コードレビュー作業において頻繁に修正されるソースコード改善内容の分析

上田 裕己　石尾 隆　伊原 彰紀　松本 健一

オープンソースソフトウェアは，多数のパッチからのソースコードの変更提案を受け入れることで高機能・高品質なソフトウェアへと進化している．しかし，パッチ開発者からの変更提案がプロジェクトの実装方針に従っていないことも多く，ソースコードの内容を検証し適切な内容へと改善するために多くの時間コストがかかっている．変更提案の検証コストを削減するためには，パッチ開発者が変更提案前にソースコード上の問題を取り除くことが望ましいが，何を確認しているか具体的な内容は明らかとなっていない．そこで本稿では，コードレビューを通して行われたソースコード改善の内容とその出現回数を分析することで，変更提案に対して修正要求されることの多いソースコード改善の内容，すなわち開発者が確認している項目を明らかにする．4 つの Python プロジェクトのデータセットに対する分析の結果，コードレビューを通して行われる修正のうち 56.0%が動作に影響をもたないソースコード改善であることを確認した．また，コード改善のうちパッチ開発者が静的解析ツールを適用することで事前に検出可能な Python 言語の規約違反の問題が 13.4%である．本稿の貢献として，コードレビューで行われているソースコードの変更内容と，静的解析ツールの利用によってパッチ投稿前に検出可能なソースコードの改善内容を明らかにした．

Open-source software evolves into high-performance and high-quality software by accepting submissions of source code changes from many patch authors. However, source code changes that are submitted to a project does not always obey projects' coding conventions defined. Hence, reviewers waste a large amount of time to verify source code and suggest source code improvements for patch authors. The time to improve source code can be reduced if patch authors solve these issues by themselves before patch submissions. To investigate an understandable checklist for developers in addition to coding conventions, this study analyzes source code improvement contents that are frequently solved on code review. From four Python project datasets, we found 56.0% code changes do not have an impact on behavior. We also found that only 13.4% of code improvements could be verified by using static analysis tools. As a contribution, we revealed the source code changes in code review and the improvement contents that can detect before submitting the patch by using the static analysis tools.

1　はじめに

コードレビューは，ソフトウェア開発プロセスの 1 つであり，パッチ開発者が機能追加や欠陥修正のためにソフトウェアプロジェクトに投稿したソースコード変更提案 (以降，**パッチ**) を，複数の開発者 (以降，**検証者**) が検証する作業である．コードレビューはソフトウェアの信頼性を確保するための重要な活動であり，Czerwonka らは，検証者がコードレビューで行う指摘のうち，15 %はパッチの機能的問題に関する内容であることを報告している [1].

オープンソースソフトウェア開発においては，多数のパッチ開発者がそれぞれ独自に記述したソースコードを投稿する．そのため，ソフトウェアプロジェクトのソースコード記述を一貫した可読性の高いものにするためにソースコードの改善が重要となる．Rigby らはコードレビューにおいてスタイルの問題に関する指摘が行われていることを報告しており [2]，Bacchelli らは開発者へのインタビューを通じて，可読性の改善

Analysis of Prevalent Code Improvements through Code Review

Yuki Ueda, Takashi Ishio, Kennichi Matsumoto, 奈良先端科学技術大学院大学, Nara Institute of Science and Technology.

Akinori Ihara, 和歌山大学, Wakayama University.

コンピュータソフトウェア, Vol.0, No.0 (0), pp.0–0.

[研究論文] 2019 年 2 月 13 日受付.

などのソースコードの品質向上がコードレビューの大きな目的の 1 つであると確認している [3].

コードレビューは重要な作業であると同時に，検証者にとっては負荷の高い作業でもある．1 件のコードレビューに約 1 日，長い場合は数週間を必要とし，1 人の検証者が平均 6 時間 / 週を費やす [4] [5]．時間を費やす原因は，一度の検証ではパッチに含まれるすべての問題を発見または改善することが困難であり，繰り返しの検証が必要なためである [6].

