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 to ensuring the absence of software defects. To improve the review process, many code review tools provide OSS(Open Source Software) project CI(Continuous Integration) tests that automatically verify code quality issues such as code convention issues. However, these tests do not cover project policy issues and code readability issues. In this study, our main goal is to understand how a code owner fixes conditional statement issues based on reviewers feedback. We conduct an empirical study to understand `if` statement changes after review. Using 69,325 review requests in the Qt project, we analyze changes of the `if` conditional statements that (1) are requested to be reviewed, and (2) that are implemented after review. As a result, we find the 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 role in software development processes, it is expensive and time consuming [1]. For example, Alberts [2] describes how code review uses around 50% of the software development resources. One of the reasons is because patch authors spend a lot of time to revising their own patches due to various issues (e.g., technical, feature, scope and process issues) [1].

What causes 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 changes are the most frequent 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` statements? To our knowledge, little is known about how the code owner fixes an `if` statement. `if` statement changes would include various issues such as bug fixes and reliability.

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

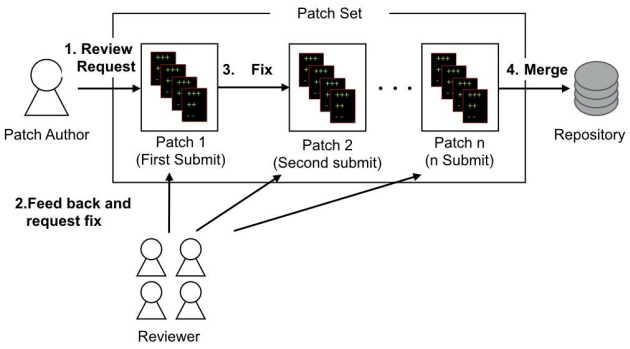

Fig. 1. Overview of the code review processes in Gerrit Code Review

The contribution of this study is that it finds out frequent patterns to fix `if` statements through code review. We think this in turn 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 our analysis of the changes after reviewing. Section VI describes the validity of our empirical study. Section VII introduces related works. Finally, Section VIII concludes our study and discusses 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, these code review tools are used for patch submission triggers, automatic tests and manual reviewing to decide whether or not a patch should be integrated into a version control system.

For automatic tests, 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/

978-1-5386-2387-9/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSREW.2017.32

flaws following Jenkins [3], and Travis CI [4]. However, these tests cannot cover high level (e.g., requirement) issues such as performance and security issues [6]. To detect these issues, reviewers may need to conduct reviews manually with their own eyes.

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

1. A patch author submits a patch to Gerrit Code Review. We define the submitted patch as $Patch_1$.
2. The reviewer(s) evaluate $Patch_1$, and provide feedback to the patch author. If $Patch_1$ has any issues, the reviewer(s) are required to revise the patch.
3. The patch author then revises $Patch_1$, and submits 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 completely addresses the concerns of reviewers, the patch will be integrated into the repository.

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

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

Most published code review studies focused on review processes or reviewers' communications. In this paper, we focus on source code changes, especially `if` changes, to reveal why submitted codes were changed. We believe `if` statements cause many source code changes in software development [3], [4]. Tan et al. [5], in particular, discussed how logical operators in conditional expression are frequently fixed in programming contests. We believe `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 analysis investigates what kinds of symbols in $Patch_1$ are likely to be fixed in a code review request. The second analysis investigates what kinds of symbol changes are likely to be fixed after reviewing. In second analysis, we investigate changes between $Patch_1$ and $Patch_n$ after review (as shown in Fig. 1).

[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 Statements in Code Review

As IF-CC pattern is changes the conditional expression of an `if` condition as in Listing 1. Pan et al. [3] found that conditional statements are more likely to be fixed than another syntactic issues and they described common patterns of conditional statements that are changed by developers (defined as the IF-CC pattern)

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

To investigate the possible patterns of conditional statements, we conduct an 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 have manually extracted.

