PAPER

How are IF-Conditional Statements Fixed Through Peer CodeReview?

Yuki UEDA[†], ***Nonmember***, **Akinori IHARA**[†], **Takashi ISHIO**[†], ***Members***, **Toshiki HIRAO**[†], ***Nonmember***, ***and*** **Kenichi MATSUMOTO**[†], ***Fellow***

SUMMARY Peer code review is key to ensuring the absence of software defects. To reduce review costs, software developers adopt code convention checking tools that automatically identify maintainability issues in source code. However, these tools do not always address the maintainability issue for a particular project. The goal of this study is to understand how code review fixes conditional statement issues, which are the most frequent changes in software development. We conduct an empirical study to understand `if`-statement changes through code review. Using review requests in the Qt and OpenStack projects, we analyze changes of the `if`-conditional statements that are (1) requested to be reviewed, and are (2) revised through code review. We find the most frequently changed symbols are "`( )`", "`.`", and "`!`". We also find project-specific fixing patterns for improving code readability by association rule mining. For example "`!`" operator is frequently replaced with a function call. These rules are useful for improving a coding convention checker tailored for the projects.

key words: *CodeReview, If statement, Code readability*

1. 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. Many open source software (OSS) and commercial projects have adopted the peer code review.

Code review requires much time [1]. Code review requires about 50% of the overall software development resources [2]. For example, patch authors tend to spend a long time revising their own patches due to technical issues [1]. Indeed, 75% of discussions in code review are about maintainability issues [3], [4].

To reduce the review cost, software developers adopt code convention checking tools that automatically identify general maintainability issues in source code. However, these tools just cover the general issues [5], in particular, programming languages. To completely detect maintainability issues, completely, reviewers may need to conduct code reviews manually with their own eyes.

The goal of our study is to understand how patch authors fix maintainability issues based on reviewers' feedback. To understand maintainability issues, we focus on `if`-statement changes. Previous studies reported that since a change in `if`-statements frequently occur [6], [7], improving `if`-statement readability is important [8]. Tan et al. [9] also reported that binary operators in the conditional expression are more frequent changes in a programming contest. While `if`-statements are considered important, little is known about how `if`-statements are fixed in practical software development.

First, as a preliminary study, we identify frequently changed symbols of the `if`-conditional statements in submitted patches (Section 4). Secondly, we discover tacit fixing patterns between submitted patches and merged patches by using association rule mining (Section 5). As a case study, we target 69,325 patches in the Qt and 60,197 patches in the OpenStack project.

This paper is an extension of our previous study [10] in two ways. First, we analyze all symbols in programming languages (e.g., Arithmetic, Logical or Relational operators and String or Number literal). Second, the previous study obtained only Qt [†] project written in C++ language. This paper includes OpenStack [††] project written in Python to obtain language-independent results.

This paper is structured as follows. Section 2 describes the background to our study. Section 3 introduces our target `if`-statement changes. Section 4 details an empirical study to analyze the changes in code review requests, and Section 5 presents our analysis of the changes through code reviewing. Section 6 considers the validity of our empirical study. Section 7 introduces related works. Section 8 concludes our study and discusses future works.

2. Background

Various dedicated tools exist for managing the peer code review process. Gerrit Code Review [†††] and ReviewBoard [††††] 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.

Figure 1 shows an overview of the code review process in Gerrit Code Review which our target Qt and OpenStack projects that large OSS projects use as a code review management tool.

Manuscript received January 1, 2015.
Manuscript revised January 1, 2015.

[†]Graduate School of Information Science, Nara Institute of Science and Technology (NAIST), Japan

DOI: 10.1587/trans.E0.??.1

[†] `https://www.Qt.io/developers/`
[††] `https://www.OpenStack.org/`
[†††] Gerrit Code Review: `https://code.google.com/p/gerrit/`
[††††] ReviewBoard: `https://www.reviewboard.org/`

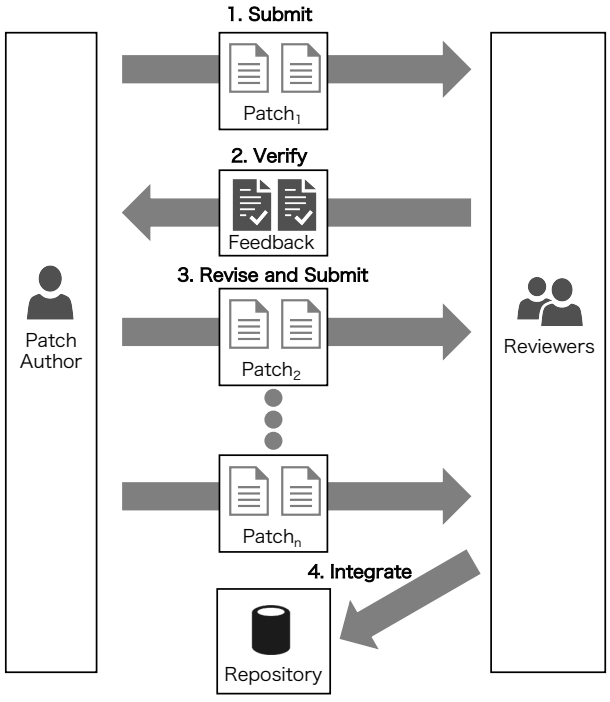


Fig. 1 Overview of the code review processes in Gerrit Code Review

1. A patch author submits a patch to Gerrit Code Review. We define the submitted patch as $Patch_1$.
2. The reviewers verify $Patch_1$. They send feedback and ask to revise the patch if it has any issues.
3. The patch author revises $Patch_1$ and submits the revised patch as $Patch_2$. The revision process may be repeated n times. We define the last patch as $Patch_n$.
4. Once the patch author completely addresses the concerns of the reviewers, the patch will be integrated into the project repository.

