G-42

プルリクエストにおける開発者の変更提案の分類

福元　春輝†　　伊原　彰紀†　　石尾　隆‡　　上田　祐己‡
Haruki Fukumoto　Akinori Ihara　Takashi Ishio　Yuki Ueda

1. はじめに

コードレビューは，実装者が機能拡張や不具合解決等のために変更したソースコードを対象に，検証者が欠陥有無の確認，可読性の評価を行う作業である．コードレビューは，高品質なソフトウェアをリリースするための作業であるため，複数人の検証者が，実装者とソースコード変更の妥当性について合意形成を図り，必要ならばソースコードを繰り返し修正する．したがって，コードレビューはソフトウェア開発プロセスの一連の作業において時間，作業量ともに最も高いコストを要する作業となっている [7].

伝統的なコードレビューでは実装者と検証者が対面で行う会議を開催し，ソースコードの検証を行っていた．昨今ではコードレビューのコスト削減，効率化のために，多くのソフトウェア開発プロジェクトが GitHub，Review Board, Gerrit などのオンラインコードレビューシステムを使用している．オンラインシステム上で実施するコードレビューは，実装者と検証者が異なる時間，異なる場所でも情報共有することが可能な方式（モダンコードレビュー [1]）として普及している．具体例には，GitHub におけるプルリクエスト機能がある．モダンコードレビューを採用するプロジェクトにおいて実装者がコードレビューを依頼する場合には，大規模な変更に比べて 5 行程度の小規模な変更の方がリリースするソフトウェアに統合されることが多い [2]. 従来研究では，変更されたソースコードの特徴に注目しており，具体的なソースコードの変更内容は明らかでない.

本論文では，実装者がコードレビューを依頼するソースコードの特徴を明らかにするために，従来研究において調査された変更規模に加えて，変更内容がリファクタリングのような命令の入出力に影響可否を調査する．さらに，ソースコードの変更内容によって，検証者の判断結果（統合/改善/不採用）の違いを明らかにする．ソースコード中の命令の入出力への影響可否の調査では，変更前後のソースコードのいずれか一方においてテストコードを生成し，他方に同テストを適用することでメソッドレベルの入出力結果の変更結果を明らかにする.

続く 2 章では関連研究を述べ，本研究の立ち位置を説明する．3 章では，本研究で用いる提案手法について説明をし，4 章で実際のデータを用いたケーススタディを行う．そして，5 章ではケーススタディで得た結果を分析し，考察を行う．最後に 6 章で，まとめと今後の課題について述べる.

† 和歌山大学，Wakayama University

‡ 奈良先端科学技術大学院大学，Nara Institute of Science and Technology

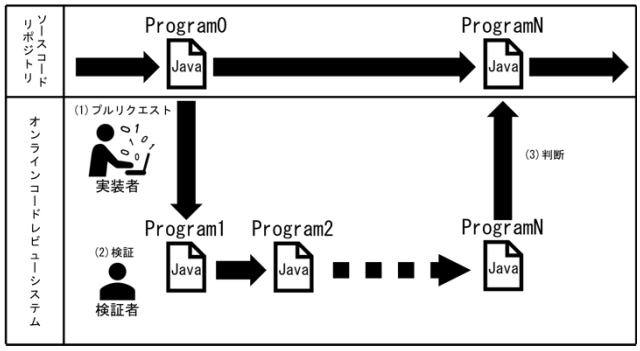

図 1 コードレビュープロセス

2. コードレビュー

2.1　コードレビュープロセス

オンラインコードレビューシステムを用いた，コードレビュープロセスでは， 3 つの作業で構成されている．図 1 は，本論文のケーススタディにおいて対象とする GitHub のコードレビュープロセスを示す.

(1) プルリクエスト

実装者はソースコードリポジトリ にあるソースコード $Program_0$ に変更を加え，変更したソースコード $Program_1$ をオンラインコードレビューシステムにアップロードする.

