How is IF Statement Fixed Through Code Review? -A case study of Qt project-

Yuki Ueda*, Akinori Ihara*, Toshiki Hirao *, Takashi Ishio*, and Kenichi Matsumoto*
* Graduate School of Information Science, Nara Institute of Science and Technology (NAIST), Japan
Email: {ueda.yuki.un7, akinori-i, hirao.toshiki.ho7, ishio, matumoto}@is.naist.jp

***Abstract*—Peer code review is key work to ensure the absence of software defects. To improve the review process, many code review tools provide OSS projects CI tests that automatically verify the code quality issues such as a code convention issue and a syntax issue. However, it does not cover every issues such as project policy issue and a code readability issue. In this study, our main goal is to understand how a code owner fixes conditional statement issues based on reviewers feedback. As a first step to achieve this goal, we conduct an empirical study to understand `if` statement changes after reviewing. Using 69,325 review requests in the Qt project, we analyze changes of the `if` conditional statement (1) that are requested to review, and (2) that are implemented after reviewing. As a result, we find that most common symbolic changes are "(" and ")" (35%) , "!" operator (20%) and "->" operator (12%). Also, "!" operator is frequently replaced with "(" and ")".**

I. Introduction

Peer code review, a manual inspection of code changes by developers who do not create them, is a well-established practice to ensure the absence of software defects. Nowaday, many open source software (OSS) and commercial projects have adapted the peer code review. While code review plays an important work in software development processes, it is expensive and time consuming [1]. For example, Alberts [2] describes how code review spends around 50% of the software development resources. One of the reasons is because patch authors spend much time to revising their own patches due to various issues (e.g., technical, feature, scope and process issues) [1].

What cases patch authors to fix their patches several times? Pan et al. [3] and Martinez et al. [4] conducted an empirical study to understand what code changes a patch author commit. They found that `if` statement change is the most frequent than any other changes. Also, Tan et al. [5] found that binary operators in conditional expression are more frequent changes in a programming contest. Why do developers often change `if` statement? In our best knowledge, little is known how the code owner fixes `if` statement. `if` statement changes would include various issues such as bug fix and reliability.

In this study, our main goal is to understand how patch author fixes conditional statement issues based on reviewers feedback. As a first step to achieve this goal, we conduct an empirical study to analyze `if` statement changes after reviewing because the conditional statement issues are likely to be changed more frequently than any other ones [3]. As a case study using 69,325 patches in the Qt project, we analyze the changes of the conditional statements (1) that are requested to review (Section IV), and (2) that are implemented after the review completion (Section V).

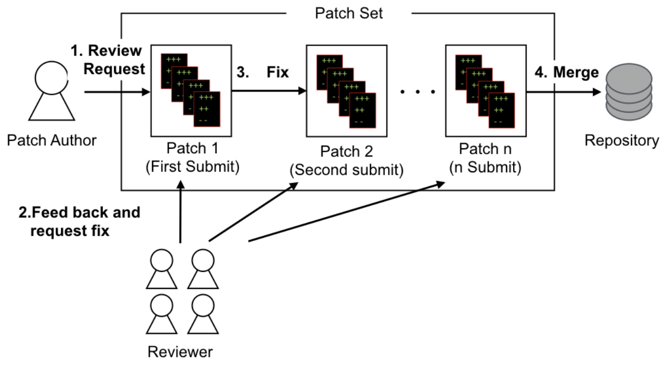

Fig. 1. The overview of the code review processes in Gerrit Code Review

The contribution of this study is to find out frequent patterns to fix `if` statement through code review. We conjecture that it may help to design an issue detection approach.

This paper is structured as follows. Section II describes the background of our study. Section III introduce our target `if` statement changes. Section IV describes an empirical study to analyze the changes in code review requests, and Section V describes to analyze the changes after reviewing. Section VI describes the validity of our empirical study. Section VII introduce the related works. Finally, section VIII concludes our study and discuss future works.

II. BackGround

Nowadays, we have various dedicated tools for managing the peer code review processes. For example, Gerrit [1] and ReviewBoard [2] are commonly used by OSS practitioners to receive the lightweight reviews. Technically, the code review tools are used for patch submission trigger, an automatic test and manual reviewing to decide whether or not a patch should be integrated into a version control system.

