社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

事前構文定義を必要としないリファクタリング検出手法の提案

古藤　寛大†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科
大阪府吹田市山田丘 1-5
E-mail: †{k-kotou,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　リファクタリングとは，ソフトウェアの外部的な振る舞いを保ちつつ内部構造を変更する作業である．リファクタリングはソフトウェアの保守性を向上させるための重要な技術であり，これまでに様々な研究が行われている．その中の一つにリファクタリング検出がある．リファクタリング検出とは，異なる二つのバージョンのソースコードを入力として，バージョン間の差分情報からリファクタリングを検出する技術である．リファクタリング検出はソフトウェアの改善に関する実証研究等でも活用されている．しかし，これまでのリファクタリング検出の手法では，あらかじめ定義した構文パターンと一致するリファクタリングしか検出できないため，定義されていないリファクタリングの検出は不可能であった．そこで，本研究では事前定義を必要としないリファクタリング検出の手法を提案する．具体的には，二つのバージョンのソースコードそれぞれからテストを自動生成し，生成テストをもう片方のソースコードに対して実行する．どちらのテストも通過した場合はリファクタリングが行われたとして検出する．本手法を適用することで，事前に構文定義できないリファクタリングの検出において，従来手法を大幅に上回る 80 % 以上の再現率を発揮した．また，リファクタリングが検出されたコミットのメッセージを解析し，リファクタリングの実施を示唆する新たな単語のパターンとして，“use”，“stream”，“reimplement” を発見した．

キーワード　リファクタリング検出，自動テスト生成，リファクタリングパターン，Java

1. はじめに

リファクタリングとは，ソフトウェアの外部的な振る舞いを保ちつつ内部構造を変更する技術である [1]．リファクタリングは実際のソフトウェア開発でも頻繁に行われており [2] [3]，ソフトウェアの保守という面で重要である．リファクタリングによって，不具合を引き起こすきっかけになりやすいソースコード内の “コードスメル” [1] [4] を取り除ける．一方で，リファクタリングを行うことでソフトウェアの品質に影響を及ぼしうる．具体的には，リファクタリングの実施がソフトウェアのバグを引き起こすきっかけになるという調査 [3] がある．そのため，リファクタリングがいつどのように実施されたか知ることが必要となる．

これらの理由から，リファクタリングを検出する研究が盛んに行われている．リファクタリング検出とは，変更が行われたソースコードの差分からリファクタリングが行われた箇所を特定する技術である．リファクタリング検出ツールの活用によって，バグを誘発する変更の特定 [5] やコードレビューの補助 [6] が可能になる．

従来のリファクタリング検出は，ソースコードを静的に解析する．具体的には，変更が生じたソースコードの差分のうち事前定義された構文パターンに該当する箇所をリファクタリングが行われたとして検出する．そのため，従来手法では事前に構文定義されていない構文パターンのリファクタリングを検出できない．また，ソースコードの一部を別の機能等価な処理に置き換えるリファクタリングは数多く存在しており，それら全てを構文定義することはできない．

そこで，本研究の目的を従来手法では検出できないリファクタリングの検出とする．そのための手段として，事前構文定義を必要としないリファクタリング検出手法を提案する．具体的には，変更前後で振る舞いが変わらないソースコードの差分箇所をテスト実行に基づいて自動的に検出する．提案手法の実現にはテストが必要なため，本研究では自動テスト生成ツールの EvoSuite [7] を用いてテストを用意した．EvoSuite は生成元のプログラムに対してその振る舞いを最大限網羅できるテストを生成する．次に，変更前のコードから生成されたテストを変更後のコードに実行し，変更後のコードから生成されたテストを変更前のコードにも実行する．リファクタリングの実施前後で入力に対する出力の結果は変わらないため，テストを実行して振る舞いに変化が生じなければリファクタリングとみなせる．

本研究では，提案手法と従来手法との間でリファクタリング検出の性能を比較した．実験では，検出された変更内容がリファクタリングであるか目視確認を行った．また，提案手法で検出されたリファクタリングを含むコミットのメッセージを分析して，コミットメッセージとリファクタリングの関係性につ