コードレビューではソースコード中の欠陥の発見はもちろん，空白の追加や，変数名の変更など動作に大きな影響をもたない可読性の改善を目的とした指摘も多数報告されている [7]．本稿は，コードレビューにおいて指摘されることが多い可読性の改善 (以降，**コード改善**) 内容を明らかにする．それにより，コードレビューにおいて，コード改善点発見に対する静的解析ツールの有用性と，将来の静的解析ツールに求められる機能の発見を目指す．

パッチとして投稿されるソースコードの品質を向上させるために，プロジェクトはコーディング規約を定めている．一部のコーディング規約違反は，静的解析ツールによって自動検出することが可能であり，パッチ開発者はツールを用いてパッチ投稿前にソースコードを検証し，プロジェクト内のソースコード記述方式に一貫性をもたせることができる．しかし，コードレビューにおいて実施されるソースコード改善のすべてがコーディング規約に含まれているわけではなく，検証者がコーディング規約以外に何を確認しているか，また検証者がどの程度コーディング規約違反を指摘しているかは明らかとなっていない．本稿では，コードレビューを通して行われるソースコード改善と，静的解析ツールが検出できる規約違反を明らかにする．図 1 に調査質問に対応する分析対象を示す．RQ1 で対象とするレビューを通した指摘と RQ2 で対象とする静的解析ツールによる警告内容の一部は重複するため，両方で各分析対象に基づいた議論を行う．具体的には，以下の 2 つの調査質問に対する分析を行う．

- **RQ1: コードレビューを通してコード改善はどの程度行われるか**：パッチ投稿後にコードレビューを通して変更されたソースコードの変更内容を目視で確認し，検証者によって指摘されやすいソースコードの内容を明らかにする．また，各コード改善について，コードレビュー中の出現回数を比較することで，パッチ開発者がパッチ投稿前に特に注意すべきコードの問題と，そのうち静的解析ツールの導入で検出が可能である問題を明らかにする．
- **RQ2: 静的解析ツールによる警告はコードレビューを通して修正されているか**：パッチ投稿直後のソースコードに静的解析ツールを実行し，コードレビューを通して修正されやすいコーディング規約違反を明らかにする．また，コードレビュー前後での規約違反の数を比較することで，静的解析ツールとして実装されたコーディング規約がどの程度検証者の指摘方針を反映しているか否かを明らかにする．

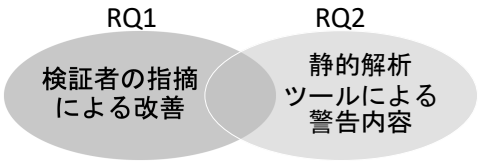


図 1: 各調査質問に対応する分析対象

調査対象として，コードレビュー管理システムである Gerrit 上で OpenStack が公開している Python プロジェクト群のコードレビューデータセット [8] と，GitHub 上で Microsoft，Google，Facebook が公開している Python プロジェクト群から収集したレビューデータセットを用いる．

本稿は，我々の過去の調査 [9] を発展させたものである．調査対象プロジェクトを複数に拡大することで一般性を高めたことと，複数の静的解析ツールを使用することで検出するコーディング規約違反の範囲を拡大したことが主要な違いである．

以降，2 章で関連研究，3 章で調査の方法と結果，4 章でその妥当性について議論し，5 章でまとめを行う．

2 関連研究

2.1 コードレビュー

オープンソースソフトウェア開発では，Gerrit [10] や Review Board [11] をはじめとしたコードレビュー管理システムを利用することで，開発者が投稿したパッチについて検証者と開発者間で円滑なコミュニケーションを行う[5]．これらのシステムに蓄積したデータを用いて，コードレビューの作業内容や役割を理解するために多くの研究が行われている [1][12][13]．コードレビューはソフトウェアの信頼性確保のために効果的だが，多くの時間的コストを消費することが確認されている [14]．レビュー時間削減のため，各パッチの内容に対して最適な活動記録を持つ検証者を選択する試みが行われている [2][15][16][17][18]．検証者の活動に注目した既存研究と異なり，本稿はソースコードの変更に注目することでレビュー時間の削減を試みている．Bacchelli らは，コードレビューの目的について開発者 873 人にインタビューを行い，337 人 (39 %) の開発者から，可読性の改善などのソースコードの品質向上が目的であると回答を得た [3]．また，コードレビューでは，ソフトウェアの動作に影響する機能的な指摘よりも可読性の問題やコーディング規約違反への指摘が最も多い [7]．検証者の指摘内容に注目した既存研究に対し，本稿はレビューを通して変更されたソースコードの内容から，パッチ投稿前に注意すべきソースコードの改善点を明らかにする．

2.2 コーディング規約

コードレビューではコーディング規約に基づいてリファクタリングを含むソースコード改善を行っている [7][19][20]．また，パッチ開発者と検証者は互いにソースコードを改善するための議論を行う [21]．特に，レビューで行われている議論の 75% はソフトウェアの保守性に注目し，機能的な問題に関する議論は 15%である [1][22]．コーディング違反の一部は静的解析ツールによって検出可能である．Panichella らはパッチ投稿前に静的解析ツールを導入することが，コードレビューで行われた修正作業の削減に貢献すると主張した [23]．本研究では個別のコーディング規約に対して，コードレビューを通して修正されやすい規約違反とそうでない規約違反を静的解析ツールを用いて調査する．