For automatic test, OSS projects often use CI (Continuous Integration) tests that automatically verify the fundamental

[1]Gerrit Code Review: https://code.google.com/p/gerrit/
[2]ReviewBoard: https://www.reviewboard.org/

flaws by such as Jenkins [3], Travis CI [4]. However, it is not able to cover high level (e.g., requirement) issues such as performance and security issues [6]. To detect these issues, reviewers may need to conduct manual review with their eyes.

Fig. 1 shows an overview of the code review process in Gerrit Code Review which our target Qt project is using as a code review management tool.

1. A patch author submit a patch to Gerrit Code Review. We define the submitted patch is $Patch_1$.
2. The reviewer(s) evaluate the $Patch_1$, and provide feedback to the patch author. If the $Patch_1$ has any issues, the reviewer(s) require to revise the patch.
3. The patch author then revises the $Patch_1$, then submit the revised patch as a $Patch_2$. If the patch is revised n times, we define the patch as $Patch_n$.
4. Once the patch author complete to address concerns of reviewers, the patch will be integrated into the repository.

Raymond et al. [7] mentioned that code review is able to detect crucial issues a large-scale code before releasing. Indeed, the validity of code review has been demonstrated by many prior studies [8], [9], [10], [11], [12]. The prior studies show the relationship of software defects after release, anti-patterns in software design and security vulnerability issues.

While code review is effective to improve the quality of software artifacts, it spend a large amount of time and request many human resources [2]. Indeed, Rigby et al. [13] found that six large-scale OSS projects spend approximately 1 month for code review. There are mainly two reasons why code review spends a large amount of time and human resources. The first factor is a process to identify the appropriate reviewer(s) before the code review starts. Previous research proposes a method of selecting an appropriate reviewer based on past reviewer's experience [14], [15], [16], [17], [18]. The second factor is the process of reviewing source codes. The code changes suggested by developers has some problems and need to be fixed multiple times [1]. Sometimes, when reviewers disagree with one another, the review time is likely to be longer [19].

Most published code review studies focused on review process or reviewers communication. In this paper, we focused source code changes, especially `if` changes, to reveal why submitted code were changed. We believe `if` statement causes many source code changes in software development [3], [4]. Especially, Tan et al. [5] discussed logical operators in conditional expression are frequently fixed in programming contests. We believe that `if` statement changes are one of the most difficult works. However, how `if` statement was not clear in previous studies.

In this paper, we conduct two analyses. The first one is to analyze what kinds of symbols in $Patch_1$ are likely to be fixed in a code review request. The second one is to analyze what kinds of symbol changes are likely to be fixed after reviewing. In second analysis, we investigate changes between $Patch_1$ and $Patch_n$ in Fig. 1. The second one is to analyze what kinds of symbol changes between $Patch_1$ and $Patch_n$ after reviewing.

[3]Jenkins: https://jenkins.io/index.html

[4]Travis CI: https://travis-ci.org/

Listing 1. IF-CC pattern Example

```
- if (getView().countSelected() == 0){
+ if (getView().countSelected() <= 1){
```

Listing 2. Example 1

```
- if (n >= 1 && path.at(0) == QLatin1Char
    ('/'))
+ if (n == 0)
+       return false;
+ const QChar at0 = path.at(0);
+ if (at0 == QLatin1Char('/'))
          return true;
```

III. Conditional Statement in Code Review

IF-CC pattern is changing the condition expression of an `if` condition like Listing 1. Pan et al. [3] found that conditional statements are most likely to be fixed than another syntax issues and the common pattern of conditional statement that are changed by developers (defined as IF-CC pattern)

In the Listing 1, although the IF-CC pattern is able to detect the condition change, it does not detect what string developer change in `if` statement (e.g., the change from "==" operator to "<=" operator). we focus how the conditional statements are changed in code reviews.