The validity of code review has been demonstrated by many prior studies [11]–[15]. Raymond et al. [16] discussed how code review is able to detect crucial issues in large-scale code before release. 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 improving the quality of software artifacts, it requires a large amount of time and many human resources [2]. Rigby et al. [17] found that six large-scale OSS projects needed approximately one month to integrate a patch. Reviewers may disagree with one another and take even longer for discussion [18]. The process also requires identifying appropriate reviewers for each patch. Various methods are proposed to select appropriate reviewers based on the reviewer's experience [19]–[23] and complexity of code changes [1].

Most published code review studies focused on review processes or the reviewers' communications. They do not focus on source code changes through the review. We focus on source code changes especially `if` changes to clarify how a submitted patches were revised, because `if`-statements are the most frequently changed [6], [7], [9].

Listing 1 Example of the deleted "&&" to splitting condition

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

Listing 2 Example of the replaced "&&" with "||" to simplifing condition

```
Source 1
if (!(nonEmpty && value.isEmpty()))
-------------------------------------------------
Source n
if (!nonEmpty || !value.isEmpty())
```

Listing 3 Example of the replaced "==" with "`isEmpty()`" to reducing the constructor call

```
Source 1
if (target.icon() == QPlaceIcon() && src.icon()
    != QPlaceIcon())
-------------------------------------------------
Source n
if (target.icon().isEmpty() && !src.icon().
    isEmpty())
```

3. Motivating Example: Conditional Statements Fixed through Peer Code Review

This section introduces `if`-statements fixed through code review. We investigate the changes between $Patch_1$ and $Patch_n$ in Figure 1.

This research targets the Qt and OpenStack projects. Qt is a cross-platform application framework, and OpenStack is a software platform for cloud computing, respectively. Table 1 shows the numbers of reviews, the numbers of `if`-statements changed in submitted patches ($Patch_1$), and the numbers of `if`-statements fixed through code review. We sample 380 patches from the original Qt review dataset and then manually read them to identify typical fixing patterns. The sample size was used to obtain a proportion of patterns within the 5% bounds of the proportion with a 95% confidential level.

Listings 1 through 3 show concrete examples of `if` fixing patterns between $Patch_1$ and $Patch_n$ obtained from the Qt project. In Listing 1, a logical AND operator ("&&") is deleted by splitting the condition into two `if`-statements [†]. In

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

Table 1 Project summary

project name	language	Time period	# Reviews	# Submitted "If" (statement / reviews)	# Fixed "If" (statements / reviews)
Qt	C++	2011-2013	69,325	5,778 / 2,120	3,343 / 1,203
OpenStack	Python	2011-2014	60,197	7,725 / 3,495	3,544 / 2,000

Listing 2, a pair of a logical NOT operator ("!") and a logical AND operator is replaced with a logical OR operator ("||") and an additional NOT operator [†]. In Listing 3, equality operators ("!=" and "==") are replaced with function calls [††].

This manual reading shows that code review often changes operators in conditional expressions of `if`-statements. Based on this observation, we characterize fixing patterns of `if`-statements using the numbers of symbols changed through code review. Section 4 details a preliminary study to analyze the changes in $Patch_1$, and Section 5 finds fixing patterns between $Patch_1$ and $Patch_n$.

4. Preliminary Study : Source Code Changes in the Review Request

This section counts symbols changed in $Patch_1$ to understand symbols frequently used in `if`-statements.

4.1 Approach

This study collects changes of `if`-statements in $Patch_1$ obtained from the review management system. Each patch is represented by a unified diff format. Although all versions of source code are available in the source code repositories of the projects, we do not use the original source code in order to shorten the time to collect analysis data. We analyze `if`-statements whose conditional expression is written in a changed line. We excluded conditional expressions across multiple lines from analysis; this filtering is not a strong limitation since multi-line `if`-statements are included in only 425 (0.6%) patches of the Qt project and 553 patches (0.9%) of the OpenStack project. Our dataset includes 5,778 (Qt) and 7,725 (OpenStack) `if`-statements changed in the submitted patches. Also, each reviews have 4 (Median) changes in Qt project and 4 (Median) changes in OpenStack Project between $Patch_1$ and $Patch_n$.

We count the number of each symbol in the condition of each `if`-statement changed in $Patch_1$. We employ ANTLR [24] with its official grammars for C++ and Python [†††] to recognize reserved words and symbols used in conditional expressions. Note that the analyzed `if`-statements include `else-if`-statements(C++) and `elif`-statements(Python) in addition to regular `if`-statements. Figure 2 describes the process of data extraction.

We collect 176 reserved words and symbols in Qt and

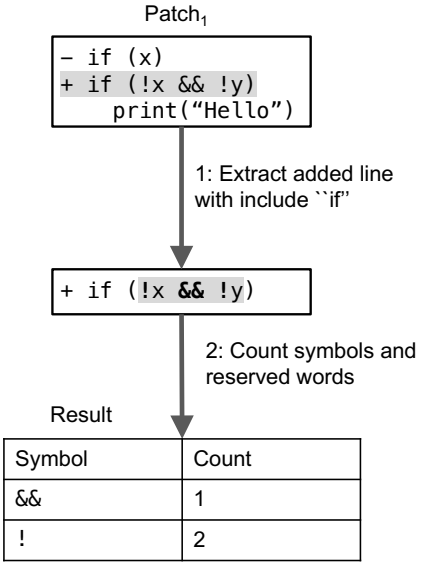


Fig. 2 Approach to extract changed symbols in `if`-statement from $Patch_1$

124 reserved words and symbols in OpenStack. Table 2 shows the frequent program elements in patches. The column "Name" indicates the name of each element used in the rest of this paper. In this analysis, we excluded identifiers to simplify the result; various identifiers are used in conditional expressions.

