修士学位論文

題目

自動テスト生成を用いたリファクタリング検出手法の提案

指導教員

楠本 真二 教授

報告者

古藤 寛大

令和５年２月１日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 4 年度 修士学位論文

自動テスト生成を用いたリファクタリング検出手法の提案

古藤 寛大

内容梗概

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．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と呼ばれる研究がある．この研究は，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ものである．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．そのため，未定義のリファクタリングパターン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，生成されたテストをもう片方のバージョン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．

主な用語

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

目次

1 はじめに 1

2 研究動機 3

3 提案手法 6

3.1 ステップ 1：変更前後のソースコードからテストスイート生成 8

3.2 ステップ 2：生成テストとソースコードの交差実行 8

4 Research Questions 10

5 評価実験 11

5.1 実験対象 11

5.2 実験設定 11

5.3 RQ1: 提案手法の検出性能 12

5.4 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 20

6 妥当性の脅威 25

7 関連研究 26

7.1 自動テスト生成 26

7.2 リファクタリング 26

7.3 リファクタリング検出ツール 26

8 あとがき 27

謝辞 28

参考文献 29

図目次

1 拡張 for 文を用いたリファクタリング 3
2 従来手法で誤検出した事例 4
3 提案手法の概要 7
4 新たな条件を追加する変更 8
5 評価指標 13
6 パイプラインを用いたループの置き換え 15
7 java.util.Objects ライブラリを用いたコード書き換え 16
8 両手法で誤検出した変更箇所 18
9 提案手法で検出できた複雑なリファクタリングの例 19
10 コミットメッセージの単語出現数 24

表目次

1 検出性能の比較 12
2 Fowler のリファクタリングパターンに関する Recall 13
3 Commons Lang における種別ごとの検出精度 14
4 Commons Collections における種別ごとの検出精度 14
5 リファクタリングを含むコミットメッセージに頻出するパターン [1] 21
6 Commons Lang のコミットメッセージを解析した結果 22
7 Commons Collections のコミットメッセージを解析した結果 23

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 [2]．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 [3, 4, 5, 6]，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” [2, 7] 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 [5] 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研究である．リファクタリング検出ツールの活用によって，ブランチの安全な統合 [8] やバグを誘発する変更の特定 [9]，コードレビューの補助 [10] が可能になる．

従来手法は，ソースコードを静的に解析してリファクタリングを検出する．変更が生じたソースコードの差分を事前に定義されたリファクタリングの構文パターンと照合して，該当するパターンが存在すれば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，本研究では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．本手法を適用して，従来手法が検出できなかったリファクタリングを検出することを研究の目的とする．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動的解析に基づいて自動的に検出する．動的解析を行うにあたり，ソースコードの差分箇所を検証するテストを実行する．そのためにはテストを用意する必要があるため，本手法では自動テスト生成ツール [11] を用いて検証用のテストを用意した．自動テスト生成ツールで生成されたテストは，与えたプログラムの振る舞いをテストスイートとして規定していると捉えられる．つまり，ソースコードの変更前後について生成されたテストを実行して共に成功すれば，振る舞いに変化がないとみなせる．

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，検出された変更の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．

以降，2 節では研究動機として，従来のリファクタリング検出手法の改善すべき点について説明し，3 節では提案手法について説明する．4 節では，本研究で設定した Research Question について説明し，5 節でその Research Question に対し答えるために実施した実験の内容と結果および考察について述べる．6 節では本研究の妥当性の脅威について述べる．7 節で本研究の関連研究を，最後に 8 節で，本研究のまとめと今後の課題について述べる．

```
public class MapUtils {
  ...
  public static <K, V, E> void populateMap(final MultiMap<K, V> map, ...) {
-     final Iterator<? extends E> iter = elements.iterator();
-     while (iter.hasNext()) {
-       final E temp = iter.next();
+     for (final E temp : elements) {
        map.put(keyTransformer.transform(temp), valueTransformer.transform(temp));
      }
  }
  ...
}
```

図 1　拡張 for 文を用いたリファクタリング

2　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してリファクタリングを検出する [12, 13]．変更が生じたソースコードの差分箇所を構文的に解析し，事前に定義されたリファクタリングの構文パターンに該当する差分箇所が存在すれば，そのリファクタリングが行われたとして検出する．ソースコードの静的解析に基づく手法の一つに，RefactoringMiner [12] というリファクタリング検出ツールがある．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．