To investigate the possible patterns of conditional statements, we conduct the empirical study on the Qt project. The Qt project is a cross-platform application framework using C++ language that is supported by the Digia corporation [5]. In our study, we target 69,325 review requests in Qt project. We sample 380 patches from the original review dataset (a sample selected to obtain proportion estimates that are within 5% bounds of the proportion with 95% confidential level). Listings 2 through 4 show the example patterns that we manually extracted.

Listings 2 is an example devide `if` statement that using "`&`" symbol [6]. Listings 3 is an example remove "`(`", "`)`" with De Morgan's laws[7]. Listings 4 is an example replace "`(`" and "`)`" to function [8]

Listings 2 through 4 describe symbol changes, In Listings 2, "`&`" operator disappear. In Listings 3, "`&`" operator symbol is replaced with "`|`" operator symbol, and "`!`" operator increase. In Listings 4, "`!=`" operator and "`==`" operator are replaced with function. Then, "`(`" and "`)`" arise.

[5]http://qt.digia.com/

[6]https://codereview.qt-project.org/#/c/16570/1..2/src/lib/tools/fileinfo.cpp

[7]https://codereview.qt-project.org/#/c/53881/1..3/src/libs/utils/consoleprocess.cpp

[8]https://codereview.qt-project.org/#/c/6041/1..6/src/plugins/geoservices/nokia/places/qplacesuppliersrepository.cpp

Listing 3. Example 2

```
– if (!(nonEmpty && value.isEmpty()))
+ if (!nonEmpty || !value.isEmpty())
```

Listing 4. Example 3

```
– if (target.icon() == QPlaceIcon() &&
    src.icon() != QPlaceIcon())
+ if (target.icon().isEmpty() && !src.
    icon().isEmpty())
```

We conduct two quantitative analysis on Qt project.

In our first analysis, we analyze what kinds of code changes of conditional statements are added in a first submitted patch. In our second analysis, we analyze what kinds of code changes of conditional statements are fixed after reviewing.

IV. Analysis 1: Source Code Changes in Review Request

A. Approach

For analysis 1, this study analyze `if` statement changes included in the diff file which is generated by the review management system for reviewing. Indeed, we would like to analyze the changes based on the diff file and the original source code to identify a spot with `if` statement. However, it takes time to collect the original source code. When we use only diff file to analyze `if` statement changes, there is one limitation which is not able to get all `if` changed contents across multiple lines. However, we believe that problem will not affect to our results since the number of `if` statement changes across multiple lines is 425 patches of our target 69,325 patches (0.006%).

For this analysis, we identify the number of each symbols in the condition of `if` statement changes in $Patch_1$ by syntax analysis. Then, we use ANTLR [9] for parsing syntax and count changed symbols for each changed block. Fig.2 described overview to get symbol changes. Table I shows common symbol, the description, and example in conditional expression.

To analyze frequent of changed symbols in the same change, we use closed frequent itemset mining which is one of the well-known and popular data mining methods. The frequent itemset mining is to identify a frequent item sets that is both closed and its support is greater than the any other item sets. For frequent itemset mining, we use `arules` package of R. Since the frequent itemset mining suggest much item sets, we target item sets with less than 7 items and the support values are 0.001 or more to filter out some item sets. When we find the item sets have inclusion relation (e.g., {“(”} and {“(”, “)”}) with same support values, the set with fewer items will be filtered out.

[9]http://www.antlr.org/

TABLE I
Symbol list.

symbols	purpose	example
=	assignment	if((a = b))
==	compare same or not	if(a == b)
!=	compare not same or same	if(a != b)
!	switch logical result	if(!a)
&	and condition	if(a & b)
|	or condition	if(a | b)
(	surround condition or call function	if((a | b) & c())
)	surround condition or call function	if((a | b) & c())
->	call member from pointer	if(a->b())
+	plus operator	if((a + b) == 0)
-	minus operator	if((a - b) == 0)
*	multiple operator	if((a * b) == 0)
/	devide operator	if((a / b) == 0)
%	ramaind operator	if((a % b) == 0)
<	compare(greater than)	if(a < b)
>	compare(less than)	if(a > b)
<=	compare(greater than or equal)	if(a <= b)
>=	比較 (less than or equal)	if(a >= b)