いて調査した．本手法を適用することで，事前に構文定義できないリファクタリングの検出において，従来手法を大幅に上回る 80 % 以上の再現率を発揮した．また，リファクタリングが検出されたコミットのメッセージを解析し，リファクタリングの実施を示唆する新たな単語のパターンとして，“use”，“stream”，“reimplement” を発見した．

2. 準　　備

2.1 自動テスト生成

自動テスト生成とは，与えたプログラムに対してその振る舞いを確かめるテストを生成する技術である．現在様々な自動テスト生成の研究が行われている．例えば，ランダムな入力から単体テストを生成する Randoop [8] や遺伝的アルゴリズムを用いる EvoSuite [7] など様々な手法が提案されている．本研究で使用する自動テスト生成の技術は上記の手法を入れ替えたり組み合わせたりできる．

実際の開発現場では，ソースコードの記述変更の妥当性を検証するために回帰テストを実行することが多い [9]．回帰テスト用にテストを作成する必要があるが，ソフトウェアの規模が大きくなると開発者に負担がかかる [9]．そのため，自動テスト生成ツールはソフトウェア開発者の負担を減らす可能性があるとして期待されている．

2.2 リファクタリング検出ツール

リファクタリング検出は，ソースコードの変更箇所のうちリファクタリングが行われた箇所を検出する技術である．これまでにリファクタリング検出に関する様々な研究が行われている．例えば，ソースコードの差分箇所を構文定義ベースで解析する手法 [10] [11] [12] や，コードクローンをベースとした手法 [13] などが挙げられる．一方で，自動テスト生成ツールである EvoSuite を用いたリファクタリング検出手法を提案し，事前構文定義ベースの手法に対する優位差を示したのは本研究が初となる．

3. 研 究 動 機

従来のリファクタリング検出は，ソースコードを静的に解析する [10] [11]．具体的には，変更が生じたソースコードの差分のうち事前定義された構文パターンに該当する箇所をリファクタリングとして検出する．ソースコードの静的解析に基づくリファクタリング検出ツールの一つに，RefactoringMiner [10] がある．RefactoringMiner は，バージョン間で変更があった Java のソースコードの差分を入力として，差分内容が事前定義された構文パターンと一致する場合はそのリファクタリングが行われたとして出力する．RefactoringMiner は，他のリファクタリング検出ツールと比べて検出性能が優れていたという調査結果がある [14].

RefactoringMiner をはじめとする従来の検出手法では，事前に構文定義されていないパターンのリファクタリングを検出できないという課題がある．例えば，Fowler [1] が提唱したリファクタリングパターンに “`Substitute Algorithm`” がある．これは，既存の処理を別の機能等価な処理で置き換えるリファクタリングである．しかし，“`Substitute Algorithm`” には構文定義が存在せず，従来の構文解析に基づいた手法では上記のリファクタリングを検出できない．なぜならば，機能等価な処理に置換するリファクタリングの方法は数多く存在しており，それら全てには対応できないためである．また，事前定義されたパターンに該当する変更であれば，その変更内容がリファクタリングに該当しなくても誤検出する問題がある [15]．変更内容がリファクタリングであるか正しく判別する方法として，テスト実行で振る舞いの変化を確認する方法がある．しかし，ソフトウェア開発におけるテスト作成は開発者にとってコストの大きい作業である [16]．そのため，従来の事前構文定義に基づいた検出手法に対するテスト実行に基づいた検出手法の有効性は示されていない．

本研究では，上記課題を解決するアイデアとして，事前構文定義を必要としないリファクタリング検出手法を提案する．具体的には，記述に変更が生じた箇所についてテストを実行し，変更前後で実行時の振る舞いが変わらないか確かめる．このアイデアでは，振る舞いの変化をリファクタリングの判断基準とするため，ソースコードの構文内容に関わらずリファクタリングを検出できる．更に，振る舞いに変化が生じていればリファクタリングとして検出しないため，リファクタリングパターンに該当する差分箇所と振る舞いの変わる差分箇所がコード中に同時に存在するような変更の誤検出を回避しうる．本研究では，自動テスト生成ツールを用いて従来手法に対する提案手法の有効性を示した．