3 ケーススタディ

本稿では，コードレビュー管理システムおよび GitHub から収集したパッチに対して，コードレビューによって行われた変更を分析する．まず，パッチ開発者が作成し，プロジェクトに投稿したパッチを初版 $Patch_1$ とする．また，検証者による指摘とパッチ開発者による修正を繰り返してプロジェクトに採用されたパッチを最終版 $Patch_n$ とする．コードレビューによる変更内容を分析するため，初版 $Patch_1$ から修正されないままプロジェクトに採用されたパッチは分析対象から除外する．

コードレビューによる変更内容は多岐にわたるため，RQ1 では，パッチが対象プロジェクトに追加するソースコードのチャンク（断片）のうち，$Patch_1$ と $Patch_n$ の間で変更されたもの（以降，**変更チャンク**）の内容を目視で分析する．変更チャンクは `diff` コマンドで出力したソースコードの差分から得られる追加行と削除行によって構成されている．このとき，文字の置き換えなどの変更は追加と削除の組み合わせによって表現する．RQ2 については，$Patch_1$ を適用した状態のソースコードと $Patch_n$ を適用した状態のソースコードに対して静的解析ツールを実行し，コーディング規約違反に関する調査を行う．

3.1 データセット

データセットとして，OpenStack，Google，Microsoft，Facebook が公開しているプロジェクト群を用いる．対象とするプロジェクトは，コードレビューの管理に Gerrit Code Review や GitHub を用いて膨大なレビュー活動を公開している．本稿ではこれらのプロジェクトに投稿されたパッチから抽出した変更チャンクを分析対象とする．

また，対象プロジェクトは Python 言語標準のコーディングスタイル PEP8 [24] に基づいた静的解析ツール Pylint を利用することによって，自動的に PEP8 に違反するソースコードを検出している．

表 1: 対象プロジェクト概要

プロジェクト名	観測期間	パッチ数	変更チャンク数	サンプルパッチ数	サンプルチャンク数
OpenStack	2011 – 2015	68,174	61,673	382	144
Google	– 2018	5,780	74,454	238	174
Microsoft	– 2018	2,827	25,795	116	60
Facebook	– 2018	658	3,345	28	5
合計	—	76,459	165,257	764	384

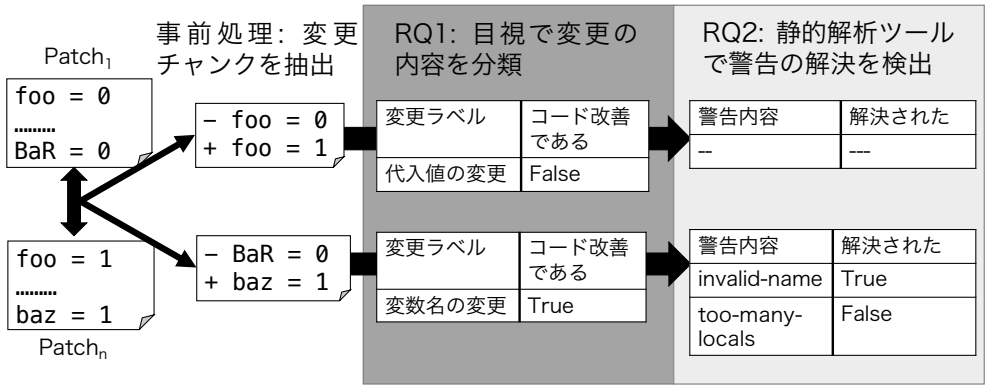


図 2: 変更チャンクと変更ラベルの抽出方法

Yang らが作成した Gerrit データセットに記録された OpenStack のコードレビューデータ [8] と Gerrit REST API [25] を利用し，OpenStack が Gerrit Code Review 上で公開しているプロジェクトから，パッチとそれに含まれる変更チャンクを収集する．また，GitHub API [26] を利用し，Google，Microsoft，Facebook が GitHub 上で公開しているプロジェクトからパッチとそれに含まれる変更チャンクを収集する．

表 1 に対象プロジェクトの概要を示す．RQ1 では各変更チャンクを目視で調査するため，変更チャンク数の偏りがでないようにすべてのプロジェクトの対象チャンクから層別サンプリングを行う．このとき，許容誤差は 5%，信頼度は 95%とする．RQ2 でも，一般性を保証するため，パッチ数の異なる Gerrit と GitHub のプロジェクトから 382 件ずつパッチをサンプリングする．

