Mining Source Code Improvement Patterns from Similar Code Review Works

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

***Abstract*—Code review is key to ensuring the absence of potential issues in source code. Code reviewers spend a large amount of time to manually check submitted patches based on their knowledge. Since a number of patches sometimes have similar potential issues, code reviewers need to suggest similar source code changes to patch authors. If patch authors notice similar code improvement patterns by themselves before submitting to code review, reviewers' cost would be reduced. In order to detect similar code changes patterns, this study employs a sequential pattern mining to detect source code improvement patterns that frequently appear in code review history. In a case study using a code review dataset of the OpenStack project, we found that the detected patterns by our proposed approach included effective examples to improve patches without reviewers' manual check. We also found that the patterns have been changed in time series; our pattern mining approach timely achieves to track the effective code improvement patterns.**

***Index Terms*—code review, source code changes, sequential pattern mining**

I. Introduction

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

To help the developers use a common implementation style, some software projects provide an own coding guideline. The coding guideline usually includes general conventions for programming languages such as PEP8 [2], CERT C, and MISRA C. Also, current coding style checkers, such as like Pylint [3] is used to detect common implementation style issues. However, there are some undefined project-specific implementation conventions in each project, it is difficult for novice developers to use them. And, the conventions would be changed in time series. Reviewers often spend much time verifying the proposed code changes through code review manually [4], [5]. This works help for achieve well maintainable software for the future development.

In our previous study, we found that reviewers needed to send the same feedback several times to solve the same type of issues that are not defined in project/language coding guideline [6]. The time that solving the similar type of issues can be reduced if patch authors fix these issues by themselves before patch submissions.

To reduce the code review cost, we propose an approach to extract source code improvement patterns from existing code review history. Since various code improvements have been recorded in the code change history. We detect similar improvements in submitted patches using a sequential pattern mining algorithm that is a kind of Type-3 code clone detection technique. Using this approach, patch authors can apply the same improvement to their code in order to improve their code quality.

The previous research proposed an approach to detect famous refactoring pattern automatically [7]. In our research, we timely detect code improvement patterns even that patterns are only used in one target project. It can clear to the most important pattern in the project.

As a case study, we analyze 228,099 submitted patches in OpenStack. To evaluate source code improvement patterns that are detected from the code change dataset, we define two research questions.

RQ1: Which code improvement patterns are frequently appear?

In our target dataset, our approach detects 1,476 improvement patterns. Especially, we find 8 frequently appeared code improvement patterns that have appeared more than 300 times and more than 0.10 accuracy to remove redundant patterns and choose threshold from similar pattern paper. Also, we classify the 8 patterns into 3 categories (i.e. OpenStack-specific problems, Python language-specific problem, and Readability problem). Despite these 8 patterns are not included in OpenStack and Python language coding guideline, we found patterns' related discussion on StackOverflow and OpenStack document. Using our approach, novice patch author can understand implicit projects' conventions.

RQ2: Which type of code improvement patterns that increase/decrease over time?

Coding style checkers are working based on fixed sets of rules, improvement pattern might be changing by project policy or environment changes. For frequency of code improvement patterns that solving python update problems, that are changing in time series. Using our approach, reviewers can timely track code improvement patterns, and easily solve similar potential issues.

The rest of the paper is organized as follows. Section II introduces the code review process. Section III describes our approach in answering the two research questions. Section IV

presents the evaluation results of research questions. Section V establishes the validity of our empirical study. Section VI introduces related works. Section VII summaries this paper and describes our future work.

II. Code Review

There are various tools for managing peer code review processes. For example, Gerrit [1] and ReviewBoard [2] are commonly used in many software projects to receive lightweight reviews. Technically, these code review tools are used for patch submission triggers, automatic tests, and manual reviewing. It helps to decide whether or not the submitted patch should be integrated into a version control system.

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

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

In previous research, most of the reviewers are considering that code improvement is the most important in code review [8]. **Code improvement** is defined as terms of readability, commenting, consistency, dead code removal and so on. However, it does not involve correctness or defects. In this study, we define code improvement patterns from code change difference between Initial Patch and Integrated Patch.