To analyze the frequency in the co-occurrence of symbols, we apply closed frequent itemset mining. The mining provides a **Support** metric for a set of items. The metric represents the relative frequency with respect to the total number of transactions, i.e. `if`-statements changed in the patches.

We employ the `arules` package [25] as an implementation of the mining algorithm. We extract item sets whose size is at most five elements and whose Support score is equal to or greater than 0.01, since the mining results in a huge number of item sets. If an item set is a superset of another item set and has the same Support score, then the algorithm uses only the larger one. For example, a set `{"LeftParen", "RightParen"}` is extracted and its subset `{"LeftParen"}` is filtered out, if the parentheses always appear as pairs.

4.2 Result

Figure 3 and Figure 4 show the rate of symbols that appeared in more than 5% of all `if`-statements of $Patch_1$. We extract 477 (Qt) and 162 (OpenStack) item sets obtained by frequent itemset mining whose Support score is greater

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

[††] https://codereview.Qt-project.org/#/c/6041/1..6/src/plugins/geoservices/nokia/places/qplacesupplier\srepository.cpp

[†††] https://github.com/antlr/grammars-v4

Table 2 Frequency appeared Symbols list.

Name	Symbol∗		Description	Example
	Qt (C++)	OpenStack (Python)		
Equal	==	==	compare same or not	if(a == b)
NotEqual	!=	!=	compare not same or same	if(a != b)
Not	!	not	inverse logical result	if(!a)
And	&&	and	and condition	if(a && b)
Or	||	or	or condition	if(a || b)
LeftParen	(	(	surround condition or call function	if((a || b) & c())
RightParen	)	)	surround condition or call function	if((a || b) & c())
FunctionBrace∗∗	func()	func()	call function	if(func(a, b))
Bracket∗∗	[]	[]	reference list items and dicitionaries (Python)	if(a[0])
Dot	.	.	reference individual members of classes	if(a.b())
Comma	,	,	separate expressions	if(func(a, b))
Less	<	<	compare(greater than)	if(a < b)
Greater	>	>	compare(less than)	if(a > b)
Arrow	->	(N/A)	call member from pointer	if(a->b())
Doublecolon	::	(N/A)	reference individual members of classes	if(a::b())
In	(N/A)	in	identified one variable has another variable	if a in b:
Is	(N/A)	is	compare objects are same or not	if a is b:
None	(N/A)	None	represent the absence of a value	if a is None:
Number	1,2,3...	1,2,3...	literal for number	if a > 0:
String	“String”	“String”	literal for string	if a == "a":

∗(N/A) means the symbol is unavailable in the programming language of the project.
∗∗ the symbol will be used in Section 5

than 0.01. Due to the limited space, Table 3 and Table 4 show frequent item sets whose Support score is greater than 0.10. For example, Id 1 in Table 3 shows “`LeftParen`” , “`RightParen`” which means “`(`” and “`)`” are included at the same time in the `if`-statement.

Parentheses are the most frequent symbols in the changed `if`-statements.

Qt: Figure 3 shows that parentheses (“`LeftParen`” and “`RightParen`”) are likely included in an `if`-statement. It should be noted that those numbers do not include the beginning and end parentheses for `if`-statements. While the parentheses often control the order of evaluation in a condition, the parentheses also call functions representing some conditions. An example of such function call is shown in Listing 4 [†].

OpenStack: Figure 4 shows that parentheses also frequently appear in OpenStack. However, the frequency of parentheses is lower than Qt project, because some functions in C++ are defined as reserved words in Python standard library. For example, “`std::find`” function in C++ are semantically similar to “`in`” reserved words in Python. Also, a patch author might define functions instead of “`is`” to compare objects.

“`Dot`” and “`Not`” are frequently used in the `if`-statements.

“`Dot`” is used for accessing a member of an object, and it is frequently used in both projects. “`Not`” is used to inverse logical result. In Table 3 and Table 4, approximately 30% `if`-statements used “`Not`” in two projects. The “`Not`” reserved word or symbol is often used to detect the fail of a function execution as in Listing 4 or to use the output of a function as in Listing 5[††].

Table 3 Qt:Frequency of changed symbol sets with `if` changes.

Id	Symbols	Support * 100
1	LeftParen, RightParen	54.9
2	Dot	34.3
3	Not	30.4
4	LeftParen, RightParen, Dot	30.1
5	Arrow	23.1
6	Equal	20.8
7	LeftParen, RightParen, Not	18.6
8	LeftParen, RightParen, Arrow	18.1
9	Number	14.8
10	Doublecolon	13.6
11	Not, Dot	11.8

Qt: In Table 3, the Support score of “`Dot`” with “Parentheses” are 30.1% (Id 4) of all’s “`Dot`” 34.3 (Id 2). Qt project could have used “`Dot`” to call other object’s function.

OpenStack: While “`Dot`” is the most frequently changed symbol with `if`-statement changes, nearly half of `if`-statements used “`Dot`” symbols (Id 1 in Table 4).

5. Analysis: Source Code Fixes after Review

This section extracts symbol changes as a fixing pattern between $Patch_1$ and $Patch_n$ by using association rule mining.