3.2 RQ1:コードレビューを通してコード改善はどの程度行われるか

本章では，コードレビューで頻繁に行われているコード改善の内容を明らかにする．コードレビュー前後のソースコード差分から $Patch_1$ と $Patch_n$ 間の変更チャンクを 384 件取り出す．また，変更内容を既存研究 [27] が定義したバグ修正パターン（Bug fix patterns）に基づきラベル付けを行い，コード改善であるか否かを目視で検出する．変更チャンクの抽出手順は以下の通りである．手順の例を図 2 に示す．

1. 事前処理として，$Patch_1$，$Patch_n$ のペアに `diff` コマンドを実行し，パッチ間の差分を得る．各パッチの内部に対応するチャンクが存在しており，かつ差分が存在しているものを変更チャンクとして抽出する．図 2 の例では，$Patch_1$ と $Patch_n$ の間で “`foo = 0`” が “`foo = 1`” に変更され，“`BaR = 0`” が “`baz = 1`” に変更されたという差分情報から，これらを 2 つの変更チャンクとして抽出している．
2. 各チャンクについて，目視で変更内容を確認

表 2: コード改善とみなす変更ラベル

改善名	コード改善内容
**ADD_REMOVE_NEWLINE	改行の追加削除
**ADD_REMOVE_SPACE	空白やインデントの追加削除
**CHANGE_VALUE_STYLE	変数名の大文字小文字，またはハイフン記号の変更
ABSTRACT_VALUE	変数の抽象化
ADD_REMOVE_ARRAY_ELEMENTS	配列要素の追加削除
ADD_REMOVE_DECORATOR	デコレータの追加削除
ADD_REMOVE_DICT_EELEMENTS	辞書要素の追加削除
ADD_REMOVE_RETURN	return 文の追加削除
CHANGE_COMENT	コメント文の変更
CHANGE_DEFINE_ORDER	変数を定義する順序の変更
CHANGE_FUNC_OBJECT	同一名の関数を呼び出すオブジェクトの変更
CHANGE_IMPORT	import 文で呼び出すライブラリの変更
CHANGE_STRING	文字列の変更
CHANGE_VALUE_NAME	“CHANGE_VALUE_STYLE” 以外の変数名の変更
UPDATE_VERSION	ライブラリのバージョン更新

** Pylint または Flake8 で自動検出可能な改善

し，ラベル付けをする．まず，チャンクが 1 種類のバグ修正パターンに該当する変更の場合は，その変更を機能的な変更とし，該当するバグ修正パターンをラベル付けする．チャンクが複数種類のバグ修正パターンを含む変更の場合には，その変更を大規模で機能的な変更とし，ラベル付を行わない．チャンクがバグ修正パターンに当てはまらない場合は，その変更をコード改善と分類し，表 2 のラベルから 1 つ割り当てる．表 2 のコード改善ラベルでは，静的解析ツールによって検出可能な改善とそうでない改善を分類しており， “ADD_REMOVE_SPACE”, “ADD_REMOVE_NEWLINE”, “CHANGE_VALUE_STYLE” の 3 つが Pylint, Flake8 の規約違反として検出可能な改善のラベルである．

図 3 に目視での分析によって分類した各変更内容の出現回数を示す．ここで， 384 件の変更のうち，ラベル付を行うことのできなかった大規模で機能的な変更 6 件 (1.6%) は省略している．また，紙面の都合上，2 回以上出現した変更内容のみを出力している．

コードレビューで行われた変更のうち，バグ修正パターンに当てはまらないコード改善は 384 件中 211 件 (54.9%) 件行われていた．この結果は，開発者がコード改善に注目していると回答した既存研究のインタビュー結果と合致する [3]．そのうち最も多く検出した改善内容は “CHANGE_VALUE_NAME”(変数名の変更) である．また，“ADD_REMOVE_SPACE” や “CHANGE_VALUE_STYLE” のように既存の静的解析ツールで自動的に検出が可能であるにもかかわらずコードレビューで行われた変更は合計 384 件中 46 件 (11.5%) であった．

“CHANGE_STRING” は，自動的に検出できない典型例である．単純な例では小文字から大文字に変更するものや出力文字列の末尾にカンマを追加するといったものがある．また，複雑な例では，固有名詞や記法の修正，依存するファイルパスやライブラリのバージョン番号の変更といったプロジェクトの知識やファイル間の依存関係などを把握しないと変更できないものまであった．変数名や文字列の推薦を行う技