III. Pattern Mining Approach

This section describes the approach of sequential pattern mining to detect code improvement patterns. We detect code improvement patterns through code review using a sequential pattern mining technique. Sequential pattern mining is a well-known method for the finding of relevant patterns from two or more item sequences [9]. Sequential pattern mining extracts frequent subsequences from a sequence dataset. A sequence in a sequence dataset is an ordered list of elements [10].

For example, if we has two sequence (`abc`) and `ace` that have three elements, we can detect pattern (`a` `c`) that appeared two times. We detect code improvement pattern from source code tokens as the sequence. A previous mining approach focused on partial order of API calling on source code [11].

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

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

This approach focuses on source code changes through code review to detect source code improvement patterns that independents from API. We extract code improvement from code review systems, then extract frequent changes as patterns. Also, we use Prefixspan algorithm [12] that is one of the most efficiency sequential pattern mining algorithm. Even redundant pattern can be found in the patterns, this approach ignores them by appeared frequency and patterns' confidence.

A. Pattern Minning Process

In this research, we collect code improvement patterns as the changed token sequence. We target source code token differences sequence between initial and integrated patches. Figure 1 shows the overview of the detecting sequence approach in this study.

1. We extract pairs of code chunks (consecutive lines of code) from the initial patch and integrated patch pairs. The pairs are simply recognized by applying `diff` between the patches.
2. We divide the difference by token; at the same time, normalize `NUMBER` literal and `STRING` literal tokens to increase detectable pattern.
3. We convert line difference to token difference sequence as the dataset of sequential pattern mining.
4. We extract token difference sequence pattern by sequential pattern mining.
5. We count how many times do patterns appeared from other patch pairs.

We adopt improvement pattern to initial patches that have **Trigger sequence**. Trigger sequence is tokens sequence to adopt improvement pattern. If the initial patch has tokens of trigger sequence, we can suggest code change based on improvement pattern. We detect trigger sequences from patterns' deleted and unchanged tokens. In Listing 1, if we detect (`i=dic` `- [` `+ .get(`) improvement patterns, and we extract trigger sequence (`i=dic` `[`) from deleted and unchanged tokens.

B. Filtering and Detecting Confidence

When we suggest a code improvement pattern to initial patch that has tokens of trigger sequence, some patterns will suggest wrong change method. To remove redundant patterns, we use four filtering approaches.

1) We ignore patterns that do not suggest changes. So only added/deleted patterns are skipped. For example (`i=dic` `- [` `- ]`) pattern will be removed because this pattern just removed two token.

2) We ignore duplicated meaning patterns. For example, (`- [` `+ .get(`) pattern are contained in (`i=dic` `- [` `+ .get(`) patterns. We then prioritize larger size pattern if there are the changes in the same line.

3) We filter less than 10% confidence patterns. We calculate patterns' confidence by below:

$$Confidence = \frac{|ActuallyChangedIntegratedPatches|}{|TriggerableInitialPatches|} \quad (1)$$

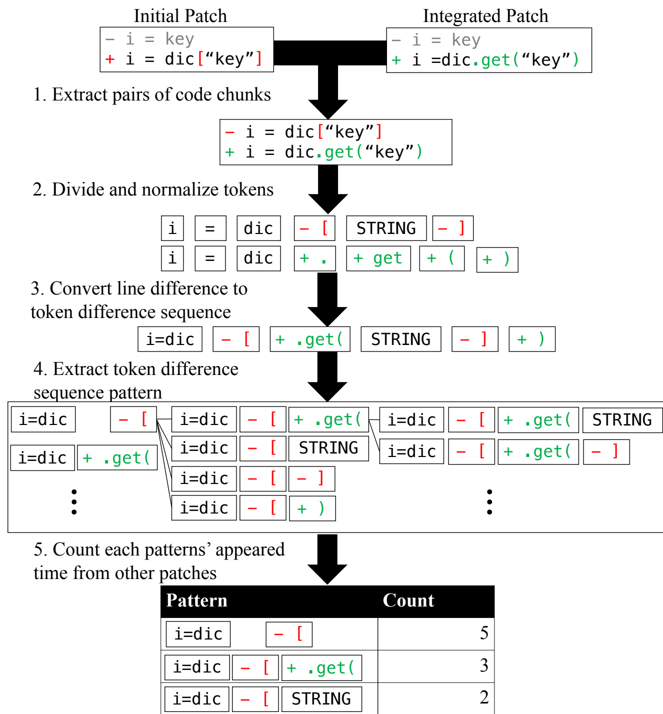

Fig. 1. Approach Overview of the Detecting Token Difference Sequence

Listing 1
EXAMPLE OF THE TRIGGERABLE INITIAL PATCH AND INTEGRATED PATCH PATTERN CASE OF PATTERN (I=DIC - [+ .GET( )

```
# Triggerable initial patch
1 i = dic['key']
# Actually changed integrated patch
1 i = dic.get('key')
# NOT actually changed integrated patch
1 i = dic['key2']
```