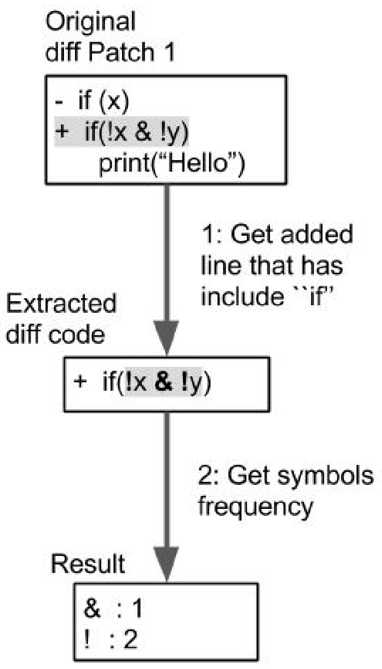

Fig. 2. Approach to extract changed symbols in `if` statement from diff file.

B. Result

In our target 69,325 review requests, we found that there were 6,956 requests (10%) including the change of `if` statement. Fig.3 shows the frequent of symbols changed in `if` statements of the submitted patch. And, the Table II shows the top 50 item sets with higher support value of 477 item sets analyzed by frequent itemset mining. For example, id3 in table II shows “ (” ∧ “) ” that means “ (” and “) ” was changed in the same time of `if` statement changes.

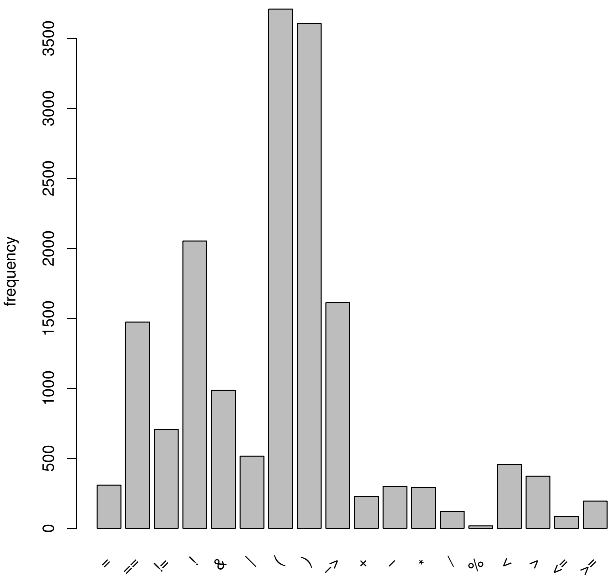

Fig. 3. Symbol change frequency with `if` statement changes.

Listing 5. Example include "(" and "("

```
+   if (!QFileInfo(systemRoot() + "/epoc32
      /release/udeb/epoc.exe").exists())
+       return false;
```

Observation 1: Parentheses(e.g., "(" and ")") is the most frequent changed symbol with "if" statement changes. Fig. 3 shows that parentheses are likely to be fixed with `if` statement. Also, Table II shows 55% of the "if" statement changes including "(" and ")" changes (id1). This target parentheses don't include begin or end parenthesis for if statement. Although parentheses are often used for define priority in statement, we interestingly found that Qt project often uses parentheses to call a function in the statement like Listing 5 [10].

Observation 2: "Arrow" operator and "not" symbol are likely to be added with parentheses frequency. Fig. 3 shows that "Arrow" and "not" are likely to be fixed with `if` statement. In Table II, 55% of `if` statement changes is with the adding parentheses. The 44.7% of them (24.6% of `if` statement changes) were changed with the adding the arrow operator (id12 to id19) since the arrow operator often call a function like Listing 6 [11].

Also, 34.7% of them (19.1% of `if` statement changes) were changed with the adding the not "!" operator (id8 to id10) since the not operator is often used to detect the fail of function

[10]https://codereview.qt-project.org/#/c/1368/1/src/plugins/qt4projectmanager/qt-s60/symbianqtversion.cpp

[11]https://codereview.qt-project.org/#/c/32/1/src/plugins/texteditor/plaintexteditorfactory.cpp

Listing 6. Example include "->"

```
+ if (editor->file()->hasHighlightWarning
    ())
      return;
```

Listing 7. Example include "!"

```
+ if (!isComponentComplete() || !d->model
    || !d->model->isValid())
      return;
```

execution like Listing 5 or to use the output of a function like Listing 7[12].