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

Listings 2 through 4 describe symbol changes, In Listings 2, the "`&`" operator disappears. In Listings 3, the "`&`" operator symbol is replaced with an "`|`" operator symbol, and the

[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())
```

"!" operator increase. In Listings 4, the "!=" operator and "==" operator are replaced with a function. Then, "(" and ")" arises.

We conduct two quantitative analyses on the Qt project.

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

IV. Analysis 1: Source Code Changes in the Review Request

A. Approach

For analysis 1, we investigate `if` statement changes included in the diff file which is generated by the review management system. Indeed, we would like to analyze the changes based on the diff file and the original source code to identify the spot with the `if` statement. However, it takes time to collect the original source code. When we use only the diff file to analyze `if` statement changes, there is one limitation, however; we are not able to get all of the `if` changed contents across multiple lines. However, we believe that this problem will not affect to our results since the number of `if` statement changes across multiple lines is 425 patches out 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 syntactic analysis. Then, we use ANTLR [9] to parse the syntax and count the changed symbols for each changed block. Fig.2 describes how we got the symbol changes. Table I shows common symbols, their description, and examples in conditional expressions.

To analyze the frequency of changed symbols in the same change, we use closed frequent itemset mining, which is a well-known and popular data mining methods. Frequent itemset mining identifies a frequent item sets that are both closed with support that is greater than the any of the other item sets. For frequent itemset mining, we use the `arules` package [10] for R. Since the frequent itemset mining results in many item sets, we target item sets with less than 7 items and support values 0.001 or more to filter out some item sets.

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

[10]https://cran.r-project.org/web/packages/arules/index.html

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)
/	divide operator	if((a / b) == 0)
%	ramainder 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)
>=	compare(less than or equal)	if(a >= b)

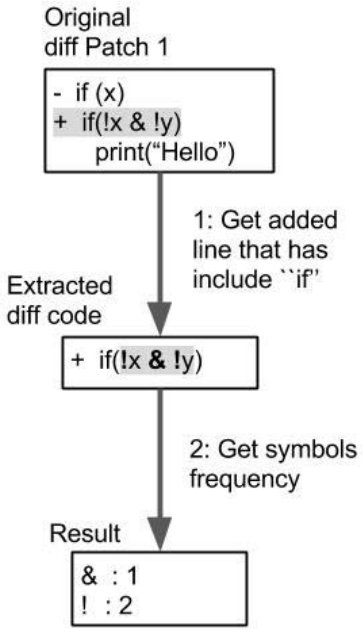

Fig. 2. Approach to extract changed symbols in `if` statement from diff file.

When we find the item sets that have an inclusion relation (e.g., {"("} and {"(", ")"}) with same support values, the set with fewer items will be filtered out.

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 frequency of symbols changed in `if` statements of the submitted patch. In addition, Table II shows the top 50 item sets with a higher support value of 477 item sets analyzed by frequent itemset mining. For example, id3

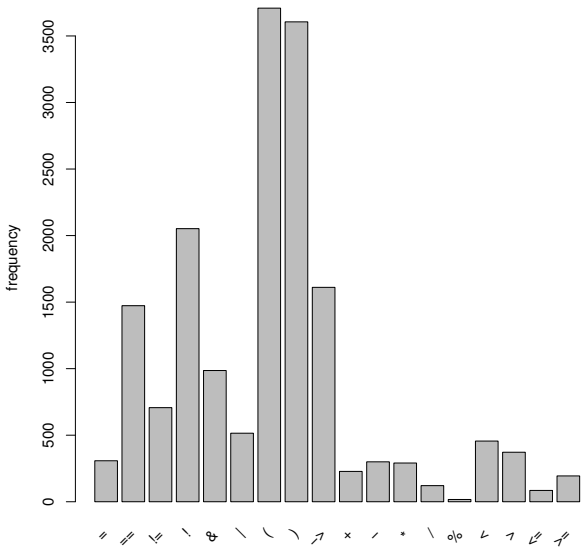

Fig. 3. Symbol change frequency with `if` statement changes.

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

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

in table II shows "(" ∧ ")" witch means "(" and ")" was changed at the same time as the `if` statement changes.

Observation 1: Parentheses(e.g., "(" and ")") is the most frequently changed symbol with "if" statement changes. Fig. 3 shows that parentheses are likely to be fixed with an `if` statement. Also, Table II shows 55% of the "if" statement changes including "(" and ")" changes (id1). This target parentheses does not include begin or end parenthesis for the if statement. Although parentheses are often used to define priority in statement, we interestingly found that the Qt project often uses parentheses to call a function in the statement such as in Listing 5 [11].

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 the `if` statement changes are with the adding parentheses. 44.7% of them (24.6% of `if` statement changes) were changed with adding the arrow operator (id12 to id19) since the arrow operator often calls a function such as in Listing 6 [12].

Also, 34.7% (19.1% of `if` statement changes) were changed

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

[12]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;
```