(2) 検証

検証者は，オンラインコードレビューシステムに登録された $Program_1$ を目視で確認し，欠陥有無の確認，可読性の評価を行う.

(3) 判断

検証者はコードレビューの結果，3 つ（統合/改善/不採用）の判断結果のいずれかを決定する．統合は，$Program_1$ を修正する必要がなく，リリースすることが可能なソースコードである．改善は，$Program_1$ を修正することでリリースできる可能性があるソースコードである．改善の場合，実装者は $Program_1$ を修正し，$Program_2$ として再度オンラインコードレビューシステムにアップロードする．検証者が統合と判断されるまで改善を繰り返し，N 回目にアップロードされたソースコードを $Program_N$ とする．不採用は，リリースすることができないソースコードである.

本論文では，実装者がコードレビューを依頼するソースコードの特徴を明らかにするため， $Program_0$ から $Program_1$ への変更内容を分析対象とする.

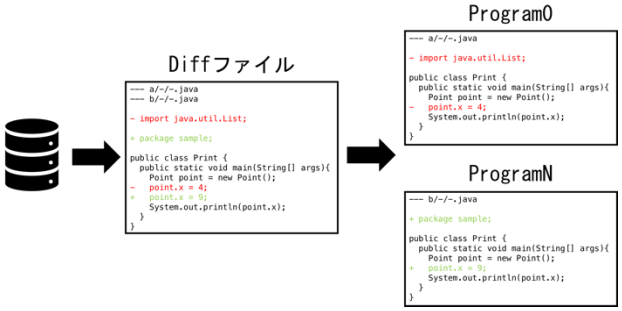

図 2 ソースコードの再現

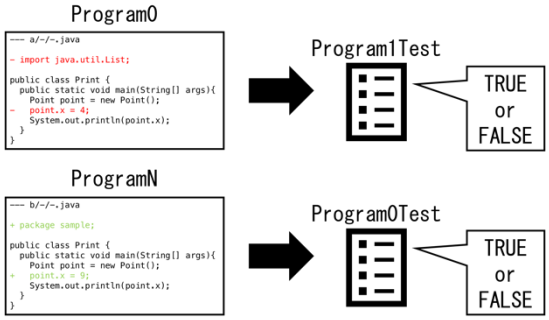

図 3 プログラムテストの実行

2.2 従来研究

ソフトウェア品質を向上するための作業として，コードレビューにおける欠陥検出の向上，作業効率を目的とする研究が多数行われている．

コードレビューにおいてソースコード中に検出される欠陥についての研究として，Rigby らは，伝統的な対面で実施するコードレビュー方式に比べて，モダンコードレビュー方式の方が依存関係のあるプログラムの追跡，過去の提案されたプログラム変更の追跡を容易にしていることを明らかにしている [5]．また，Tao らは，コードレビューにより検出されるソースコードの問題を実際のコードレビューの指摘を調査することで明らかにしている [4]．Weißgerber らは，実装者がコードレビューを依頼する場合には，大規模な変更に比べて 5 行程度の小規模な変更の方がリリースするソフトウェアに統合されることが多いことを明らかにしている [2]．これらの研究では，コードレビューによる効果，コードレビューの方法に明らかにすることを目的としているため，コードレビューを効率的に実施するためのソースコードの実装方法については明らかにしていない．検証者はソースコードの欠陥有無の確認よりもソースコード改善の方が多く実施されているため [1]，コードレビュー作業の効率化に向けて提案されるソースコードの具体的な変更内容について調査が必要である．

Ueda らは，モダンコードレビューに記録された，実装者が変更したソースコードと，検証者によるコードレビューを経て改善されたソースコードを収集し，ソースコードの改善パターンを収集する手法を提案している [3]．提案手法では，ソースコード中の変更された文字列を収集することで，ソースコードの可読性を高くするための空白の追加や，命名規則違反などの変更パターンを発見している．本手法は，ソースコードの記法を改善するパターンを抽出することが可能であるが，ソースコードの命令を改善するパターンを捉える事は容易ではない．