V. Analysis 2: Source Code Changes after Reviewing

A. Approach

For analysis 2, we analyze how code owner fixed `if` statements after reviewing. As Fig. 4 shows, we count the number of symbol which the code owner changes from $Patch_1$ to $Patch_{merged}$. For example, in Fig. 4, we find one `&` and two in $Patch_1$. After reviewing the $Patch_1$, we can find that the patch owner added "`|`", "(" and ")" as , and deleted "`&`" and "`!`". Then, we count the difference number of symbols between $Patch_1$ and $Patch_{merged}$ like "&_*delete*: 1". Using this dataset, this analysis conducts an empirical study to understand the updates after reviewing using the same frequent itemset mining as Analysis 1.

B. Result

Table III shows top 50 rules with the highest support score in 364 rules which were detected by frequent itemset mining.

Observation 3: 35% of code fixes after reviewing includes the adding or deleting parentheses. 23% of all fixes with `if` statements includes the adding parentheses (id1 in Table III). On the other hand, 12% of all fixes with `if` statements includes the deleting parentheses (id5-id9 in Table III). Totally, 35% of all fixes with `if` statements includes the parentheses fixes because of too many function calls or lacking function calls like the following examples [13],[14].

Observation 4: Patch owner is likely to add "->", "&" after reviewing. 8% of all fixes with `if` statements includes the adding "->" (id15 in Table III). The number of the adding "->" is more than the deleting one (id27 in Table III). Same as "->", 7% of all fixes with `if` statements includes the adding "`&`" (id22 in Table III) like the Listing 8.

Observation 5: Patch owner is likely to delete "!" after reviewing. 13% of all fixes with `if` statements includes the

[12]https://codereview.qt-project.org/#/c/2481/1/src/declarative/items/qsggridview.cpp

[13]https://codereview.qt-project.org/#/c/1843/1..2/src/plugins/qt4projectmanager/qt-desktop/simulatorqtversion.cpp

[14]https://codereview.qt-project.org/#/c/1779/1..2/src/plugins/qmlprojectmanager/qmlprojectruncontrol.cpp

TABLE II
FREQUENT OF CHANGED SYMBOL SETS WITH IF CHANGES.

id	symbols	support * 100
1	“(”	56.7
2	“)”	55.1
3	“(” ∧ “)”	55.1
4	“!”	31.4
5	“->”	24.6
6	“==”	22.5
7	“(” ∧ “->”	19.4
8	“!” ∧ “(”	19.4
9	“!” ∧ “)”	19.1
10	“!” ∧ “(” ∧ “)”	19.1
11	“)” ∧ “->”	19.1
12	“(” ∧ “)” ∧ “->”	19.0
13	“&”	15.1
14	“==” ∧ “(”	12.5
15	“&” ∧ “(”	12.1
16	“==” ∧ “(” ∧ “)”	11.8
17	“&” ∧ “)”	11.7
18	“&” ∧ “(” ∧ “)”	11.7
19	“!=”	10.8
20	“!” ∧ “->”	8.8
21	“|”	7.9
22	“!” ∧ “(” ∧ “->”	7.5
23	“!” ∧ “)” ∧ “->”	7.4
24	“!” ∧ “(” ∧ “)” ∧ “->”	7.4
25	“!=” ∧ “(”	7.2
26	“!=” ∧ “(” ∧ “)”	7.1
27	“<”	7.0
28	“|” ∧ “(”	6.5
29	“|” ∧ “(” ∧ “)”	6.4
30	“==” ∧ “->”	6.1
31	“&” ∧ “->”	6.0
32	“>”	5.7
33	“!” ∧ “&”	5.6
34	“&” ∧ “(” ∧ “->”	5.5
35	“&” ∧ “(” ∧ “)” ∧ “->”	5.4
36	“(” ∧ “<”	5.2
37	“(” ∧ “)” ∧ “<”	5.2
38	“==” ∧ “&”	5.2
39	“!” ∧ “&” ∧ “(”	5.1
40	“!” ∧ “&” ∧ “)”	5.0
41	“!” ∧ “&” ∧ “(” ∧ “)”	5.0
42	“==” ∧ “(” ∧ “->”	4.7
43	“=”	4.7
44	“==” ∧ “(” ∧ “)” ∧ “->”	4.7
45	“−”	4.6
46	“*”	4.5
47	“==” ∧ “&” ∧ “(”	4.3
48	“(” ∧ “>”	4.3
49	“)” ∧ “>”	4.2
50	“(” ∧ “)” ∧ “>”	4.2