Where *Triggerable Initial Patches* are initial patches that have trigger sequence. Where *Actually Changed Integrated Patches* are integrated patches that initial patch is triggerable and that changed based on improvement pattern. In Listing 1, we can detect triggerable initial patch that has trigger sequence (i=dic [ ). If that patch is changed by integrated patch based on a pattern (i=dic - [+ .get( ), that patch is actually changed integrated patch. If the pattern has a less than 0.10 confidence, we remove the pattern to reduce evaluation time on research questions.

4) We count support value that means how many occurred patterns in the dataset. And we filter patterns that only appeared once.

IV. CASE STUDY

This section introduces the evaluation of the approach of each detected pattern. To evaluate patterns, we measure the accuracy of each pattern. In section III, we calculate pattern and confidence from same dataset. In this section, we calculate accuracy by the same approach. We detect patterns from training dataset and calculate patterns' accuracy from test dataset.

A. Case Study Design

This study targets the OpenStack project that is a software platform for cloud computing, respectively. We detect 1,476 patterns from OpenStack code review dataset. Particularly, we focus on 8 most frequently appeared code improvement patterns.

To evaluate detected code improvement patterns, we define two research questions.

RQ1: Which code improvement patterns are frequently appear?

The software project might have implicit coding conventions that are not included in projects' coding guideline. If patch author can refer code improvement patterns that frequently appears and high accuracy, patch author could fix source code before submitting to code review. In RQ1, we evaluate frequently appeared code improvement patterns. As the dataset, we divide diff dataset by two to 555,050 (Training diff datasets) and 61,673 changes (Test diff datasets). First, we detect 1,476 code improvement patterns that appeared more than one of 300 times from training diff dataset. Second, we calculate patterns' accuracy by using Section III-B's confidence approach from test diff datasets.

RQ2: Which type of code improvement patterns that increase/decrease over time?

Some patch author needs to change old convention code to the latest convention code. If the frequently appeared pattern can be changing by times, a project needs to include the latest pattern. In RQ2, to detect which patterns' frequencies transition, we divide dataset. We defined the period of the divided dataset to period 1 to period 5. Each period has 121,652 change difference from 616,723 changes. We detect patterns from period n as the training dataset and calculate patterns' accuracies by diff set on the period (n+1) as the test dataset.

B. RQ1 Result: Which code improvement patterns are frequently appear?

Table I shows most frequently appeared code improvement patterns and accuracy. We found 8 frequently appeared code improvement patterns that have appeared more than 300 times and more than 0.10 accuracy. We filtered out the other frequently patterns that are subsets of other patterns.

By our manual study, we name each pattern to discuss in this paper, and survey related document from OpenStack documents, StackOverflow and Python documents.

Almost these patterns are described on OpenStack documents or discussed on StackOverflow, and each description has a footnote of the related document. These documents are not only from OpenStack project. Even OpenStack has a coding guideline that shows general coding conventions, detected code improvement patterns are not included in OpenStack coding convention pages [10]. This result shows this approach

[10] OpenStack coding convention https://docs.openstack.org/hacking/latest/

TABLE I
THE MOST FREQUENTLY APPEARED CODE IMPROVEMENT PATTERNS FROM 555,050 CHANGES IN OPENSTACK

Pattern category	Pattern name	Description in StackOverflow or OpenStack document	Detected Pattern	Support	Accuracy

Project-specific	disk2disk_api	Changing dependency of refactoring on OpenStack [3]	`return fs_type in (`
			`- disk`
			`+ disk_api`
			...same type of fix after 7 lines
				5,754	1.00

	stubs.Set2stub_out	Use `stub_out` function instead of `self.stubs.Set` depended `mox` which is not maintained package [4]	`self .`
			`+ stub_out`
			`- stubs . Set`
			`)`
				5,696	0.32