本論文では，変更内容の特徴による検証者の判断結果（統合/改善/不採用）の違いを明らかにする．

3. 分析手法

3.1 検証結果における仮説

従来研究では，変更規模が小さいプログラムであるほど採択されやすいことが明らかになっているが，プログラムの変更内容までは調査されていない．

変更内容がリファクタリングなどのような容易な変更であれば，検証者が欠陥の有無を確認しやすく，欠陥が入りにくく，採択となりやすいと考える．また，機能の追加など，変更が複雑になる程，欠陥が入りやすく，不採択になるか，何度も繰り返し改善するのではないかと考えられる．

3.2 概要

本論文では，実装者がコードレビューを依頼するソースコードの変更内容を明らかにするために，従来研究において調査された変更規模に加えて，変更内容が命令の入出力に影響与えるか否かを調査する．変更内容を明らかにするために，変更前後のソースコードのいずれか一方においてテストコードを生成し，他方に同テストを適用することでメソッドレベルの入出力結果の変更結果を明らかにする．本論文では，テストスイート自動生成ツールである EvoSuite を使用するが，EvoSuite は Java1.8.X で動くので Java 言語で実装され，MAVEN でパッケージを管理しているプロジェクトを対象とする．また，ソースコードの変更内容による検証者の判断結果（統合/改善/不採用）の違いを明らかにする．

3.3 変更内容の計測方法

(1) $Program_0$ を有するスナップショットの取得
分析対象とするプロジェクトのリポジトリをクローンし，$Program_0$ がコミットされた時点のソースコード一式を取得する．

(2) $Program_0$ のファイルパスの取得
図 2 に示すように，$Program_0$ と $Program_1$ の diff（差分情報）から $Program_0$ のファイルパスを取得する．

(3) $Program_0$ を対象としたテストコード作成
(2)で取得した $Program_0$ のファイルパスを用いて，変更前の $Program_0$ からテストコードを生成する．テストコードの作成にはテスト自動生成ツール EvoSuite を用いる [6]．Evosuite は Java クラスのテストスイートを自動的に生成し，テストの実行を行う事ができるツールである．テスト生成対象である Java クラスのクラス名とクラスパスを与える事で，コードカバレッジを最大化する Junit テストを生成する事ができる．Evosuite は Eclipse などのプラグインとして使用する事ができるが，本研究ではコマンドラインで使用した．また，本論文では，テストコードの作成後に，作成したテストを実行するために依存関係のあるライブラリを MAVEN でダウンロードする．コマンドによりライブラリはほとんど自動取得ができるが，EvoSuite 本体である evosuite-1.0.6.jar と evosuite-standalone-runtime-1.0.6.jar，テストを実行するためのの junit-4.12.jar，hamcrest-core-1.3.jar は取得できないので別で取得した．

(4) $Program_1$ を有するスナップショットの取得
$Program_1$ を含むブランチはプルリクエストを閉じた，もしくは merge した後に，ブランチから消される事が

あるため，GitHub から直接クローンすることで取得できないことがある．したがって，本論文では，図 2 に示すように $Program_1$ と diff ファイルから $Program_1$ のスナップショットを再現する．

(5) $Program_1$ を対象としたテスト実行

図 3 のように，(3)で作成した $Program_0$ のテストケースを $Program_1$ に実行し，テスト結果を取得する．テストの実行結果が TRUE の場合，当該ソースコードの変更は出力結果に影響を与えないものであると考えられる．一方で，FALSE の場合，当該ソースコードの変更が出力結果に影響する変更であると考えられる．

変更内容の計測方法では，$Program_0$ を対象にテストコードを生成し，$Program_1$ に実行した．これとは反対に，$Program_1$ を対象にテストコードも生成し，$Program_0$ へのテスト実行も行う．その理由は，機能追加のようにプログラムが単純に追加された変更の場合は $Program_0$ において作成したテストコードに追加されたプログラムのテストケースを含まないため $Program_1$ へのテスト実行において True が出力される．一方で，$Program_1$ で生成したテストケースを $Program_0$ にテスト実行すると False が出力されるためである．したがって，本論文ではテストコードを生成元のソースコード，および，その出力結果によって 4 種類に分類する．