5.1 Approach

This analysis describes how a patch author fixed `if`-statements through the review. In this section, we identify changed `if`-statements using diffs’ changed chunks between

[†]`https://codereview.Qt-project.org/#/c/1368/1/src/plugins/Qt4projectmanager/Qt-s60/symbianQtversion.cpp`

[††]`https://codereview.Qt-project.org/#/c/2481/1/src/declarative/items/qsggridview.cpp`

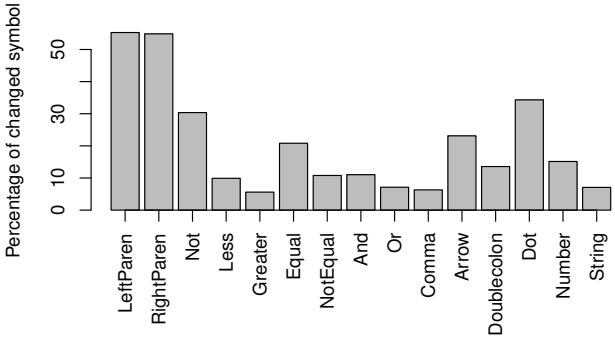


Fig. 3 Symbol change frequency with `if`-statement changes(Qt:C++).

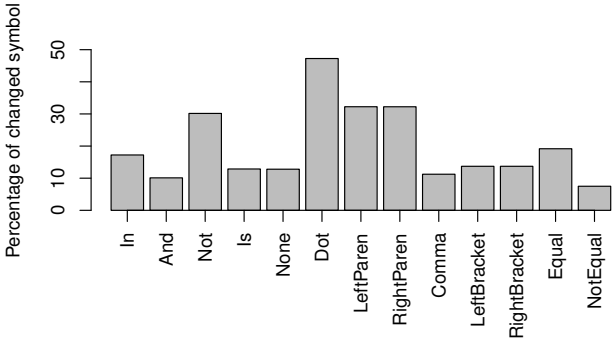


Fig. 4 Symbol change frequency with `if`-statement changes(OpenStack:Python)

Listing 4 Example of the added "(" and ")" to the function call

```
Source 1
if (!QFileInfo(systemRoot() + "/epoc32/release/
    udeb/epoc.exe").exists())
    return false;
```

Listing 5 Example of the added "!" to detecting the fail of function execution

```
Source 1
if (!isComponentComplete() || !d->model || !d->
    model->isValid())
    return;
```

source code that has been fixed by $Patch_1$ and $Patch_n$. Figure 5 shows an approach to extract fixed symbols.

1. Get the added single line with includes the `if`-statement
2. Count the number of symbols in 3,343 `if`-statements for Qt and 3544 `if` for OpenStack statements that the patch author fixed from $Patch_1$ to $Patch_n$. For example, Figure 5 shows one "&&" and two "!" changes in $Patch_1$. After reviewing $Patch_1$ to $Patch_n$, the patch author added "||", "(" and ")" in $Patch_n$, and deleted "&&" and "!".
3. Compare the difference in the number of symbols between $Patch_1$ and $Patch_n$ such as "*And_deleted*" (*And* symbol(s) is deleted between $Patch_1$ and $Patch_n$), "*Or_added*" (*Or* symbol(s) is added between $Patch_1$ and $Patch_n$), "*LeftParen_added*", "*RightParen_added*" and "*Not_deleted*" to understand changed contents.

Table 4 OpenStack:Frequency of changed symbol sets with `if` changes.

Id	Symbols	Support * 100
1	Dot	47.2
2	LeftParen	32.2
3	LeftParen, RightParen	32.2
4	Not	30.2
5	Dot, LeftParen	23.1
6	Dot, LeftParen, RightParen	23.1
7	Equal	19.2
8	In	17.2
9	Not, Dot	14.4
10	LeftBracket, RightBracket	13.7
11	Is	12.9
12	None	12.8
13	Is, None	11.8
14	Comma	11.2
15	Not,LeftParen, RightParen	10.8

Using this dataset, we conducted an empirical study to understand the fixed symbols through code review using an association rule mining technique that is a popular method for the generation of usage rules [26], [27].

Association rule mining is a method to extract a relationship between two or more items as an association rule from a combination of a large number of items. The precondition and post-condition are called **LHS** (Left-Hand-Side) and **RHS** (Right-Hand-Side).

We discover two kinds of rules with both changed symbols (e.g., step 2 in Figure 5) and fixed symbols (e.g., step 3 in Figure 5) by the association rule mining.

1. Replaced symbols pairs
   (e.g. "*And_deleted*" $\Rightarrow$ "*Or_added*")
2. Added symbols pairs
   (e.g. "*And*" $\Rightarrow$ "*Or_added*"))

We measure three evaluation scores from the association rule mining. They are **Support**, **Confidence** and **Lift**. **Support** of a rule is its relative frequency with respect to the total number of transactions in the history. **Confidence** is its relative frequency of the rule with respect to the number of historical transactions containing the antecedent LHS.

$$Confidence(\{LHS\} \Rightarrow \{RHS\}) = \frac{Support(\{LHS\} \Rightarrow \{RHS\})}{Support(LHS)} \quad (1)$$

Lift measures the magnification of the data where precondition LHS and post-condition RHS exist in rules with post-condition RHS.