しかし，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．例えば，Fowler [2] が提唱したリファクタリングパターンに "Substitute Algorithm"(アルゴリズムの置き換え) がある．これは，既存のコード部分を別の実行方法に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しない．すなわち，従来の構文解析に基づいた手法では上記のようなリファクタリングを検出できない．

例えば， 図 1 に示すコード箇所[*1]は，Iterator を活用した while 処理を拡張 for 文で再実装している．この変更は，拡張 for 文という別の仕組みで while 処理を置き換えており，"Substitute Algorithm" が実行されている．しかし，現状のリファクタリング検出ではそのようなスモールケースに対して網羅的に対応することは困難である．なぜならば，図 1 に示したケース以外にも，機能等価な処理に置換するリファクタリングは数多く存在しており，それら全てには対応できないからだ．したがって，有限個のパターンにしか対応できない従来手法に変わる新たなリファクタリング検出手法が求められる．

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

[*1] https://github.com/apache/commons-collections/commit/29d79003

```
public class Validate {
  ...
  public static void isTrue(final boolean expression, final String message, ...) {
    if (!expression) {
-       throw new IllegalArgumentException(String.format(message, values));
+       throw new IllegalArgumentException(getMessage(message, values));
    }
  }
  ...
+ private static String getMessage(final String message, final Object... values) {
+   return values.length == 0 ? message : String.format(message, values);
+ }
  ...
}
```

図 2　従来手法で誤検出した事例

なくても誤検出する問題がある．

図 2 は RefactoringMiner が誤検出した変更[*2]である．この変更は，“Extract Method”(メソッドの抽出) というリファクタリングとして検出されている．具体的には，変更前の isTrue メソッド内に存在した String.format(message, values) のコード片が getMessage メソッドとして抽出されたと RefactoringMiner が検出した．しかし，実際には抽出先の getMessage メソッドにおいて values.length が 0 の時に message という別の値が return される仕様追加が行われている．したがって，リファクタリングでない変更をリファクタリングとして RefactoringMiner が誤検出している．

変更箇所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 [14]．そのため，従来の事前構文定義に基づいた検出手法と比較した時の，テスト実行に基づいたリファクタリング検出の有効性を示されていない．

まとめると，従来手法には以下の課題点がある．

- 事前構文定義されたリファクタリングパターンしか検出できない
- パターンに該当する記述の変更であれば，振る舞いの変化の有無に関わらずリファクタリングとして検出する

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

[*2] https://github.com/apache/commons-lang/commit/7f7d4b88

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してもリファクタリングとして誤検出しない．そのため，図 2 のようなリファクタリングでない変更の誤検出を回避しうる．

3　提案手法

2 節でも述べたように，従来のリファクタリング検出手法では事前に構文定義されたパターン以外のリファクタリングを検出できない．そこで，本研究では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．本手法を適用して，従来手法が検出できなかったリファクタリングを検出することを研究の目的とする．

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 3 に示す．

本手法を実現するためには，差分箇所に関するテストを用意する必要がある．そこで，本手法では自動テスト生成ツール [11] を用いて実現する．自動テスト生成ツールで生成されたテストは，与えたプログラムの振る舞いをテストスイートとして規定していると捉えられる．つまり，ソースコードの変更前後について生成されたテストの実行結果に変化がなければ，振る舞いにも変化がないとみなせる．手法で検出された変更箇所の内容を目視確認して，リファクタリングかどうかを確認する．提案手法における入力情報は変更前後それぞれのソースコードであり，出力情報はその変更がリファクタリングであるかどうかの結果である．