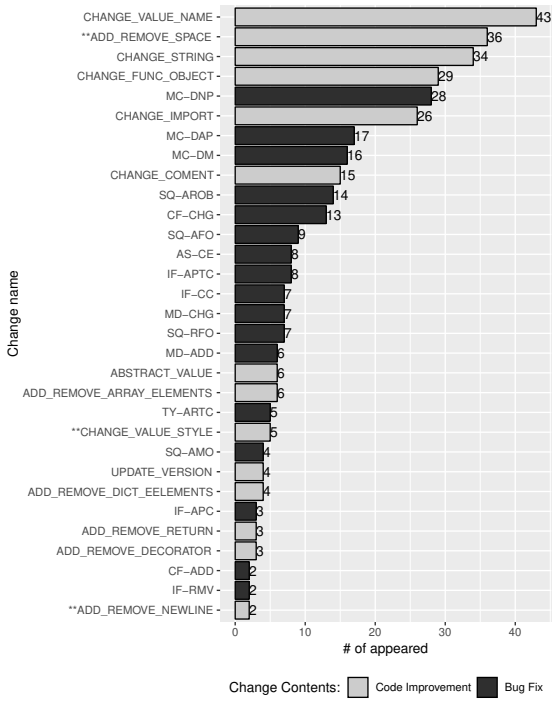


図 3: コードレビューを通して実施されたコード改善とバグ修正の出現回数
(総変更数: 384 件，うちコード改善は 211 件)

術 [28] などを静的解析ツールとしてプロジェクトが採用することでより多くのコーディング規約違反を検出可能になる．

RQ1 に対する答えは以下の通りである．

コードレビューを通して検出されるコード改善： 目視で 384 件のソースコード変更履歴を分類した結果，コードレビューで行われるソースコードの変更内容 384 件中 211 件 (56.0%) がコード改善であることを確認した．特に多く出現した変更は “CHANGE_VALUE_NAME” である．

コード改善のうち “CHANGE_VALUE_STYLE” や “ADD_REMOVE_SPACE” のようにパッチ投稿前に既存の静的解析ツールを利用することでパッチ投稿前に改善が可能な変更は 11.5% である．既存の静的解析ツールでは検出が困難な，プロジェクトの固有名詞や出力文字列の記法に関する修正，各プロジェクトがコーディングガイドライン †6, †7 を定義し，開発者間で共有することでこれらの問題は予防できる．また，将来静的解析ツールの機能を拡張，改善する際には表 2 で挙げた改善の自動検出が期待される．

3.3 RQ2: 静的解析ツールによる警告はコードレビューを通して修正されているか

RQ1 で分類したコードレビューで行われるコード改善のうち， 11.5% は静的解析ツールの実行により，パッチ投稿前に改善可能であることを確認した．本節では，静的解析ツールである Pylint と Flake8 を利用し，コードレビューを通して修正が行われたか否かに関わらず自動的な検出が可能なコーディング規約の違反を分析する．静的解析ツールの警告の中にはプロジェクトの複数のファイルから影響を受けるものもあり，プロジェクトの規模による警告数の偏りがでないように，パッチ数の異なる Gerrit と GitHub のプロジェクトから 382 件ずつパッチをサンプリングする．次に，$Patch_1$ から検出可能な規約違反と修正された規約違反の出現回数をそれぞれ求める．静的解析ツールが $Patch_1$ の状態のソースコードからコーディング規約違反を検出し，かつ $Patch_n$ のソースコードからコーディング規約違反を検出しなかったとき，コードレビューがその規約違反を修正したと判断する．$Patch_1$ と $Patch_n$ の両方の変更チャンク内から規約違反を検出した場合は，静的解析ツールは検出可能であったが，コードレビューはその規約違反を修正しなかったと判断する．このとき，コード改善に注目するため，プログラミングエラーとして分類されている警告は無視する．図 2 の例では，$Patch_1$ の変数 `BaR` を含む行から変数名の命名規約違反である “invalid-name” が検出され，$Patch_n$ の変数 `baz` を含む行からは違反が検出されないため，コードレビューによる修正として扱う．

表 3 に，パッチ投稿時に最も多く検出した，10 種類の警告とその出現回数を静的解析ツールごとに示す．警告の出現回数が総パッチ数よりも多いのは，1 つのパッチが変更するソースコードに複数の警告が含まれる場合があるためである．対象としたパッチ 382 件中 51 件 (13.4%) は，パッチ投稿前に静的解析ツールで検出が可能な修正である．パッチ開発者が