(Pattern1) ［TT: True-True］

$Program_0$，$Program_1$ のそれぞれで作成したテストコードは，テスト実行において True を出力する．この場合のソースコード変更では，出力結果に影響を与える変更が行われていないと考えられる．したがって，実装者はリファクタリングやインデントの変更などのプログラムの出力に影響を与えていない変更を提案していると考えられる．

(Pattern2) ［TF: True-False］

$Program_0$ で作成したテストコードを $Program_1$ において実行した場合は True を出力し，$Program_1$ で作成したテストコードを $Program_0$ において実行した場合は False を出力する．これは，$Program_0$ に対する操作や入力データを $Program_1$ も同様に処理することができるが，$Program_1$ のみが受け付けるような追加の操作や入力データが存在することから，プログラム追加の変更であると考えられる．

(Pattern3) ［FT: False-True］

$Program_0$ で作成したテストコードを $Program_1$ において実行した場合は False を出力し，$Program_1$ で作成したテストコードを $Program_0$ において実行した場合は True を出力する．これは，$Program_1$ に対する操作や入力データを $Program_0$ も同様に処理することができるが，$Program_0$ のみが受け付けるような追加の操作や入力データが存在することから，プログラム削除の変更であると考える．

(Pattern4) ［FF: False-False］

$Program_0$，$Program_1$ のそれぞれで作成したテストコードは，テスト実行において False を出力する．これは，$Program_0$，$Program_1$ に対するそれぞれの操作や入力データをお互いに処理することができないということから，出力結果に影響を与えるプログラム追加・削除の両方が行われていると考えられる．

3.4 変更規模の計測手法

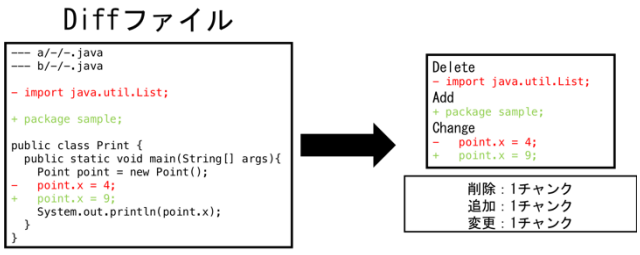

図 4 規模の分析

$Program_0$ と $Program_1$ の 2 つのファイルを対象に diff コマンドを用いることで，図 4 のようにソースコード変更が 3 種類（プログラムの追加・削除・変更）に分類される．図 4 のように，変更行が連続するプログラム片を 1 チャンクとし，各行の文頭の記号が「−」の場合は 1 チャンク削除，「＋」の場合は 1 チャンク追加，「−」「＋」が混在する場合は 1 チャンク変更とそれぞれ表される．

3.5 検証者の判断結果に寄与する変更の特徴分析

本分析では，実装者が変更したソースコードの変更内容による検証者の判断結果（統合/改善/不採用）の違いを明らかにするために決定木分析を行う．決定木分析は，従属変数に影響する説明変数を見つけ．樹木状のモデルを作成する分析方法である．本分析では，説明変数として変更内容の特徴を用いる．具体的には，変更されたソースコードのファイル数，3.3 で述べたプルリクエストとごとのテストコードの 4 種類の出力結果の割合，プログラム追加，削除，変更のチャンク数（3.2，3.3 において述べたテストコードの出力結果 4 パターン出力結果の割合，変更規模を用いる．また，従属変数を検証者の判断結果（統合/改善/不採用）とする．