4. 提 案 手 法

本研究の目的を従来手法では検出できないリファクタリングの検出とする．そのための手段として，事前構文定義を必要としないリファクタリング検出手法を提案する．

本研究の具体的な手法は，コミット間の差分箇所に関するテストを実行して，テスト結果から変更前後で振る舞いが変わらないソースコードの差分箇所を検出することである．提案手法の概要を図 1 に示す．

提案手法は以下のプロセスで構成される．

入力　変更前後のソースコード

ステップ 1　テスト生成

変更前後のソースコードから自動テスト生成ツールの EvoSuite [7] を用いてテストを生成する．検証の対象はメソッドボディ内部の変更とした．なぜなら，自動テスト生成ツールはメソッド単位で対象のソースコードからテストを生成するためである．変更前後のソースコードの抽象構文木を取得して，メソッドのシグネチャが一致する箇所を探す．ここで，シグネチャの一致について，“戻り値の型とパラメータの型，メソッド名が全て一致すること” と定義する．シグネチャの一致するメソッドが変更前後のソースコード内に存在する時，メソッドボディ内部の構文に変化があったか確認する．

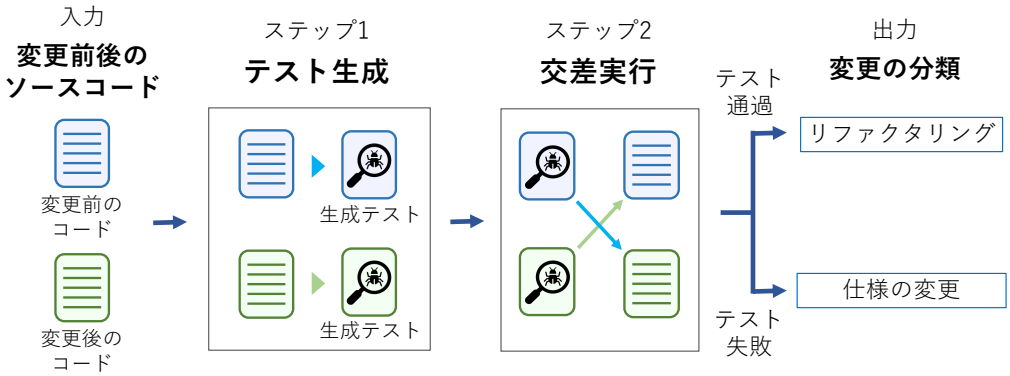

図 1: 提案手法の概要

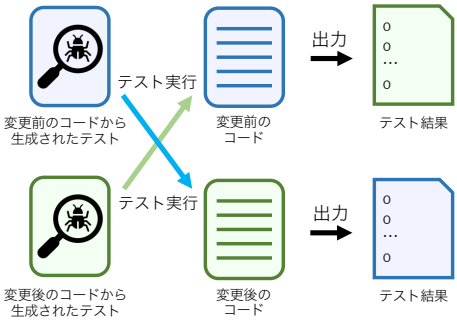

図 2: 交差実行の概要

ステップ 2　交差実行

EvoSuite は生成元のプログラムに対してその振る舞いを最大限網羅できるテストを生成する．したがって，変更後のソースコードが変更前のソースコードから生成されたテストを通過する場合は，変更後でも変更前の振る舞いを保持しているといえる．しかし，変更後のソースコードをテストするだけでは，リファクタリングでない新たな仕様追加の変更を検出できない場合がある．そこで，変更後のソースコードからもテスト生成して変更前のソースコードに対してテスト実行する (以後，このアイデアを “交差実行” と表記)．リファクタリングの実施前後で入力に対する出力の結果は変わらないため，交差実行で全テストを通過した場合，変更前後で振る舞いが変わらなかったことからリファクタリングとみなせる．一方，交差実行で失敗したテストがあれば，仕様の変更とみなす．

出力　全テストを通過すればリファクタリングとして検出，そうでなければ非リファクタリングとして検出

5. Research Questions

以下の Research Question（RQ）を設定した．

RQ1: 提案手法の検出性能

本調査では，従来手法では検出できなかったリファクタリングを提案手法で検出できているか確認することを目的として，RefactoringMiner [10] と提案手法で検出性能を比較する．