リファクタリング検出のプロセスは以下のステップで構成される．

入力　変更前後それぞれのソースコード

ステップ 1　変更前後のソースコードからテストスイート生成

ステップ 2　生成元のソースコードとテストスイートを入れ替えて交差実行

出力　ステップ 2 の交差実行を通過した場合はリファクタリングとして検出

以降では，各ステップについて説明する．

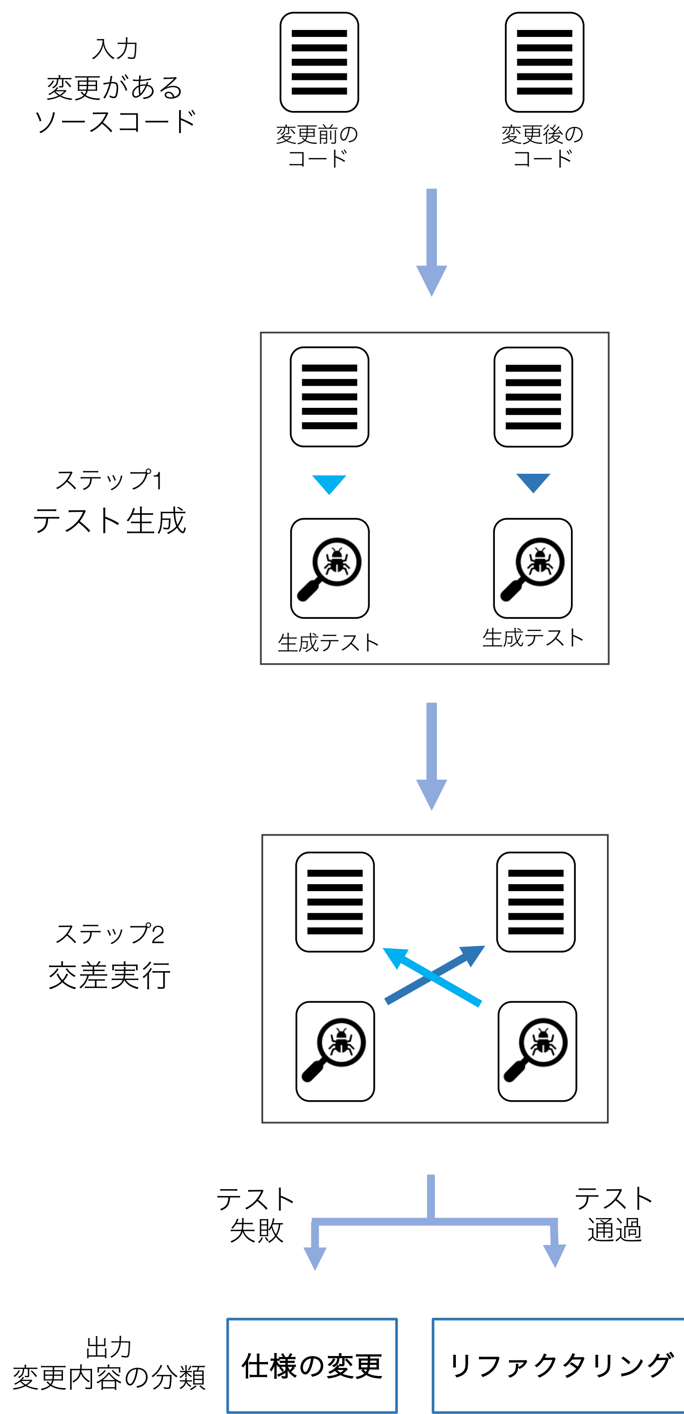

図 3　提案手法の概要

```
public class HaltonSequenceGenerator {
  ...
  public double[] skipTo(final int index) {
+   if (index < 0) {
+     throw new NotPositiveException(index);
+   }
+
    count = index;
    return get();
  }
  ...
}
```

図 4　新たな条件を追加する変更

3.1　ステップ 1：変更前後のソースコードからテストスイート生成

ソースコード内で変更が生じた箇所を調査する．ここで，調査の対象としたのはメソッドボディ内部の変更である．メソッドボディ内部の変更を対象としたのは，自動テスト生成ツールが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からだ．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，シグネチャの一致を以下のように定義する．

（戻り値の型が一致）　かつ　（パラメータの型が全て一致）　かつ　（メソッド名が一致）

シグネチャの一致するメソッドが存在する時，メソッドボディ内部の構文に変化があったか確認する．変更箇所が存在する場合はそのメソッドを検証するために変更前後それぞれのソースコードからテストを生成する．

3.2　ステップ 2：生成テストとソースコードの交差実行

変更前後のソースコードから生成されたテストスイートは，それぞれのソースコードの振る舞いを示している．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スイートを通過する場合は，変更後でも変更前のプログラムが満たすべき振る舞いを保持しているといえる．