Language-specific	assert-equals2equal	`assertEquals` is defined as dupurecated alias in Python3 [5]	`- assertEquals` `+ assertEqual`	4,015	0.99
	xrange2range	Both works are similar in Python2, and `xrange` is removed from Python3 [6]	`- xrange` `+ range`	1,159	0.86
	iteritems2items	`iteritems` method is removed from Python3 [8]	`- iteritems` `+ items`	818	0.12

Readability-improvement	reverse-assert-arguments	Left argument shows expected result, right shows actual in test case output [7]	`self.assertEqual(`
			`+ NUMBER ,`
			`- , NUMBER`
				1,149	0.53

	remove-redundant-in	Remove redundant "`in`" for readability e.g. `self.assertTrue(x in list)` to `self.assertIn(x, list)`	`self .`
			`- assertTrue`
			`+ assertIn`
			`- in`
				930	0.55

Other	directly-dictionary-access	Avoiding to get `None` value if directory key is missing [9] `STRING` can be changed other parameter such as `NUMBER`	`+ (`
			`- . get (`
			`STRING`
			`- )`
			`+ ]`
				419	0.11

[3] disk2disk_api: https://wiki.openstack.org/wiki/VirtDiskApiRefactor
[4] stubs.Set2stub_out: https://docs.openstack.org/nova/13.1.2/api/nova.test.html#nova.test.TestCase.stub_out
[5] assert-equals2equal: https://stackoverflow.com/questions/930995/assertequals-vs-assertequal-in-python
[6] xrange2range: https://stackoverflow.com/questions/15014310/why-is-there-no-xrange-function-in-python3
[7] reverse-assert-arguments: https://stackoverflow.com/questions/2404978/why-are-assertequals-parameters-in-the-order-expected-actual
[8] iteritems2items: https://stackoverflow.com/questions/10458437/what-is-the-difference-between-dict-items-and-dict-iteritems
[9] directly-dictionary-access: https://stackoverflow.com/questions/11041405/why-dict-getkey-instead-of-dictkey

can detect patterns that are implicit and useful on the real development process.

One complicated pattern "directly-dictionary-access" has low accuracy since it has a small impact on code behavior. Listings 2 is example of adopting "directly-dictionary-access" pattern in OpenStack [11]. On other hands, despite "`dict[STRING]`" to "`dict.get("STRING")`" pattern has appeared only 139 times, it avoid `KeyError` when if directory key is missing. One of the causes, some OpenStack code are already written to avoid `KeyError` likely Listings 3.

Second, "reverse-assert-arguments" pattern has no impact to source code behavior and code readability. This pattern improves output readability such as Listings 4. And this ordering is assumed for creating the failure message to the patch author.

Although, We define 3 types of patterns for improvement purposes.

1) **Project-specific Pattern**: The patterns that solve a OpenStack-specific problem; they are occurred by OpenStack dependency update (e.g. `disk2disk_api`, `stubs.Set2stub_out`)

[11] Example of directly-dictionary-access: https://review.openstack.org/#/c/174036/8..9/jenkins_jobs/modules/publishers.py

Listing 2
EXAMPLE OF THE "DIRECTLY-DICTIONARY-ACCESS" PATTERN FOR AVOIDING "KEYERROR"

```
# Initial patch
1 XML.SubElement(hipchat, 'completeJobMessage').text = str(
2    data.get('complete-message', ''))
---
# Integrated patch
1 if 'complete-message' in data:
2    XML.SubElement(hipchat, 'completeJobMessage').text =
       str(
3      data['complete-message'])
```

Listing 3
EXAMPLE OF THE UNNECESSARY REVERSE "DIRECTLY-DICTIONARY-ACCESS" PATTERN FOR AVOIDING KEYERROR IF DIRECTORY KEY IS MISSING

```
# Initial patch
1 if 'key' in data:
2    x = data['key']
---
# Integrated patch
1 x = data.get('key')
```

2) **Language-specific Pattern**: The patterns that solve difference between python2 and python3; they are occurred by Python language update (`xrange2range`, `assert-equals2equal`, `iteritems2items`)

TABLE II
CHANGING FREQUENCY AND ACCURACY OF EACH PERIOD CHANGES (EACH N = 121,652)