deleting “!” (id4 in Table III). The number of the deleting “!” is more than the adding one (id16 in Table III). Same as “->”, 7% of all fixes with `if` statements includes the adding “&” (id22 in Table III) like the Listing 8. Especially, 69% of the deleting “!” (9% of all fixes with `if` statements) is fixed with the adding “(”, “)” because the patch owner often use any functions instead of “!” like Listing10[15]. The other case, “==” is also likely to delete after reviewing because “==” also replace to function(e.g., isEmpty()).

[15]https://codereview.qt-project.org/#/c/2422/1..8/src/declarative/items/qsgcanvas.cpp

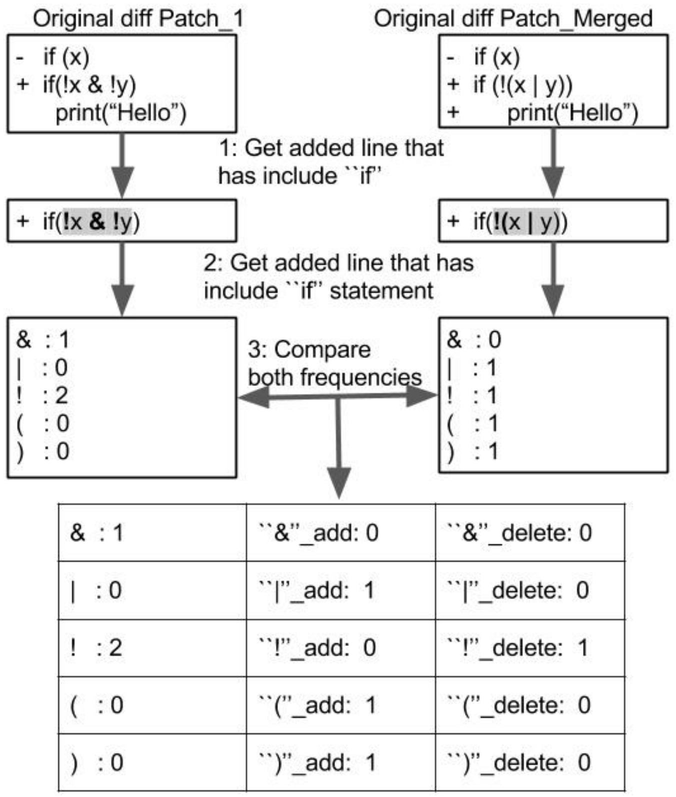

Fig. 4. Approach to extract changed symbols after reviewing.

Listing 8. Example add “(” and “)”
```
– if (qmlviewerCommand().isEmpty())
+ if (qtVersion() >= QtSupport::
    QtVersionNumber(4, 7, 0) &&
    qmlviewerCommand().isEmpty())
```

VI. THREATS TO VALIDITY

Internal validity. To analyze code changes before and after reviewing, we extracted symbol changes in textttif statement changes by syntax analysis.Indeed, we would like to analyze the changes based on the diff file and the original source code to identify a spot with `if` statement. However, since it takes time to collect the original source code, we simply focus on only diff files. we believe that any problems will not affect to our results since the number of `if` statement changes across multiple lines is 425 patches of our target 69,325 patches (0.006%).

External validity. We conduct the empirical study using only Qt project code review dataset. When we target the other projects, some findings of our study may be different. For example, the other project may be often use “<” or “>=” in Table 3 instead of “>” or “<=”. We believe that our approach would be helpful to understand individual rules or trend fixes in each project.