表 3: コードレビューを通して頻繁に静的解析ツールで検出される警告 (出現回数順)

Pylint 警告名	Pylint 警告内容	Gerrit 修正数/出現回数		GitHub 修正数/出現回数	
missing-docstring	関数やクラスにコメント文がない	10.4%	14 / 135	20.2%	41 / 203
invalid-name	命名規則違反	9.8%	10 / 102	17.4%	38 / 218
bad-continuation	可読性を下げる改行	16.4%	10 / 61	24.6%	35 / 142
no-self-use	クラス変数を利用しない関数	8.3%	5 / 60	13.0%	16 / 123
too-few-public -methods	public メソッドが少ないクラス	4.0%	2 / 50	13.0%	22 / 116
wrong-import-order	モジュールを読み込みがモジュール名のアルファベット順でない	2.6%	1 / 39	18.5%	23 / 124
unused-argument	使用されない引数の定義	2.0%	1 / 51	22.5%	23 / 102
fixme	解決されていない FIX ME コメント	5.9%	2 / 34	22.0%	24 / 109
too-many-arguments	1 関数に 10 以上の引数	13.2%	5 / 38	7.6%	8 / 105
too-many-locals	1 関数に 15 以上の変数定義	7.1%	2 / 28	11.8%	13 / 110
Flake8 警告名	Flake8 警告内容	Gerrit 修正数/出現回数		GitHub 修正数/出現回数	
E501	一行あたりに 80 文字以上記述されている	0.7%	1 / 147	50.5%	569 / 1,126
E114	コメント文のインデントが 4 文字の空白以外で構成されている	0.0%	0 / 134	44.0%	477 / 1,085
H405	複数行文字列中の不要な改行	11.1%	27 / 243	40.9%	375 / 917
H306	可読性を下げるモジュール読み込み順	2.0%	1 / 51	52.2%	554 / 1,061
H301	一行に 2 つ以上のモジュールの読み込み	0.6%	1 / 147	39.8%	303 / 762
H404	複数行文字列の行頭の改行	6.0%	16 / 270	31.0%	192 / 619
F821	定義されていないオブジェクトの利用	0.6%	1 / 167	27.5%	184 / 670
E302	関数定義の間に空行がない	0.0%	0 / 138	27.5%	167 / 646
F405	*を利用したモジュールの読み込み	0.0%	0 / 311	11.4%	54 / 473
F401	利用されないモジュールの読み込み	0.0%	0 / 96	28.4%	177 / 623

パッチ投稿前に静的解析ツールを利用することで，検証者のコストを下げることが可能になる．

Pylint は合計 1,950 件の警告を検出しており，Flake8 は合計 11,015 件の警告を検出している．2 つのツール間での警告検出数に差が生じた原因を調査するため，目視でプロジェクトのリポジトリを確認した結果，Flake8 よりも Pylint を採用しているプロジェクトが多く，パッチ開発者が Flake8 の警告を見る機会がないことが考えられる．例えば，最も多くのパッチをもつ Google は，自身のコーディングスタイルガイドライン [29] で Pylint の利用を推奨しているが，Flake8 については言及していない．

GitHub で管理されたプロジェクトにおいて，警告の修正率は 7.6%から 52.2%であり，ツールが警告する規約違反の多くはコードレビューでは検証者に指摘されないまま，プロジェクトに統合されている．同様に，Gerrit で管理された OpenStack プロジェクトでは，静的解析ツールによる警告のうち修正される割合は多くとも “bad-continuation” の 16.4% (10/61) である．特に Flake8 で検出した警告のいくつかはコードレビューを通して全く修正されない．Pylint によって最も多く検出した警告 “invalid-name” （変数や関数の命名規則違反）は表 2 中の “CHANGE_VALUE_STYLE” の改善に対応するが，GitHub プロジェクトにおいてこの警告の 82.5% (180/218) はレビューを通して修正されていない．

静的解析ツールで検出した規約違反の修正がレビューで行われない理由として，ツールの警告内容が検証者の修正方針と合致していない可能性が考えられる．例えば，OpenStack プロジェクトは独自のコーディングガイドライン [30] に基づいてソースコードを記述している．Flake8 によって検出した警告のうち，一度も修正されていない警告が存在する原因の 1 つとして， OpenStack のコーディングガイドに記述されていない，または記述されていたとしても