Pattern name	Support / Accuracy period 2 (2015-6-23 ˜ 2015-7-23)	period 3 (˜ 2015-12-3)	period 4 (˜ 2016-5-25)	period 5 (˜ 2016-11-10)
disk2disk_api	0 / —	0 / —	2850 / 1.00	2625 / 1.00
stubs.Set2stub_out	0 / —	0 / —	0 / —	3133 / 0.99
assert-equals2equal	1140 / 0.98	0 / —	0 / —	0 / —
xrange2range	984 / 0.99	78 / 0.60	52 / 0.65	0 / —
iteritems2items	3222 / 0.27	229 / 0.58	163 / 0.26	225 / 0.32
reverse-assert-arguments	764 / 0.47	0 / —	730 / 0.46	0 / —
remove-redundant-in	807 / 0.02	0 / —	693 / 0.20	1351 / 0.27
directly-dictionary-access	0 / —	1631 / 0.17	0 / —	0 / —

Listing 4
EXAMPLE OF THE "REVERSE-ASSERT-ARGUMENTS" PATTERN FOR MOVE EXPECTED VALUE TO LEFT

```
# Initial patch
1 x = 2
2 self.assertEqual(x, 1)
# or
1 y = 'hell0'
2 self.assertEqual(y, 'Hello')

# Output:
# AssertionError: 2 != 1 or
# AssertionError: 'hell0' != 'Hello'
# - hell0
# + Hello
```

```
# Integrated patch
1 x = 2
2 self.assertEqual(1, x)
# or
1 y = 'hell0'
2 self.assertEqual('Hello', y)

# Output:
# AssertionError: 1 != 2 or
# AssertionError: 'Hello' != 'hell0'
# - Hello
# + hell0
```

3) **Readability-improvement Pattern**: The patterns that improve code readability (`remove-redundant-in`, although `reverse-assert-arguments` improve output readability,)
4) Other Pattern (`directly-dictionary-access` is reversed when developer would not avoid `None` value)

Python language has an already strict coding convention such as PEP8. For example, these conventions defined whitespace position and number of character on each line. To detect these issues, OpenStack and some python projects are using coding style checker. Detected Language-specific patterns are not supported current coding style checker, and it can improve current coding style checker. This result helps other python projects' coding convention not only OpenStack projects.

On the other hands, Project-specific pattern depends on project dependency. Our approach can detect these patterns before new patch author submits code to the project. OpenStack has an own coding style guideline that defined based on python language. Detected patterns are not included in their guideline, it can be added as a new coding convention on OpenStack guideline. As the future work, we might detect other code improvement pattern instead of the OpenStack-specific patterns if we would adapt to the other projects.

We could not find out `remove-redundant-in` documents and that improvement pattern does not have an impact on execution. However, that change might improve code readability, and this approach helps to detect these implicit patterns.

> Our approach can detect frequently appeared code improvement patterns to solve similar problems easily. These patterns are not written on a project of language coding guideline. Detected patterns can be classified into 3 categories by purpose (Project-specific, Language-specific, and Readability-improvement pattern).

C. RQ2 Result: Which type of code improvement patterns that increase/decrease over time?

Figure 2 shows transition of code improvement patterns frequencies in OpenStack, and Table II shows transition of code improvement patterns accuracy. All count is different from Table I since this result does not contain in period 1 (2011-2-23 ˜ 2015-6-15) that are only used for detecting pattern. Although, period 5 is only used to evaluate patterns that from period 4.

Project-specific Pattern Category: This pattern category (`disk2disk_api` and `stubs.Set2stub_out`) appeared only in period 4 and period 5, and had 1.00 accuracy. Because these patterns are caused by OpenStack dependency changes and reviewers might be well-known changes.

Language-specific Pattern Category: Some language-specific pattern category (`xrange2range` and `assert-equals`) appeared period 2 and they has a more than 0.98 accuracy. However, in period 5, these patterns have not appeared than period 2. One of the reasons, all detectable these patterns might be fixed automatically such as by `six` library [12]. From this result, this approach can be used to survey language version popularity. Even `iteritems2items` depends on language function, it only has less than 0.58 accuracy. Because `iteritems` and `items` functions' return different value types between Python2 and Python3.