RQ2: 検出したリファクタリングを含むコミットのメッセージを解析

提案手法で検出できたリファクタリングを含むコミットのメッセージを解析して，リファクタリングの実施についてどの程度の割合で言及しているか調査する．本調査では，先行研究で紹介されているリファクタリングを含むコミットに頻出するパターン [17] を用いる．併せて，コミットメッセージにどのような単語が多く含まれているかも調査する．

6. 評 価 実 験

6.1 実 験 対 象

実験対象は実際のオープンソースソフトウェア (OSS) である Commons Lang および Commons Collections のうち，ソースコードの記述に変更があった直近の 141 コミットである．

また，実験対象である各 OSS は，プロダクトコードとテストコードで構成されている．テストコードは引数と戻り値がない JUnit で実行されるメソッド群から構成されており，EvoSuite はそれらのメソッドからテストを生成できないため，プロダクトコードのみを対象とする．実験対象のコミットのうち，筆者が目視でリファクタリングと確認できた変更を本研究におけるリファクタリングとしている．そのため，目視確認が困難な巨大コミットなどは本実験の対象から除外している．

6.2 実 験 設 定

提案手法を実現するにあたり，自動テスト生成ツールとして EvoSuite[1]を利用した．比較対象は RefactoringMiner[2]である．各手法を実験対象のコミットに実行して，検出した差分箇所のうち，目視確認でリファクタリングと判断できた変更を検出に成功したケースとして抽出する．

RQ1: 提案手法の検出性能

本実験では，RefactoringMiner と提案手法のリファクタリング検出性能の指標として Recall, Precision, F 値を用いる．

各手法を適用した結果を表 1 に示す．表では RefactoringMiner を “RMINER” と略記する．表 1 より，全ての指標において RefactoringMiner と比べて提案手法の検出性能が上回っていることが分かる．

(注1)：2022 年 10 月時点で最新バージョンの 1.2.0
(注2)：2022 年 10 月時点で最新バージョンの 2.3.2

次に，リファクタリングのうち Fowler が提唱するリファクタリングパターンを母集団とした時の Recall を表 2 に示す．1 行目は，全てのリファクタリングパターンに関する Recall を表している．2 行目と 3 行目はそれぞれ Fowler のリファクタリングパターンのうち，事前構文定義可能なパターンと事前構文定義不可能なパターンの Recall を表している．

事前構文定義不可能なパターンは以下の三つである．

- “Replace Loop with Pipeline”
- “Substitute Algorithm”
- “Replace Inline Code with Function Call”

表 1 を見ると，構文定義可能なリファクタリングパターンについては RefactoringMiner の検出性能が上回っている一方で，構文定義ができないリファクタリングの検出能力は提案手法が大幅に上回ることが確認できた．

また，リファクタリングパターンごとの検出数を調査して考察を行う．調査するパターンは Fowler 氏が定めた種別 [1] である．調査結果を表 3 に示す．

“Extract Method” と “Inline Variable” は各手法で Recall の値が大きい．

“Extract Method” はメソッドボディの一部を別のメソッドとして抽出するリファクタリングである．抽出後のメソッドは抽出前と振る舞いが同じであるため検出漏れの度合いが低かったと考えられる．“Inline Variable” に関しても同様で，変数宣言をインライン化するリファクタリングであることから，振る舞いが同じであり提案手法で検出できたと考えられる．

“Replace Loop with Pipeline” については，両方の手法で検出できた場合と提案手法でのみ検出できた場合があった．前者は拡張 for 文をそのまま forEach に置き換えており，anyMatch や allMatch といったパイプラインで置換していることが分かった．前者は for 文中の処理をそのまま forEach のラムダ式として置き換える変更であることから，事前構文定義が可能なパターンである．後者はパイプラインに置換される前のループ処理の書き方として様々なパターンが想定されるため，構文定義が困難であり提案手法でのみ検出できたと考えられる．

“Substitute Algorithm” は提案手法でのみ検出できた．このリファクタリングは明確な構文定義が存在しない．そのため，変更箇所を実行してみて振る舞いが変わらないか確認しなければリファクタリングか判断できない．以上のことから，“Substitute Algorithm” は，振る舞いの変化をテスト実行で確認する提案手法でのみ検出できた．