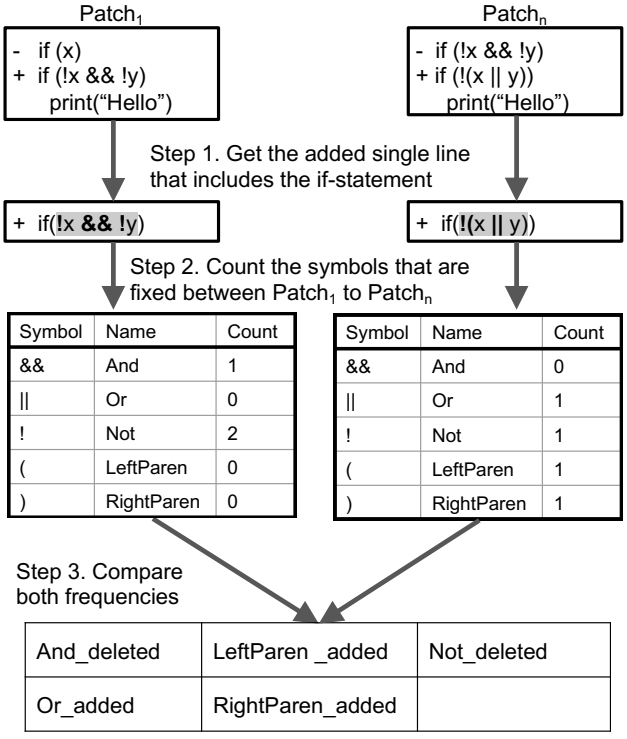


Fig. 5 Approach to extract fixed symbols after reviewing.

$$Lift(\{LHS\} \Rightarrow \{RHS\}) = \frac{Confidence(\{LHS\} \Rightarrow \{RHS\})}{Support(RHS)} \quad (2)$$

For association rule mining, we use the `arules` package again. Since the association rule mining outputs many rules, we extract item sets with less than 6 items as well as the extracted item sets whose support score is greater than 0.01, confidence score is greater than 0.1, and Lift score is greater than 1.0.

In our preliminary study, we found that more than 99% of "`LeftParen`" and "`RightParen`" pairs represent function calls. Similarly, "`LeftBracket`" and "`RightBracket`" pairs usually represent array access. Hence, we regard those pairs as "`FunctionBrace`" and "`Bracket`" in this analysis.

5.2 Result

Figures 6 and 7 show the rate of fixed symbols that appeared is more than 5% of all `if`-statement fixes. Table 5 and Table 6 show the extracted 7 rules and 31 rules for the fixing patterns from Qt and OpenStack by association rule mining.

Observation 1: `if`-statement fixes frequently add or delete `FunctionBrace` through code review.

Qt: 35% of `if`-statement are added (23%) or deleted (12%) `FunctionBrace` in Figure 6. In our manual reading, we found that there are examples for excepting redundant function calls (Listings 6[†]) or including necessary function calls (Listings 7[††]).

OpenStack: From Figure 7, "`FunctionBrace`" frequently deleted or added too. 40% of `if`-statements are added (20%) and deleted (20%)in `FunctionBrace` in Figure 7. As we showed in our preliminary study, these findings are according to the python language in OpenStack such as the reserved words ("`in`" and "`is`") instead of the functions ("`contain`" and "`equal`").

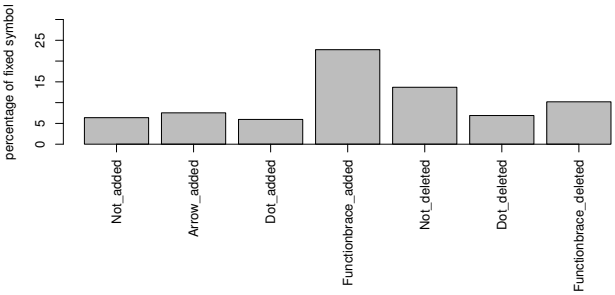


Fig. 6 The added or deleted rate of symbols with `if`-statement fixes (Qt:C++).

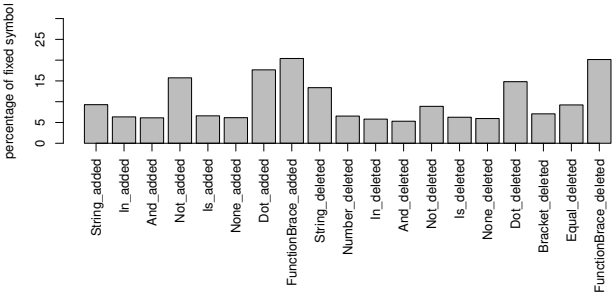


Fig. 7 The added or deleted rate of symbols with `if`-statement fixes (OpenStack:Python).

Listing 6 Example of the deleted "FunctionBrace" to excepting redundant function call

```
Source 1
if (config->QtVersion() && QtSupport::
    QmlObserverTool::canBuild(config->QtVersion()
    ))
-------------------------------------------------
Source n
if (QtSupport::QmlObserverTool::canBuild(config->
    QtVersion()))
```

Observation 2: Patch authors are likely to replace "`Not`" with "`FunctionBrace`" through code review in Qt project. Figure 6 shows the rate of "Not" is likely to be deleted (13% for "`Not_deleted`") more than added (6% for "`Not_added`") through code review. From Id 4 in Table 5, we found that "`Not`" is likely to be deleted to use

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

[††]`https://codereview.qt-project.org/#/c/1843/1..2/src/plugins/qt4projectmanager/qt-desktop/simulatorqt\version.cpp`

Table 5 Frequency of changed symbol sets with 3,343 `if` fixes (sort by Lift) (Qt:C++)

Id	LHS	RHS	Support * 100	Confidence * 100	Lift
1	FunctionBrace, Doublecolon_deleted	Arrow_added	1.0	30.4	4.03
2	Number_deleted	Not_added	1.1	24.8	3.89
3	Doublecolon_deleted	Arrow_added	1.0	24.8	3.29
4	Not_deleted	FunctionBrace_added	10.0	72.8	3.20
5	Doublecolon, FunctionBrace	Arrow_added	1.2	13.9	1.84
6	FunctionBrace_deleted	Not_added	1.2	11.7	1.84
7	Not	FunctionBrace_added	11.6	36.2	1.59

Table 6 Frequency of changed symbol sets with 3,544 `if` fixes (sort by Lift) (OpenStack:Python).