Readability-improvement and Other Pattern Category: Readability-improvement

[12] six library: https://pythonhosted.org/six/

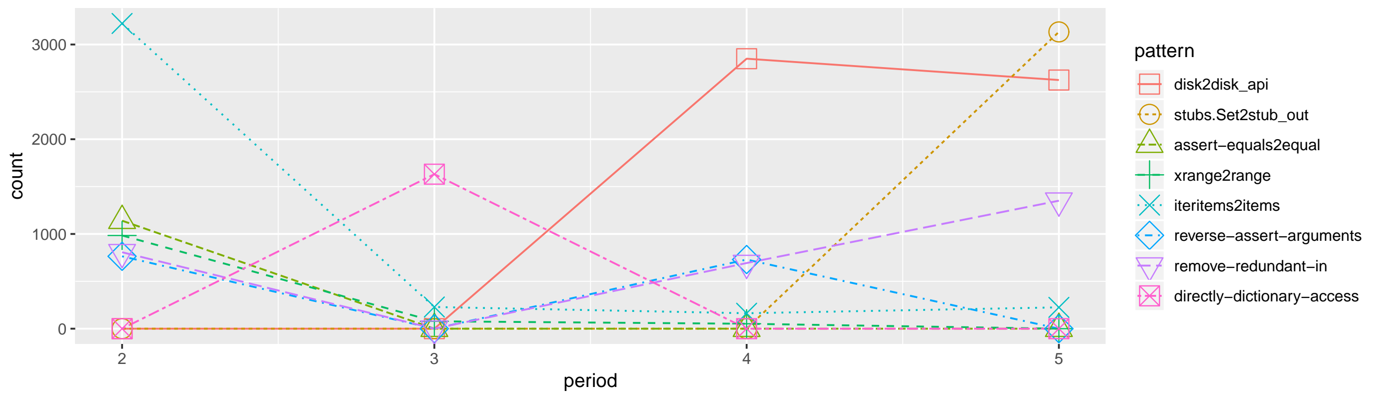

Fig. 2. Transition of Code Improvement Patterns Frequencies

pattern (remove-redundant-in and reverse-assert-arguments) and Other pattern (directly-dictionary-access) appear depends on periods, and the accuracies are less than 0.50. Since these improvement patterns do not have an impact on behavior, reviewers could ignore them. However, these improvement patterns frequently appear, the patterns would be useful to increase Readability-improvement patterns' accuracy.

In this RQ2, we confirmed that our proposed approach is likely to detect code improvement patterns. Using this approach, patch authors fix a source code based on the latest trend of project or language policy before submitting to code review. In our future study, we will survey how many code changes we need to analyze to detect efficient code improvement patterns.

> Detected pattern categories' frequencies and accuracies are changing over time. Our approach can track trends of projects to solve the latest problems.

V. Threats to Validity

External validity: We target source code that is only Python files from the OpenStack project. Also, we target only Python source code files. Surely, when we target other projects which use other programming languages, we may find different improvement patterns. Then, we can adapt our approach to analyze other software with other programming languages.

Internal validity: Our analysis compares only Initial patch and Integrated patch; it may misrecognize potential issues fixed through multiple revisions as another type of change. However, each revision also may include redundant changes that are not reflected in the Integrated patch. Our analysis conducts to investigate the effect of code review ignoring such changes.

To detect many improvement pattern, we normalized STRING and NUMBER literals. If we adopt another normalization way, such as coiling loop [13] that detects identifiable semantic idiom pattern, we may find more pattern that normalized identifier or concreted STRING or NUMBER literals.

To evaluate patterns, we divide dataset to 1:9 and 5 divisions. That did not clear which data combination is the best. However, at least 5 divisions can detect enough pattern frequencies changing by time series. We will conduct an empirical study to decide the best division in the future work.

Construct validity: We detect patterns from diffs of patches that changed through code review. Patterns depend on reviewers' policy even reviewer missed problems on the source code. To check patterns' usability, we will check patterns behavior in the future work.

Reliability: We classify patterns into three categories manually based on OpenStack document and StackOverflow information. To generalize our approach, we will classify patterns automatically by comparing with other project and platform in the future work.