OpenStack 独自の例外を設けている [31] 点が挙げられる．そのため，プロジェクトが検証者に静的解析ツールの利用を促した場合，方針に合わない警告内容を確認するコストが余分に発生する．静的解析ツールは検出対象とするコーディング規約を選択することが可能であるにもかかわらず，多くのプロジェクトで対象とする規約の設定が変更されず運用されている事がわかっている [32]．警告の確認コストを削減するために，検証者のレビュー方針に合わせてツールを自動設定する技術が必要である．

RQ2 に対する答えは以下の通りである．

静的解析ツールによる警告はコードレビューを通して修正されているか： コードレビューを通して行われた変更のうち 13.4%のパッチはパッチ投稿前に静的解析ツールを実行することで事前に修正が可能な修正である．一方で，パッチ投稿前に静的解析ツールで検出可能な違反であっても警告の半分以上は，コードレビューを通して検証者に指摘されないままプロジェクトに統合されていた．検証者のレビュー方針と合致するよう，検証者，またはプロジェクト管理者は静的解析ツールで検出する規約違反をプロジェクトに合わせて設定する必要がある．

4 妥当性への脅威

内的妥当性： 目視によるソースコードの変更内容分析ではソースコードの差分，すなわち diff 形式のデータを使用して変更内容を分析している．ソースコードの差分情報の分析だけではソースコードの動作に与える影響の有無を確認できないパッチが存在する．したがって，RQ1 では，変更がコード改善であるか否かを diff 形式のデータだけで判断できない場合には Gerrit や GitHub で管理するソースコード全体を目視で確認した．今後はソフトウェアのテスト結果を用いることで動作へソースコードの動作に与える影響の有無を自動的に判断する．

コード改善として “ADD_REMOVE_SPACE” や “CHANGE_VALUE_NAME” のように修正方法によってはソースコードの動作に影響を与える可能性のあるコード改善が存在する．目視での検出で影響がないことを確認したが，外部ファイルからの参照などを調査することで影響が発見されることも考えられる．

外的妥当性： 本稿ではコーディング規約違反の検出に Pylint と Flake8 を利用している．RQ1 の分析の結果，ソースコードの改善だけでなく，固有名詞や記法の修正が多く発生していることを確認した．今後の課題として，スペルチェックツールを利用し，文字列の修正方法を検証する．

本稿では Python 言語で構成されたソースコードのみを対象としているため． Java 言語のようなインデントに意味を持たないプログラミング言語を対象とした場合に異なる分析結果になることが考えられる．例えば表 2 のラベルは本稿で対象としたソフトウェアの変更内容に基づいて分類している．そのため，異なるソフトウェアや言語を対象とした場合，定義したラベルでの網羅は保証できない．今後は異なる言語を利用するプロジェクトも対象に分析する．

5 おわりに

本稿ではコードレビューを通して行われるコード改善とその出現回数の分析と，静的解析ツールによるコード改善への効果を分析した．その結果，対象プロジェクトのコードレビューで行われる変更のうち 56.0%はコード改善であることを明らかにした．レビューを通して修正が行われたパッチのうち，13.4% はレビューを行う前にパッチ開発者が静的解析ツールで検出可能な修正である．例えば “ADD_REMOVE_SPACE”（空白やインデントの追加削除）のようなソースコードのスタイルの修正をコードレビューでの変更 384 件中，36 件行っている．一方で，“CHANGE_VALUE_NAME”（変数名の変更）や “CHANGE_STRING”（文字列の変更） のように，既存の静的解析ツールで検出ができないコード改善が頻繁に実施されている．

今後は，静的解析ツールでの解決が困難なコード改善例をパターンとして抽出することで，パッチ投稿前に開発者が注意すべきソースコードとその改善方法を自動的に示すシステムの構築を目指す．

謝辞 本研究は JSPS 科研費 JP18H03221，JP17H00731，JP15H02683，JP18KT0013 及びテレコム先端技術研究支援センター SCAT 研究の助成を

受けたものです.

参 考 文 献

[1] Jacek Czerwonka, Michaela Greiler, and Jack 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)*, pp. 27–28, 2015.

[2] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pp. 541–550, 2011.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pp. 712–721, 2013.

[4] Amiangshu Bosu and Jeffrey C Carver. Impact of developer reputation on code review outcomes in oss projects: an empirical investigation. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, pp. 33–42, 2014.

[5] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pp. 202–212, 2013.

[6] Amiangshu Bosu and Jeffrey C Carver. Impact of peer code review on peer impression formation: A survey. In *Proceedings of the 7th International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, pp. 133–142, 2013.

[7] Yida Tao, Donggyun Han, and Sunghun Kim. Writing acceptable patches: An empirical study of open source project patches. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*, pp. 271–280, 2014.