“Replace Inline Code with Function Call” は，メソッドボディの一部を既存メソッドの呼び出しで置き換えるリファクタリングである．従来手法ではコミット間での差分情報を基にリファクタリング検出を行うが，既存のメソッドやライブラリはコミット間で変更が行われていないため検出できない．一方，提案手法は実行時の振る舞いの変化に基づきリファクタリングを検出する．メソッドボディ内で具体的にどのような変更が行われたかに依存せずリファクタリングを検出できるため，Recall が従来手法に比べて高い．

一方で，Recall が従来手法に比べて低いリファクタリングパターンも存在した．本調査においては，Commons Collections のコミットに含まれていた “Pull Up Method” の検出性能が従来手法に劣っていた．その原因は，変更前後でシグネチャが一致するメソッドが存在しないことである．“Pull Up Method” はあるクラスのメソッドを継承元のクラスへ移動させるリファクタリングであり，シグネチャの一致を手法の適用条件とする提案手法で検出できない．

提案手法により，目視での判断が難しいリファクタリングを検出できる可能性がある．提案手法で検出できたが目視で判断

表 1: 検出性能の比較

	Commons Lang		Commons Collections	
	RMINER	提案手法	RMINER	提案手法
Recall	0.306	0.855	0.390	0.727
Precision	0.358	0.688	0.400	1.00
F 値	0.330	0.763	0.395	0.842

表 2: Fowler のリファクタリングパターンに関する Recall

パターン	Commons Lang		Commons Collections	
	RMINER	提案手法	RMINER	提案手法
全て	0.308	0.821	0.358	0.717
構文定義可能	0.818	0.727	1.00	0.125
構文定義不可能	0.107	0.857	0.081	0.973

表 3: 各 OSS におけるリファクタリングパターンごとの検出精度

(a) Commons Lang における検出精度

リファクタリング	個数	Recall	
		RMINER	提案手法
Extract Method	6	1.00	0.667
Inline Method	2	0	0.500
Inline Variable	2	1.00	1.00
Rename Variable	1	1.00	1.00
Replace Loop With Pipeline	16	0.188	0.813
Substitute Algorithm	12	0	0.917

(b) Commons Collections における検出精度

リファクタリング	個数	Recall	
		RMINER	提案手法
Extract Method	2	1.00	1.00
Pull Up Method	4	1.00	0
Rename Method	10	1.00	0
Replace Inline Code with Function Call	17	0	0.941
Replace Loop With Pipeline	5	0.600	1.00
Substitute Algorithm	15	0	1.00

が難しかった例として，while ループの実行前に行われていた処理をループ内に集約するリファクタリングがある．変更内容が複雑であるため，一見してリファクタリングと判断することは困難であるが，提案手法はテスト実行時の振る舞いが同じであればリファクタリングとして検出できる．そのため，実行経路が複雑で目視による判断が困難な場合でもリファクタリングを検出できると考えられる．

> **RQ1 への回答：**提案手法を適用することで，従来手法では検出できなかったリファクタリングの検出に成功した．特に，事前に構文定義ができない “Substitute Algorithm” や “Replace Inline Code with Function Call” に対しては顕著な成果を示した．

RQ2: 検出したリファクタリングを含むコミットのメッセージを解析

本実験では，提案手法で検出できたリファクタリングを含むコミットメッセージについて調べる．調査方法としては，提案手法で検出できたリファクタリングを含むコミットのメッセージ内に先行研究 [17] にて紹介されているキーワードが含まれるか文字列検索を行う．また，コミットメッセージに含まれている単語の出現回数を調査して，先行研究で紹介されたキーワード [17] 以外にもリファクタリングの実施を示唆するためのキーワードがないか調査した．

先行研究で紹介されたキーワードをメッセージ中に含んでいたコミットは，提案手法で検出できた全コミットのうち約 35.5 % であった．このことから，コミットメッセージのマイニングでは 4 割未満しかリファクタリングを検出できないことが分かった．