adding the not "!" operator (id8 to id10) since the not operator is often used to detect the fail of a function execution as in Listing 5 or to use the output of a function as in Listing 7[13].

V. Analysis 2: Source Code Changes after Review

A. Approach

For analysis 2, we investigated how the code owner fixed `if` statements after review. As Fig. 4 shows, we counted the number of symbols the code owner changed from $Patch_1$ to $Patch_{merged}$. For example, in Fig. 4, we find one `&` and two "!" in $Patch_1$. After reviewing $Patch_1$, we find that the patch owner added "|", "(" and ")" as, and deleted "&" and "!". Next, we counted the difference in the number of symbols between $Patch_1$ and $Patch_{merged}$ such as "&_$delete$: 1". Using this dataset, we conducted an empirical study to understand the updates after reviews using the same frequent itemset mining as in Analysis 1.

B. Result

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

Observation 3: 35% of the code fixes after review included the adding or deleting of parentheses. 23% of all fixes with `if` statements included the adding parentheses (id1 in Table III). On the other hand, 12% of all fixes with `if` statements included the deleting parentheses (id5-id9 in Table III). In total, 35% of all fixes with `if` statements included the parentheses fixes because of too many function calls or because of the lack of function call as in the following examples [14],[15].

Observation 4: The patch owner is likely to add "->", "&" after review. 8% of all fixes with `if` statements include adding "->" (id15 in Table III). The number of addition of "->" is more than the deletions (id27 in Table III). Likewise, when adding "->", 7% of all fixes with `if` statements include adding "&" (id22 in Table III) as in Listing 8.

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

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

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

TABLE II
FREQUENCY 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

Observation 5: The patch owner is likely to delete “!” after review. 13% of all fixes with `if` statements include the deletion of “!” (id4 in Table III). The number of the deletion of “!” is more than thir additions (id16 in Table III). Likewise, when adding “->”, 7% of all fixes with `if` statements include the addition of “&” (id22 in Table III) as in Listing 8. In particular, 69% of the deletions “!” (9% of all fixes with `if` statements) are fixed with the addition of “(”, “)” because the patch owner often uses functions other than “!” as in Listing10[16]. In the other case, “==” is also likely to be deleted after review because “==” also replaces a function(e.g., isEmpty()).