TABLE III
SAME TIME CHANGED ITEMS IN CODE REVIEW

id	symbols	support ⋆ 100
1	“(”_add	23.32
2	“)”_add	23.32
3	“(”_add ∧ “)”_add	23.09
4	“!”_delete	13.27
5	“)”_delete	12.46
6	“(”_delete	12.20
7	“(”_delete ∧ “)”_delete	12.05
8	“(” ∧ “)”_delete	12.00
9	“)” ∧ “(”_delete	11.59
10	“!” ∧ “)”_add	11.24
11	“!” ∧ “(”_add	11.24
12	“!” ∧ “(”_add ∧ “)”_add	11.18
13	“(”_add ∧ “!”_delete	9.12
14	“(”_add ∧ “)”_add ∧ “!”_delete	9.09
15	“->”_add	8.02
16	“!”_add	7.46
17	“(” ∧ “)”_add	6.94
18	“(” ∧ “(”_add	6.85
19	“(” ∧ “(”_add ∧ “)”_add	6.71
20	“(” ∧ “)” ∧ “(”_add	6.65
21	“(” ∧ “)” ∧ “)”_add	6.59
22	“&”_add	6.59
23	“(” ∧ “)” ∧ “(”_add ∧ “)”_add	6.51
24	if._add	4.97
25	“(”_add ∧ “->”_add	4.82
26	“(”_add ∧ “)”_add ∧ “->”_add	4.79
27	“->”_delete	4.71
28	“==”_delete	4.59
29	“(” ∧ “&”_add	4.47
30	“(” ∧ “)” ∧ “&”_add	4.39
31	“&”_delete	4.36
32	“==”_add	4.33
33	“!” ∧ “)”_delete	4.24
34	“(” ∧ “->”_delete	4.15
35	“!” ∧ “(” ∧ “)”_delete	4.12
36	“(” ∧ “)” ∧ “->”_delete	4.10
37	“(” ∧ “!”_add	4.10
38	“(” ∧ “)” ∧ “!”_add	4.04
39	“!” ∧ “(”_delete	3.98
40	“->” ∧ “(”_delete	3.95
41	“!” ∧ “(”_delete ∧ “)”_delete	3.92
42	“->” ∧ “)”_delete	3.92
43	“->” ∧ “(”_delete ∧ “)”_delete	3.89
44	“)” ∧ “->” ∧ “(”_delete	3.86
45	“!” ∧ “)” ∧ “(”_delete	3.86
46	“(” ∧ “->” ∧ “)”_delete	3.86
47	“(” ∧ “!”_delete	3.80
48	“(” ∧ “)” ∧ “!”_delete	3.78
49	“(” ∧ “->”_add	3.60
50	“(” ∧ “)” ∧ “&”_delete	3.60

Listing 9. Example delete “(” and “)”

```
– if (config->qtVersion() && QtSupport::
     QmlObserverTool::canBuild(config->
     qtVersion()))
+ if (QtSupport::QmlObserverTool::
     canBuild(config->qtVersion()))
```

Listing 10. Example delete “!”

```
–     if (!hoverItems)
+     if (hoverItems.isEmpty())
```

VII. RELATED WORK

A. Code Review

Many studies have conducted an empirical studies to understand code review works [8], [9], [10], [11], [12], [22], [23], [24], [25]. Most published code review studies focused review process or reviewers communication.

For example, Tsay et al. found that patch authors and reviewers often discuss and propose solutions to fix a patch each other [22]. In particula, Czerwonka et al. [23] found that 15% of the discussion for a patch fix is about functional issues. Also, Mäntylä et al. [24] and Beller et al. [25] found that 75% of the discussion for a patch fix is about software maintenance and 15% of the discussion is about functional issues. From these studies, we could understand issues which we should solve in code review process. Next, we should focus on how we fix those issues. As the first step, we focused source code changes through code review, especially `if` changes.

