "How is IF statement fixed through code review? - a case study of qt project -," in *Proc. IWPD*, 2017, pp. 207–213.

[13] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.