決定木の作成には，R パッケージ rpart を使用する．

4. ケーススタディ

4.1 データセット

本分析では，Google が開発する guava プロジェクトを対象にケーススタディを行う．guava プロジェクトは，java core library であり，Google が開発する多くの Java ベースのソフトウェアで利用されるライブラリである．GitHub において公開される Java 言語で実装されているソフトウェアの中でスター数が 2019 年 3 月時点 1 位であるソフトウェアであり，多数の研究で分析対象として利用されている．guava プロジェクトでは，コードレビューの依頼は GitHub のプルリクエスト機能を使って行われ，2019 年 X 月時点において検証者によるコードレビューが完了しているプルリクエスト 343 件が登録されていた．

4.2 分析結果

分析対象プロジェクトにおける分析対象となるプルリクエスト 343 件を対象に，変更内容の計測を行った．ただし，対象となるプルリクエストには，変更対象のファイルに java プログラムを含まない，プロダクトコードを含まない（テストプログラムのみ含まれる），EvoSuite によってテストコードが自動生成されないプルリクエストが存在していた．本論文ではこれらプルリクエストを分析対象外としたため，158 のプルリクエストが残った．また guava プロジェクトでは，Google の検証者以外のプルリクエストが採

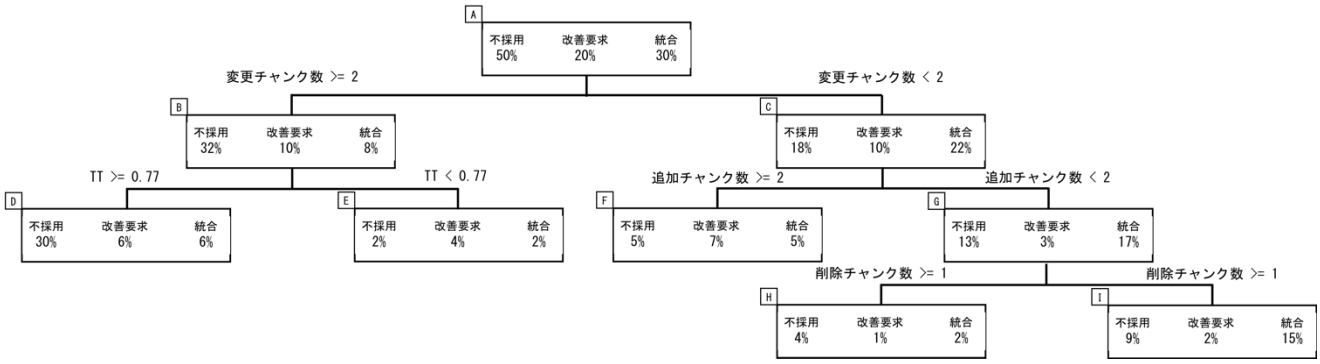

図 5 レビュー結果の決定木

択されることがないため，Google の検証者以外が有益なプルリクエストを行った場合，一度不採択となり，Google の検証者から別のプルリクエストとして再投稿されることがある．また，変更されたソースコードに関係のない理由で検証者が判断結果を下すことがある．これらのプルリクエストは本論文では対象ではない．ただし，このようなプルリクエストは自動抽出することができないため，各プルリクエストに投稿された検証者のメッセージを目視で確認した．その結果，本論文で対象とするプルリクエストは 102 件となった．この中で変更されたソースコードが不採用/統合/改善と検証者に判断されるのは，それぞれプルリクエスト全体の 50%，20%，30%であった．図 5 は，本論文で分析対象としたプルリクエスト 102 件において決定木分析を行った結果を示す．決定木の枝と葉にはそれぞれ ID として(A)〜(I)を付している．この結果から，本論文では 3 つの知見（コードレビューに投稿される変更内容の特徴，検証者が統合/改善/不採用と判断したそれぞれの変更内容の特徴）を得た．