Id	LHS	RHS	Support * 100	Confidence * 100	Lift
1	Is_added, Number_deleted	None_added	1.0	94.9	15.42
2	None_added, Number_deleted	Is_added	1.0	100.0	15.15
3	Is_added, FunctionBrace_deleted	None_added	1.1	90.5	14.71
4	FunctionBrace, FunctionBrace_added, Bracket_deleted	Dot_added	1.2	97.8	5.54
5	FunctionBrace_added, Bracket_deleted	Dot_added	1.3	95.9	5.43
6	FunctionBrace_added, String_deleted	Dot_added	1.9	89.5	5.07
7	Dot_added, In_deleted	FunctionBrace_added	1.7	92.2	4.52
8	String, FunctionBrace_added, In_deleted	Dot_added	1.3	79.3	4.49
9	String, Equal_deleted	In_added	1.2	20.4	3.21
10	String, FunctionBrace, Bracket_deleted	Dot_added	2.1	40.3	2.28
11	String, Bracket_deleted	Dot_added	2.2	39.8	2.25
12	String, FunctionBrace_deleted	In_added	1.2	13.9	2.19
13	Equal_deleted, FunctionBrace_deleted	Not_added	1.4	33.8	2.15
14	None, Is_deleted	Not_added	1.8	33.0	2.09
15	Is_deleted, None_deleted	Not_added	1.7	32.8	2.08
16	Is, None_deleted	Not_added	1.7	32.6	2.07
17	String_deleted, Bracket_deleted	Dot_added	1.6	36.4	2.06
18	Is_deleted	Not_added	2.0	32.0	2.03
19	None_deleted	Not_added	1.9	31.8	2.02
20	Number_deleted	Not_added	2.1	31.5	2.00
21	FunctionBrace, Bracket_deleted	Dot_added	2.3	35.0	1.98
22	String, In_deleted	Dot_added	1.4	34.5	1.95
23	Bracket, String_deleted	Dot_added	1.7	34.3	1.94
24	Bracket_deleted	Dot_added	2.4	34.3	1.94
25	String, In_deleted	FunctionBrace_added	1.6	39.2	1.92
26	FunctionBrace, Equal_deleted	Not_added	2.1	29.0	1.84
27	In_deleted	FunctionBrace_added	2.2	37.4	1.83
28	Equal_deleted	Not_added	2.6	28.4	1.81
29	In_deleted	Dot_added	1.8	31.1	1.76
30	None	Not_added	2.8	24.7	1.57
31	String, Bracket	Dot_added	3.2	26.3	1.49

Listing 7 Example of the added "`FunctionBrace`" to including function call

```
Source 1
if (qmlviewerCommand().isEmpty())
-----------------------------------------------
Source n
if (QtVersion() >= QtSupport::QtVersionNumber
    (4,7,0) && qmlviewerCommand().isEmpty())
```

Listing 8 Example of the deleted "`!`" to make clear the object's type

```
Source 1
if (!hoverItems)
-----------------------------------------------
Source n
if (hoverItems.isEmpty())
```

"`FunctionBrace`" as in Listing 8[†]. In this example, using the function instead of "`Not`" made it easier for the developer to understand object's type such as the array. Also, from Id 6 in Table 5, "`FunctionBrace`" is less likely to be deleted to use "`Not`". Hence, we found that the Qt project often used "`FunctionBrace`" instead of "`Not`" as one of the project specific rules.

Observation 3: Patch authors are likely to delete "`String`" through code review in the OpenStack project. When using python, some patch authors often use "`String`" and "`Bracket`" to identify the object's dictionary status. However, we found in Id 5, 6 in Table 6, "`String`" or "`Bracket`" should be replaced with "`Dot`" to call the function in OpenStack as in Listing 9[†] or Listing 10[††]. In

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

[†]`https://review.openstack.org/#/c/60425/1..3/cafe/engine/http/behaviors.py`

[††]`https://review.openstack.org/#/c/15708/1..5/glance/`

Listing 9, the patch author improved readability by using "`bytes_.startswith()`" and "`.endswith()`", especially expecting "`String`". In Listing 10, the patch author deleted "`String`" to reduce access to "`values`". Replacing "`Dot`" with "`Bracket`" not only improves maintainability but avoids the error. Hence, our approach can extract rules for not only maintenance but can contribute to improving performance.

Listing 9 Example of the replacing "`Bracket`" with "`FunctionBrace`" to avoiding index error

```
Source 1
if bytes_[0] != '-' and bytes_[-1] != '-':
-----------------------------------------------
Source n
if bytes_.startswith('-') or bytes_.endswith('-')
    :
```

Listing 10 Example of the deleting "`String`" to reduce the value access

```
Source 1
if 'size' in values and values['size']:
-----------------------------------------------
Source n
if values.get('size') is not None:
```

5.3 Summary

In summary, we found valuable code fixing patterns that can be an additional coding rule to reduce the costs of a reviewer. Also, we compared our fixing rules to each projects' coding rules(Qt [†††] and OpenStack [††††]). Our fixing rules are not included in these rules. Hence, the fixing rules that we identified are valuable for patch authors to reduce redundant fix. We can recommend to reviewers the symbols that are likely to be fixed in the submitted patch ($Patch_1$).