次に，コミットメッセージで頻出するキーワードを調査する．この調査の目的は，先行研究で紹介されたキーワード以外にリファクタリング実施時に使われやすいキーワードを見つけてリファクタリング検出に寄与することである．キーワードの出現数を図 3 に示す．図 3 の頻出する単語のうち，“use”，“stream”，“reimplement” はリファクタリングを示唆する新たなパターンとして活用できると予想する．

“use” キーワード

コミットメッセージを確認したところ，“use” の目的語はソースコード内に存在するメソッドや Java で提供されている Stream ライブラリやメソッドであった．このことから，ソースコードの一部を既存のメソッドやライブラリで置き換えたことを言及する際に用いられやすいキーワードではないかと考えられる．

“stream” キーワード

“Replace Loop with Pipeline” のリファクタリングを行う場合，Java のライブラリである Stream API を利用して for ループ処理と置き換えられる．そして，コミットメッセージ内に “stream” を含んでいたコミットでは Stream API を利用した “Replace Loop with Pipeline” が実行されていた．このことから，“stream” キーワードは当該リファクタリングを行ったことを明示する時に用いられやすいと考えられる．

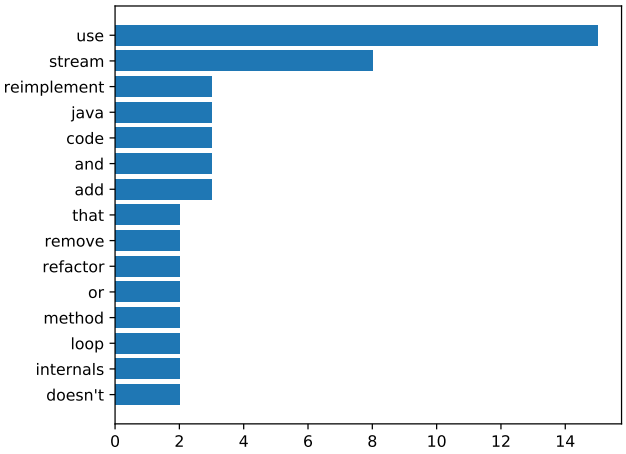

図 3: コミットメッセージの単語出現数

“reimplement” キーワード

“reimplement” は，既に存在している機能を実装し直すというニュアンスを持っている．何らかの機能追加やバグ修正を行うのであれば，“reimplement” よりも “add” や “fix” を用いる方が自然だと思われる．コメントを付与する変更を “implement” という単語で表現するとも考えづらい．このことから，“reimplement” は変更前の機能を保ちつつ内部構造を整理するリファクタリングと関連が深いと考えられる．

> **RQ2 への回答：**提案手法検出したリファクタリングを含むコミットのメッセージのうち，リファクタリングの実施を示唆するキーワードを含んでいたのは全体の 3 割程度であった．また，コミットメッセージ内の出現単語のうち，リファクタリングの実施を示唆していると考えられる新たなキーワードの候補を発見した．

7. 妥当性の脅威

本研究の評価実験では，テストの生成に EvoSuite [7] を用いた．他の自動テスト生成ツール，あるいは手動でテストケースを追加した場合には異なる実験結果を得られる可能性がある．

本研究で交差実行の対象とするのはメソッドボディ内に変更があったメソッドである．そのため，メソッドボディ以外のコード箇所が変更されている場合，同一コミット内でメソッドボディが変更されており，かつ変更があったメソッドからメソッド外部の変更箇所を参照していなければ検証できない．

また，提案手法では，生成テストの網羅率が低いと対象の差分箇所について仕様変更を検証しきれず誤検出を起こしうる．

自動テスト生成ツールによって生成されたテストが FlakyTest の場合だとリファクタリングを正しく検出できないケースもある．FlakyTest とは，テストの結果が一意に定まらないテストケースである [18]．RQ1 の実験において，テストの実行結果が一意に定まらない FlakyTest が生成された．それにより，リファ