（知見 1）検証者が不採用と判断した変更内容の特徴

分析対象のプルリクエスト全体の 30%は，検証者が不採用と判断し，その変更内容は 2 チャンク以上の変更をし，変更を加えたソースコードをテストした結果，約 77%以上のプログラムが出力結果に影響を与えない変更であった（図 5(D)を参照）．我々は，多くの変更がリファクタリングのような出力に影響を与えないため，検証者は統合と判断する可能性が高い変更であると考えていたが異なっていた．本論文では検証者が不採用と判断した理由を明らかにするためにコードレビュー のコメント，および，テストコードを目視で確認した．その結果，不必要な変更や記述内容は仕様である等を理由にリファクタリングでも不採用と判断されていた．また，本論文で EvoSuite が作成したテストコードが，実装者が追加，変更したプログラムのテストケースを自動生成していないことがあり，テストケースの自動生成方法は今後の課題とする．

（知見 2）検証者が統合と判断した変更内容の特徴

分析対象のプルリクエスト全体の 17%は，検証者が統合と判断し，その変更内容は 2 チャンクより少ない変更，かつ，2 チャンクより少ない追加であることがわかった（図 5(G)を参照）．さらに削除が 1 チャンクもない場合（図 5(I)を参照）は，全体 15%が採用されていた．従来研究 [2] の結果と同様に，変更規模が小さい場合，検証者は統合と判断する可能性が高いことを確認した．

（知見 3）検証者が改善と判断した変更内容の特徴

分析対象のプルリクエスト全体の 4%は，検証者が改善と判断し，その変更内容は 2 チャンク以上のプログラム変更，かつ，変更を加えたソースコードをテストした結果，約 77%未満のプログラムが出力結果に影響を与えない変更であった（図 5(E)を参照）．改善を要求されたプルリクエストは，分析対象の中で少なく，また，決定木分析において特定の変更の種類に偏っていないため，検証者の判断結果を予測するのは困難であると考えられる．

5. 制約と今後の課題

本論文では，guava プロジェクトが保有する 102 件のプルリクエストを対象に分析を行った．リポジトリのプルリクエストを対象として分析を行った．限られた分析対象プロジェクト，および，プルリクエスト数の結果から，異なるプロジェクトにおいても同様の結果が得られるとは限らない．しかしながら，変更内容が命令の入出力への影響に基づき検証者が下す判断結果を検討した論文はない．調査した，変更内容の調査から得た知見は，実装者がプルリクエストを提出する前に変更したソースコードを向上する方法を提案できると期待する．

また，Evosuite によるテストコードの自動生成を行ったが，変更部分のテストを正しく行うことが出来ていないケースを確認した．今後は，Evosuite によるテストケースの生成数を増加することで解決方法を検討する．

6. まとめ

本論文では，コードレビューにおいて，実装者の変更提案から検証結果を予測する手法して，実装者の変更提案したプログラムの内容を理解するためにプログラムテストを使った手法を提案した．

Google が開発する guava プロジェクトを対象にケーススタディとし，ソースコードの変更内容や変更規模を抽出し，決定木により検証結果を分類した．変更規模の小さい提案は採用される可能性が高いことが分かった．

5 章で確認したように，プログラムのテストツールの検討や対象とするリポジトリの範囲拡大をすることで，手法の精度が向上すると考えられる．今後はこれらの問題を解決することで，検証者の判断結果をプルリクエスト提出前に予測する方法の確立を目指す．

謝辞

本研究は JSPS 科研費 JP18KT0013 の助成を受けたものです．

参考文献

[1] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In Proc. ICSE'13, pp. 712–721

[2] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In Proceedings of MSR '08, pages 67–76. ACM, 2008.

[3] Yuki Ueda, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto, "Mining Source Code Improvement Patterns from Similar Code Review Works," in 2019 IEEE 13th International Workshop on Software Clones (IWSC)

[4] Y. Tao, D. Han, and S. Kim, "Writing Acceptable Patches: An Empirical Study of Open Source Project Patches," in Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 271–280.

[5] Rigby, P. C. and Bird, C.: Convergent Contemporary Software Peer Review Practices, Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, pp. 202–212 (2013).

[6] http://www.evosuite.org/

[7] ABDEL-HAMID, T. K. 1988. The economics of software quality assurance: A simulation-based case study. MIS Quart. 12, 3, 395–411.