- *The Qt (C++) project is likely to replace "`Not`" with function calling through code review.*
  "`Not`" is one of the most frequently used symbols in `if`-statement change submits; however, for readability, the Qt project uses a function call instead of "`Not`". Our study found such project-specific rules extracted from review data are unlike previous studies from integrated data [28]. Furthermore, our approach found the fixing patterns that do not have an effect on source code behavior.
- *The OpenStack (Python) project replaces reserved words with "`Number`" and "`String`" through code review*
  Python has reserved words such as "`in`" and "`not`". These reserved words instead of used "`0`" and the dictionary to improve readability and to avoid the index errors.

The concept of these project-specific rules are similar for improving readability and maintainability. However, the concrete rules are difference between programming languages or projects. Our approach extracted the project-specific concrete rules that are not supported by current coding check tools like pylint [†††††].

6. Threats to Validity

6.1 Internal validity

A factor that potentially affects the internal validity of our study is that we extracted symbol changes in `if`-statement by syntactic analysis to extract fixed code patterns through code review. The changes are analyzed based on the patch files and the original source code to identify spots with `if`-statements. However, to collect the original source code, we focused on patch files and we collected `if`-statements on each single line change. This methodology has no significant impact on our results since the number of `if`-statement changes across multiple lines is merely 425 patches of our target 69,325 patches (0.6%) and 553 patches out of OpenStack project's 60,197 patches (0.9%).

We collected not only `if`-condition statements, but also `switch`, and `for` and `while` condition statements. `Switch`, `for` and `while` condition statements do not appear more frequently than the `if`-statements in the code review as with integrated code changes [6].

Also, we compared $Patch_1$ and $Patch_n$ to detect source code changes in code review. Although the number of changed times (size of n) may be related to change contents, the analysis is out of scope of this paper.

6.2 External validity

The project-specific nature of our dataset has many limits. This research conducted an empirical study using only the Qt and OpenStack 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 "<=". Despite such variability, we contend that our approach should provide a framework for understanding individual rules or trend fixes in each project.

7. Related Work

7.1 Code Review

Many researchers have conducted empirical studies to understand code review [3], [4], [11]–[15], [29], [30]. Unlike our focus most published code review studies focus on the review

db/sqlalchemy/api.py

[†††] https://wiki.qt.io/Coding_Conventions

[††††] https://docs.openstack.org/hacking/latest/user/hacking.html

[†††††] https://www.pylint.org/

process or reviewers' communication. For example, Tsay et al. found those patch authors and reviewers often discuss and propose solutions with each other to fix patches [29]. In particular, Czerwonka et al. [30] found that 15% of the discussions for patch fixes are about functional issues while Mäntylä et al. [3] and Beller et al. [4] found that 75% of discussions for patch fixes are about software maintenance and 15% are about functional issues. These studies help us understand which issues should be solved in the code review process and our work focuses on how those issues are fixed. Towards this goal, we focused on source code changes through code review, specifically those involving `if` changes.

7.2 Coding Conventions

Code convention issues also relate to our study because some code reviews are refactoring based on coding convention [5][31][32]. Smit et al. [32] found that `CheckStyle` is useful for detecting whether or not source codes follow its coding rules. Also, some convention tools such as Pylint released by Thenault check the format of coding conventions. In addition, Allamanis et al. [33] have developed a tool to fix code conventions. However, to the best of our knowledge, little is known about how a patch author fixes conditional statement issues based on reviewers feedback.

8. Conclusion

This research conducted an empirical study on how `if`-statements are fixed based on reviewers feedback.

The results of our case study on the Qt and OpenStack project showed that in each specific fixing pattern that approximately 35% of the code is likely to be added or deleted parentheses through code review. The contribution of this study is the discovery of frequent patterns for fixing `if`-statements through code review. We think this, in turn, may help to design an issue detection approach. Also, we created a coding convention checker that detects project-specific rules. If a patch author detects the possibility of changing these symbols before the code review request, the reviewers might be able to spend more time on other additional review requests and thus same time and costs.

In the future, we intend to propose a method to review and automatically fix a symbol in `if`-statements, such as a change impact analysis that can be conducted based on code review data with a history of integrated code changes.

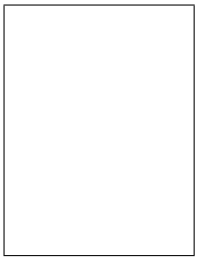
Acknowledgments

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. For proofreading our paper, thanks to Prof. Hiromi Teramoto.