B. Coding Conventions

Appropriate coding conventions prevent software faults [**?**]. As a code fix study, refactoring study is the most popular one in software engineering field [20]. Particularly, code convention issue is much relate to our study. Smit et al. [20] found that `CheckStyle` is useful to check whether or not a source code follows their coding rule. Also, some convention tool has released to check a format of coding convention such as Pylint[16]. Furthermore, Allamanis et al. [21] developed a tool to fix code convention. However, in our best knowledge, little is known how a code owner fixes conditional statement issues based on reviewers feedback.

VIII. CONCLUSION

In this paper, our empirical study discuss how a patch author fix `if` statement based on reviewer feedback. According to the results of our case study on Qt project, While 55% of the “if” statement changes including “(” and “)” changes, 35% of code fixes after reviewing includes the adding or deleting parentheses. In the most of cases, a patch author often add parentheses to call a function. In addition, we found “->” and “&” are likely to be added, and “!” is likely to be deleted after reviewing. These might be changed to fix a potential bug. And, if patch authors check the possible to change these before request code review, the reviewers could spend more time for the other review requests. In the future, we would like to propose a method to review and fix a symbol in `if` statement automatically.

ACKNOWLEDGMENT

Support Center for Advanced Telecommunications (SCAT) Technology Research, Foundation This work was supported by Grant-in-Aid for JSPS Research Fellow Number 17J09333.

[16]https://www.pylint.org/

REFERENCES

[1] P. C. Rigby and M.-A. Storey, “Understanding broadcast based peer review on open source software projects,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 541–550.

[2] D. S. Alberts, “The economics of software quality assurance,” in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*. New York, NY, USA: ACM, 1976, pp. 433–442.

[3] K. Pan, S. Kim, and E. J. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[4] M. Martinez, L. Duchien, and M. Monperrus, “Automatically extracting instances of code change patterns with AST analysis,” *IEEE International Conference on Software Maintenance, ICSM*, pp. 388–391, 2013.

[5] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, “Codeflaws: a programming competition benchmark for evaluating automated program repair tools,” in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 180–182.

[6] Y. Tao, D. Han, and S. Kim, “Writing acceptable patches: An empirical study of open source project patches,” in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2014, pp. 271–280.

[7] E. S. Raymond, “The cathedral and the bazaar,” *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[8] 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 *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 192–201.

[9] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis, “An empirical investigation of socio-technical code review metrics and security vulnerabilities,” in *Proceedings of the 6th International Workshop on Social Software Engineering*, 2014, pp. 37–44.

[10] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Investigating code review practices in defective files: An empirical study of the qt system,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015, pp. 168–179.

[11] 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 *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 171–180.

[12] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “An empirical study of the impact of modern code review practices on software quality,” *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[13] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.

[14] 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 *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 141–150.

[15] V. Balachandran, “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation,” in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 931–940.

[16] 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.

[17] M. M. Rahman, C. K. Roy, and J. A. Collins, “Correct: Code reviewer recommendation in github based on cross-project and technology experience,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 222–231.

[18] X. Xia, D. Lo, X. Wang, and X. Yang, “Who should review this change? putting text and file location analyses together for more accurate recommendations.” in *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, 2015, pp. 261–270.

[19] T. Hirao, A. Ihara, Y. Ueda, P. Phannachitta, and K. Matsumoto, “The impact of a low level of agreement among reviewers in a code review process,” in *The 12th International Conference on Open Source Systems*, 2016, pp. 97–110.

[20] J. Tsay, L. Dabbish, and J. Herbsleb, “Let ’ s talk about it: Evaluating contributions through discussion in github.” in *Proceedings of the 22nd International Sym posium on Foundations of Software Engineering*, 2014, pp. 144–154.

[21] J. Czerwonka, M. Greiler, and J. Tilford, “Code reviews do not find bugs: How the current code review best practice slows us down,” in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 27–28.

[22] M. V. Mäntylä and C. Lassenius, “What types of defects are really discovered in code reviews?” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2009.

[23] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, “Modern code reviews in open-source projects: Which problems do they fix?” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 202–211.

[24] C. Boogerd and L. Moonen, “Assessing the value of coding standards: An empirical study,” *IEEE International Conference on Software Maintenance, ICSM*, pp. 277–286, 2008.

[25] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, “Code convention adherence in evolving software,” in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 504–507.

[26] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.