VI. Related Works

A. Code Review

Many researchers have conducted empirical studies to understand code review [14]–[16]. Unlike our focus, most published code review studies focus on the review process or the reviewers' communication. While code review is effective in improving the quality of software artifacts, it requires a large amount of time and many human resources [17]. Various methods are proposed to select appropriate reviewers based on the reviewer's experience [18]–[21] and complexity of code changes [4]. In our work here, we focus on code changes based on feedback from the reviewers.

Code reviews are refactoring based on coding conventions [22]. Also, patch authors and reviewers often discuss and propose solutions with each other to revise patches [23]. 75% of discussions for revising a patch are about software maintenance and 15% are about functional issues [16], [24]. These studies help us understand which issues should be solved in the code review process. In particular, our approach detects some patterns (e.g. remove-redundant-in) that do not have an impact on code behavior. This result follows previous research results. However some patterns (e.g.

`directoly-dictionary-access`) have a small impact of source code behavior, even code review does not change other parts of source code. Our approach detects not only refactoring pattern, but this approach also detects patterns that have a small impact on the project.

B. Coding Conventions

Code convention issues also relate to our study because some code reviews are improving code based on coding convention [22], [25], [26]. Smit et al. [26] found that CheckStyle is useful for detecting whether or not source codes follow its coding conventions. Also, some convention tools such as Pylint released by Thenault check the format of coding conventions. In addition, Allamanis et al. [27] have developed a tool to fix code conventions. Also, Negara et al. [28] developed atool to detect code change pattern. However, to the best of our knowledge, little is known about how a patch author fixes to source code based on reviewers feedback.

VII. Summary

This study introduced a mining approach for code improvement patterns that are code changes based on reviewers feedback.

Using sequential pattern mining, we detect 3 categories of code improvement pattern. The results of our case study on the OpenStack project showed our approach can detect code improvement patterns that are not defined on OpenStack and Python coding guideline. They can help to project manager and language developer to improve current coding guideline. Although, each code improvement patterns' appeared frequencies are changing by time series. Our approach can track pattern transition timely, and it can help patch author to understand project policy.

The contribution of this study is the discovery of frequent patterns through code review. This proposed approach may help to design an issue detection. Also, we created a coding convention checker that detects project-specific patterns. If a patch author detects the possibility of changing these patterns before the code review request, the reviewers might be able to spend more time on other additional review requests.

Acknowledgment

We would like to thank the Support Center for Advanced Telecommunications (SCAT) Technology Research, Foundation. This work was supported by JSPS KAKENHI Grant Numbers JP18H03222, JP17H00731, JP15H02683, and JP18KT0013.

References

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

[2] G. van Rossum, B. Warsaw, and N. Coghlan, "Pep 8: style guide for python code," *Python. org*, 2001.

[3] S. Thenault *et al.*, "Pylint. code analysis for python."

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

[5] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: an empirical investigation," in *Proc. ESEM*, 2014, pp. 33–42.

[6] Y. Ueda, A. Ihara, T. Ishio, and K. Matsumoto, "Impact of coding style checker on code review -a case study on the openstack projects-," in *Proc. IWESEP*, 2018, pp. 355–359.

[7] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories," in *Proc. MSR*, 2017, pp. 269–279.

[8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proc. ICSE*, 2013, pp. 712–721.

[9] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, "Freespan: frequent pattern-projected sequential pattern mining," in *Proc. KDD*, 2000, pp. 355–359.

[10] T. Ishio, H. Date, T. Miyake, and K. Inoue, "Mining coding patterns to detect crosscutting concerns in java programs," in *Proc. WCRE*, 2008, pp. 123–132.

[11] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proc. ESEC/FSE*, 2007, pp. 25–34.

[12] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *Proc. ICDE*, 2001, pp. 215–224.

[13] M. Allamanis, E. T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, "Mining semantic loop idioms," *IEEE Transactions on Software Engineering*, vol. 44, pp. 651–668, 2018.

[14] 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 *Proc. MSR*, 2015, pp. 168–179.

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

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

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

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

[19] 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.

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

[21] 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 *Proc. ICSME*, 2015, pp. 261–270.

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

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

[24] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. MSR*, 2014, pp. 202–211.

[25] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Proc. ICSM*, 2008, pp. 277–286.

[26] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *Proc. ICSM*, 2011, pp. 504–507.

[27] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. FSE*, 2014, pp. 281–293.

[28] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," *Proc. ICSE*, pp. 803–813, 2014.