しかし，それだけでは新たな仕様追加の場合でもテストスイートを通過しうる．例えば，図 4 のソースコード[*3]にある `if` 条件の追加が行われたとする．変更前のソースコードから生成されたテストスイート内に負数を引数とした `skipTo` メソッドを呼び出すテストが生成されなければ，仕様の変化を検出できずリファクタリングとして誤検出しうる．

そこで，本手法では変更後のソースコードからもテストスイートを生成して変更前のソースコードに対してテスト実行を行う (以後，このアイデアを “交差実行” と表記)．自動テスト生成ツールは条件分岐内の記述に関するテストも網羅的に生成するため，図 4 の例であれば，`if` 条件箇所に関するテストも

[*3] https://github.com/apache/commons-math/commit/01ba89bf

生成してリファクタリングの誤検出の可能性を減らせる．交差実行に成功した場合，変更前後で振る舞いが変わらなかったとみなせるため，リファクタリングとして検出する．一方，テストスイートの実行で失敗があれば，振る舞いが変わる変更，すなわちリファクタリングではない変更が行われたとみなす．

提案手法は上記のアイデアに基づき，変更前後のソースコードそれぞれから生成したテストスイートを交差実行し，振る舞いが同じソースコードの記述箇所を取得する．

4 Research Questions

提案手法を評価するにあたり，以下の Research Question（RQ）を設定した．

RQ1: 提案手法の検出性能

本研究の目標は，従来手法では検出できないリファクタリングの検出である．したがって，本調査では事前構文定義ベースのリファクタリング検出手法との間で検出性能を比較する．本調査の意図は，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である．実験題材として，GitHub 上で公開されているリポジトリのコミット履歴のうち，Java のソースコードの変更を対象としてリファクタリングを検出できているか調査する．本調査では，近年のリファクタリング検出手法において優れた検出精度を誇る RefactoringMiner [12] を比較対象として採用した．

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

実際の開発では，リファクタリングの実施内容をコミットメッセージに記載することが多い [1]．そのため，リファクタリングが行われたか判断する方法として，コミットメッセージからリファクタリングの情報をマイニングする手法が考えられる．そこで，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，リファクタリングが実施されたことを言及しているか調査する．本 RQ 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 [1] を用いる．コミットメッセージ内にパターンを含んでいるコミットの割合を調査すると同時に，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併せて調査する．

5 評価実験

5.1 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である Commons Lang[*4]及び Commons Collections[*5]のうち，ソースコードの記述に変更があった全 141 コミットである．本研究では，各リポジトリのうち，デフォルトブランチである `master` ブランチのみを対象とした．

また，実験対象である各リポジトリは，プロダクトコードとプロダクトコードの妥当性を検証するためのテストコードで構成されている．本調査では問題を簡易化するために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，巨大コミットなど目視での確認が困難な変更は本調査の対象から除外している．

5.2 実験設定

提案手法では自動テスト生成ツールとして，EvoSuite [11] を利用した．実験で用いた EvoSuite のバージョンは 2022 年 10 月時点で最新バージョンの 1.2.0 である．ステップ 1 で，変更前後それぞれのソースコードに関して EvoSuite で変更箇所に関するテストを生成した．比較対象として，RefactoringMiner[*6]を採用した．実験で用いた RefactoringMiner のバージョンは 2022 年 10 月時点で最新のバージョンの 2.3.2 である．RefactoringMiner は，私の知る限り最も多くの構文定義と照合できるリファクタリング検出ツールであるため，本研究の提案手法との比較対象として選択した [12]．RefactoringMiner を調査対象のコミットに実行して，ツールが検出したリファクタリングの可能性がある変更のうち，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．検証対象のリポジトリに含まれるソースコードをビルドする際には maven を用いた．

[*4] https://github.com/apache/commons-lang
[*5] https://github.com/apache/commons-collections
[*6] https://github.com/tsantalis/RefactoringMiner