[8] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. Mining the modern code review repositories: A dataset of people, process and product. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR'16)*, pp. 460–463, 2016.

[9] 上田裕己, 伊原彰紀, 石尾隆, 松本健一. コードレビューを通じて行われるコーディングスタイル修正の分析. 第 25 回ソフトウェア工学の基礎ワークショップ（FOSE'18）, pp. 52–63, 2018.

[10] Gerrit code review. `https://www.gerritcodereview.com/`. Accessed: 2019-12-17.

[11] Review rboard. `https://www.reviewboard.org`. Accessed: 2019-12-17.

[12] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu 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)*, pp. 168–179, 2015.

[13] Rodrigo Morales, Shane McIntosh, and Foutse 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)*, pp. 171–180, 2015.

[14] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed 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)*, pp. 192–201, 2014.

[15] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Kenichi 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)*, pp. 141–150, 2015.

[16] Motahareh Zanjani, Huzefa Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, Vol. 42, No. 6, pp. 530–543, 2015.

[17] Mohammad M. Rahman, Chanchal K. Roy, and Jason 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 (ICSE'16)*, pp. 222–231, 2016.

[18] Xin Xia, David Lo, Xinyu Wang, and Xiaohu 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)*, pp. 261–270, 2015.

[19] Cathal Boogerd and Leon Moonen. Assessing the value of coding standards: An empirical study. *IEEE International Conference on Software Maintenance (ICSM'08)*, pp. 277–286, 2008.

[20] Michael Smit, Barry Gergel, H James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pp. 504–507, 2011.

[21] Jason Tsay, Laura Dabbish, and James 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)*, pp. 144–154, 2014.

[22] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar 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)*, pp. 202–211, 2014.

[23] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Would static analysis tools help developers with code reviews? In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, pp. 161–170, 2015.

[24] Pep 8 – style guide for python code. `https://www.python.org/dev/peps/pep-0008`. Accessed: 2019-12-17.

[25] Gerrit code review - rest api. `https://gerrit-review.googlesource.com/Documentation/rest-api.html`. Accessed: 2019-12-17.

[26] Github api v3. `https://developer.github.com/v3/`. Accessed: 2019-12-17.

[27] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, Vol. 14, No. 3, pp. 286–315, 2009.

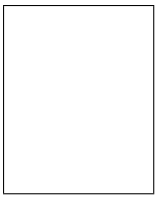
[28] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, pp. 281–293, 2014.

[29] Google python style guide. `https://github.com/google/styleguide/blob/gh-pages/pyguide.md`. Accessed: 2019-12-17.

[30] Openstack style guidelines. `https://docs.openstack.org/hacking/latest/user/hacking.html`. Accessed: 2019-12-17.

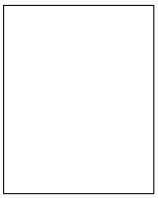
[31] F405 への違反に関する例外. `https://docs.openstack.org/hacking/latest/user/hacking.html#imports`. Accessed: 2019-12-17.

[32] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1, pp. 470–481, 2016.



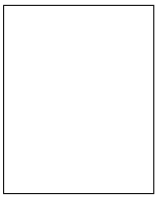
上田 裕己

2017 年島根大学総合理工学部卒業．2019 年 奈良先端科学技術大学院大学博士前期課程修了，同年同大学博士後期課程入学．ソフトウェア工学，特にコードレビュー支援の研究に従事．IEEE 学生会員．



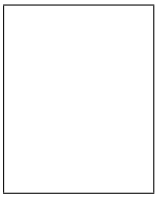
石尾 隆

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．2007 年大阪大学大学院情報科学研究科助教．2017 年奈良先端科学技術大学院大学情報科学研究科准教授．2018 年奈良先端科学技術大学院大学先端科学技術研究科准教授．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．電子情報通信学会，情報処理学会，IEEE，ACM 各会員．



伊原 彰紀

2007 年龍谷大学理工学部卒業．2009 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．2012 年同大学博士課程修了．同年同大学情報科学研究科・助教．博士（工学）．2018 年和歌山大学システム工学部・講師．ソフトウェア工学，特にオープンソースソフトウェア開発・利用支援の研究に従事．電子情報通信学会，IEEE 各会員．



松本 健一

1985 年大阪大学基礎工学部情報工学科卒業．1989 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．1993 年奈良先端科学技術大学院大学助教授．2001 年同大学教授．工学博士．エンピリカルソフトウェア工学，特に，プロジェクトデータ収集／利用支援の研究に従事．電子情報通信学会，情報処理学会 各会員，IEEE Senior Member．