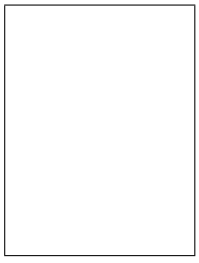
References

[1] P.C. Rigby and M.A. Storey, "Understanding broadcast based peer review on open source software projects," Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp.541–550, 2011.
[2] D.S. Alberts, "The economics of software quality assurance," Proceedings of the National Computer Conference and Exposition, pp.433–442, 1976.
[3] M.V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?," Proceedings of the IEEE Transactions on Software Engineering (TSE'09), vol.35, no.3, pp.430–448, 2009.
[4] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?," Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14), pp.202–211, 2014.
[5] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14), pp.271–280, 2014.
[6] K. Pan, S. Kim, and E.J. Whitehead, "Toward an understanding of bug fix patterns," Empirical Software Engineering (ESE'09), vol.14, no.3, pp.286–315, 2009.
[7] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with AST analysis," Proceedings of the IEEE International Conference on Software Maintenance (ICSM'13), pp.388–391, 2013.
[8] D. Spinellis, Code quality: the open source perspective, 2006.
[9] S.H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury, *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," Proceedings of the 39th International Conference on Software Engineering Companion (ICSE'17), pp.180–182, 2017.
[10] 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 -," Proceedings of the 8th IEEE International Workshop on Program Debugging (IWPD'17), 2017.
[11] 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," Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14), pp.192–201, 2014.
[12] 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," Proceedings of the 6th International Workshop on Social Software Engineering (IWSSE'14), pp.37–44, 2014.
[13] P. Thongtanunam, S. McIntosh, A.E. Hassan, and H. Iida, "Investigating code review practices in defective files: An empirical study of the qt system," Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15), pp.168–179, 2015.
[14] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), pp.171–180, 2015.
[15] 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.
[16] E.S. Raymond, "The cathedral and the bazaar," Knowledge, Technology & Policy, vol.12, no.3, pp.23–49, 1999.
[17] P.C. Rigby and C. Bird, "Convergent contemporary software peer review practices," Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13), pp.202–212, 2013.
[18] 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," Proceedings of the 12th International Conference on Open Source Systems (OSS'16), pp.97–110, 2016.
[19] 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 mod-

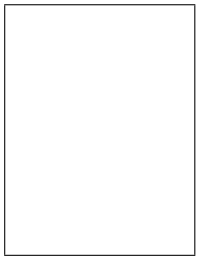
ern code review," Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), pp.141–150, 2015.
[20] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," Proceedings of the 35th International Conference on Software Engineering (ICSE'13), pp.931–940, 2013.
[21] M. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review.," Transactions on Software Engineering (TSE'15), vol.42, no.6, pp.530–543, 2015.
[22] M.M. Rahman, C.K. Roy, and J.A. Collins, "Correct: Code reviewer recommendation in github based on cross-project and technology experience," Proceedings of the 38th International Conference on Software Engineering (ICSE'16), pp.222–231, 2016.
[23] 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.," Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME'15), pp.261–270, 2015.
[24] T. Parr, The definitive ANTLR 4 reference, Pragmatic Bookshelf, 2013.
[25] M. Hahsler, S. Chelluboina, K. Hornik, and C. Buchta, "The arules r-package ecosystem: analyzing interesting patterns from large transaction data sets," Journal of Machine Learning Research (JMLR), vol.12, no.Jun, pp.2021–2025, 2011.
[26] C. Zhang and S. Zhang, Association rule mining: models and algorithms, Springer-Verlag, 2002.
[27] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," Proceedings of the International Conference on Management of Data (SIGMOD'00), pp.1–12, 2000.
[28] T. Rolfsnes, L. Moonen, and D.a. Binkley, "Predicting relevance of change recommendations," Proceedings of the International Conference on Automated Software Engineering (ASE'17), 2017.
[29] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: Evaluating contributions through discussion in github.," Proceedings of the 22nd International Sym posium on Foundations of Software Engineering (SANER'15), pp.144–154, 2014.
[30] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs: How the current code review best practice slows us down," Proceedings of the 37th International Conference on Software Engineering (ICSE'15), pp.27–28, 2015.
[31] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," IEEE International Conference on Software Maintenance (ICSM'08), pp.277–286, 2008.
[32] M. Smit, B. Gergel, H.J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11), pp.504–507, 2011.
[33] M. Allamanis, E.T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14), pp.281–293, 2014.



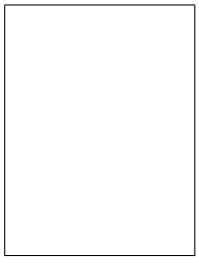
Yuki Ueda Yuki Ueda received the B.E. degree in Department of Mathematics and Computer Science, Interdisciplinary Faculty of Science and Engineering, Shimane University, Japan. He is currently working toward the M.E. degree in Nara Institute of Science and Technology in Japan. His research interests include program analysis and code review.



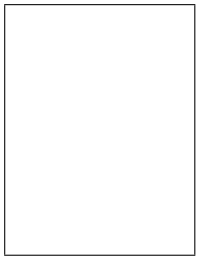
Akinori Ihara Akinori Ihara received the B.E. degree in Science and Technology from Ryukoku University,Javan in 2007, and the ME degree (2009) and DE degree (2012) in information science from Nara Institute of Science and Technology, Japan. He is currently Assistant Professor at Nara Institute of Science and Technology. His research interests include the quantitative evaluation of open source software development process. He is a member of the IEEE and IPSJ



Takashi Ishio Takashi Ishio received the Ph.D. degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006-2007. He was an assistant professor at Osaka University from 2007-2017. He is now an associate professor of Nara Institute of Science and Technology. His research interests include program analysis, program comprehension, and software reuse. He is a member of the IEEE, ACM, IPSJ and JSSST.



Toshiki Hirao Toshiki Hirao is currently working in the Ph.D. Program in Information Science at Nara Institute of Science and Technology in Japan. He has been a fellowship researcher of the JSPS (DC1) since April 2017. His research interests include empirical software engineering and mining software repository. He received the B.E. degree (2015) in department of education from Osaka Kyoiku University, and the M.E. degree (2017) in Information Science from Nara Institute of Science and Technology.



Kenichi Matsumoto Kenichi Matsumoto received the B.E., M.E., and Ph.D. degrees in Engineering from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute Science and Technology, Japan. His research interests include software measurement and software process. He is a senior member of the IEEE and a member of the IPSJ and SPM.