クタリングの検出漏れが発生したため，提案手法の検出性能に影響を及ぼしうる．

本研究を行うにあたり，リファクタリングの目視確認を一人で行なったため主観に大きく依存する．

8. おわりに

本研究では，既存のリファクタリング検出手法が検出できないリファクタリングの検出を目的として，変更前後のコードから自動生成されたテストの交差実行により振る舞いの変わらない変更を検出する手法を提案した．提案手法では，変更前後のソースコードに対して自動テスト生成ツールを用いてテストを生成し，生成元のソースコードと生成したテストの組み合わせを入れ替えて相互に実行した結果を用いて分類を行う．評価実験を行った結果，既存のリファクタリング検出ツールである RefactoringMiner では検出できなかったリファクタリングの検出に成功した．また，提案手法で検出できたリファクタリングを含むコミットのメッセージを解析して，頻出するとされていたキーワードに含まれない単語が頻出していると分かった．

今後の研究課題として，提案手法で検出できるリファクタリングの種類を増やすことを考えている．具体的には，自動テスト生成の対象を親子関係にあるクラスまで拡大し，継承に由来するリファクタリングの検出へ対応することが挙げられる．加えて，本研究では Commons Lang と Commons Collections の二つの OSS に対してのみ実験を実施したが，それ以外の OSS においても実験を行い手法の有効性を示すことも今後の課題である．今回の提案手法では，`private` でないメソッドを対象としたリファクタリング検出を実行したため，`private` メソッド内での変更については検証していない．今後の研究の課題としては，それらの変更を提案手法で検出可能にすることも挙げられる．また，RQ2 において提案したキーワードが本当にリファクタリングを示唆する際に用いられるのか，大規模な実験により検証する必要がある．

謝辞 本研究は JSPS 科研費（JP20H04166，JP21K18302，JP21K11820，JP21H04877，JP22H03567，JP22K11985）の助成を得て行われた．

文　献

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] E. Murphy-Hill, C. Parnin, and A.P. Black, “How we refactor, and how we know it,” IEEE Transactions on Software Engineering, pp.5–18, 2012.
[3] M. Kim, T. Zimmermann, and N. Nagappan, “An empirical study of refactoring challenges and benefits at microsoft,” IEEE Transactions on Software Engineering, pp.633–649, 2014.
[4] E. Van Emden and L. Moonen, “Java quality assurance by detecting code smells,” Proc. the Ninth Working Conference on Reverse Engineering, pp.97–106, 2002.
[5] E.C. Neto, D.A.d. Costa, and U. Kulesza, “Revisiting and improving szz implementations,” International Symposium on Empirical Software Engineering and Measurement, pp.1–12, 2019.
[6] E.L.G. Alves, M. Song, and M. Kim, “Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits,” Proc. the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.751–754, 2014.
[7] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-Oriented Software,” Proc. Symposium and the European Conference on Foundations of Software Engineering, pp.416–419, 2011.
[8] C. Pacheco and M.D. Ernst, “Randoop: Feedback-Directed Random Testing for Java,” Proc. Companion to the Conference on Object-Oriented Programming Systems and Applications Companion, pp.815–816, 2007.
[9] A.K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, “Regression testing in an industrial environment,” Commun. ACM, pp.81–86, may 1998.
[10] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” IEEE Transactions on Software Engineering, pp.930–950, 2022.
[11] D. Silva, J.P. daSilva, G. Santos, R. Terra, and M.T. Valente, “Refdiff 2.0: A multi-language refactoring detection tool,” IEEE Transactions on Software Engineering, pp.2786–2802, 2021.
[12] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, “Ref-finder: A refactoring reconstruction tool based on logic query templates,” Proc. the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.371–372, 2010.
[13] B. Biegel, Q.D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, “Comparison of similarity metrics for refactoring detection,” Proc. the Working Conference on Mining Software Repositories, pp.53–62, 2011.
[14] L. Tan and C. Bockisch, “A survey of refactoring detection tools,” Proc. the Collaborative Workshop on Evolution and Maintenance of Long-Living Systems, pp.100–105, 2019.
[15] C. Görg and P. Weißgerber, “Detecting and visualizing refactorings from software archives,” International Workshop on Program Comprehension, pp.205–214, 2005.
[16] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” IEEE International Symposium on Software Reliability Engineering, pp.201–211, 2014.
[17] E. AlOmar, M.W. Mkaouer, and A. Ouni, “Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages,” Proc. International Workshop on Refactoring, pp.51–58, 2019.
[18] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” Proc. the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.643–653, 2014.