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

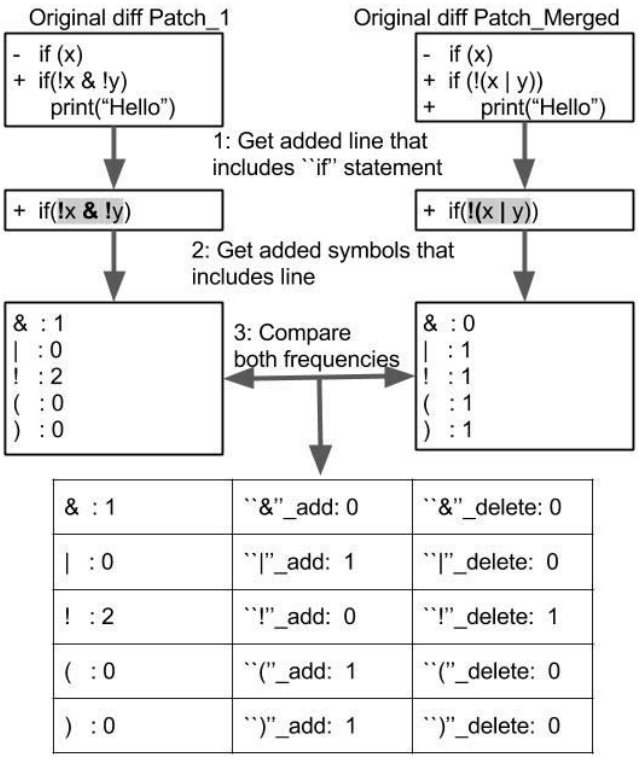

Fig. 4. Approach to extract changed symbols after reviewing.

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 review, we extracted symbol changes in `if` statement changes by syntactic analysis. We would prefer to analyze the changes based on the diff file and the original source code to identify spots with `if` statements. However, since it takes time to collect the original source code, we simply focus on diff files. We believe no problems will 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 conducted our empirical study using only the Qt project code review dataset. When we target the other projects, some findings of our study may be different. For example, other projects may use “<” or “>=” in Table 3 instead of “>” or “<=”. We believe that our approach will be helpful in understanding 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

VII. RELATED WORK

A. Code Review

Many researchers have conducted empirical studies to understand code review [8], [9], [10], [11], [12], [20], [21], [22], [23]. However, most published code review studies focus on the review process or reviewers' communication.

For example, Tsay et al. found that patch authors and reviewers often discuss and propose solutions with each other to fix patches [20]. In particular, Czerwonka et al. [21] found that 15% of the discussions for patch fixes are about functional issues. Also, Mäntylä et al. [22] and Beller et al. [23] found that 75% of discussions for a patch fixes are about software maintenance and 15% are about functional issues. These studies help us understand which issues we should be solved in the code review process and our work focuses on how we fix those issues. As the first step, we focused source code changes through code review, especially `if` changes.

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())
```

B. Coding Conventions

Appropriate coding conventions prevent software faults [24]. In code fix studies, refactoring studies are the most popular in the software engineering field [25]. Code convention issues also relate to our study. Smit et al. [25] found that `CheckStyle` is useful for checking whether or not source code follows its coding rules. Also, some convention tools such as Pylint[17] have been released to check the format of coding conventions . In addition, Allamanis et al. [26] has developed a tool to fix code conventions. However, to the best of our knowledge, little is known about how a code owner fixes conditional statement issues based on reviewers feedback.

VIII. CONCLUSION

In this paper, our empirical study analysed how a patch author can fix `if` statements based on reviewer feedback. The results of our case study on the Qt project showed that while 55% of the “if” statement changes included “(” and “)” changes, 35% of code fixes after reviewing included the addition or deletion of parentheses. In the most cases, a patch author added parentheses to call a function. In addition, we found “->” and “&” are likely to be added, and “!” is likely to be deleted after review. If a patch author checks the possibility of to changing these symbols before the request code review, the reviewers might be able to have more time to spend on other additional review requests. In the future, we would like to propose a method to review and automatically fix a symbol in `if` statements.

ACKNOWLEDGMENT

We would like to thank the Support Center for Advanced Telecommunications (SCAT) Technology Research, Foundation. This work was supported by JSPS KAKENHI Grant Number JP17J09333 and 17H00731.

[17]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 National Computer Conference and Exposition*, 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'13)*, 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 (ICSE'17)*, 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 (MSR'14)*, 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 (SSE'14)*, 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 (MSR'15)*, 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 (SANER'15)*, 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 (FSE'13)*, 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 (SANER'15)*, 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 (ICSE'13)*, 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 (ICSME'15)*, 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 (OSS'16)*, 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 Symposium on Foundations of Software Engineering (FSE'14)*, 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 (ICSE'15)*, 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 (MSR'14)*, 2014, pp. 202–211.

[24] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'08)*, 2008, pp. 277–286.

[25] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*. 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 International Symposium on Foundations of Software Engineering (FSE'14)*, 2014, pp. 281–293.