5.3 RQ1: 提案手法の検出性能

本調査では，RefactoringMiner と提案手法のリファクタリング検出性能を調査する．

リファクタリング検出性能の比較

本手法で用いる指標を図 5 で表す．Recall は高いほど検出漏れが少なく，Precision は高いほど誤検出の度合いが少ないため，リファクタリング検出をはじめとする分類問題で評価指標として広く利用されている．また，検出性能を計測するために F 値も併せて算出する．F 値は Recall と Precision の調和平均を表している．両者は二律背反なので，Recall と Precision のどちらかに検出性能が偏っていないか F 値の算出結果から確認する．以下は各指標の計算式である．

$$\text{Recall} = \frac{TP}{FN + TP}$$

$$\text{Precision} = \frac{TP}{FP + TP}$$

$$\text{F 値} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

各手法を適用した結果を表 1 に示す．表では RefactoringMiner を “RMINER” として略記する．

表 1 より．Recall と Precision のいずれにおいても RefactoringMiner に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．

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 の検出性能を表 2 にまとめる．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，事前構文定義で解析が可能なパターンと事前構文定義では検出できないリファクタリングパターンの Recall を表している．具体的には，“`Replace Loop with Pipeline`”，“`Substitute Algorithm`”，“`Replace Inline Code with Function Call`” のリファクタリングパターンを “事前構文定義不可能なパターン”，それ

表 1 検出性能の比較

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

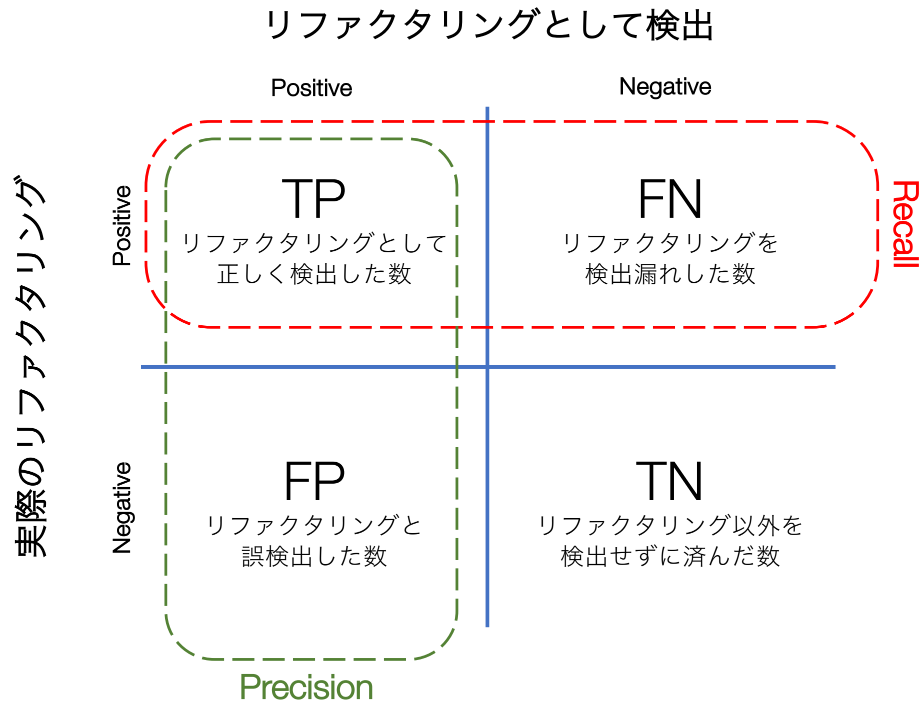

図5　評価指標

以外のリファクタリングパターンを “事前構文定義可能なパターン” とする．前者のリファクタリングパターンは開発者ごとに記述の方法が異なり，事前に構文定義を定められないため構文定義ができないパターンとした．

結果を見ると，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パターン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．

上記結果について，次項よりリファクタリングの種別ごとの検出数を調査して考察を行う．

表2　Fowler のリファクタリングパターンに関する Recall

	CommonsLang		CommonsCollections	
	RMINER	提案手法	RMINER	提案手法
全パターン	0.308	0.821	0.358	0.717
構文定義可能なパターン	0.818	0.727	1.00	0.125
構文定義不可能なパターン	0.107	0.857	0.081	0.973

リファクタリングパターンごとの検出性能の比較

Fowler が定めたリファクタリングパターンの種別ごとの検出数を調査した．調査結果を表 3 及び表 4 に示す．

“`Extract Method`” と “`Inline Variable`”(変数のインライン化) は各手法で高い Recall となった．

“`Extract Method`” はあるメソッドから部分的にコード片を抽出して新規作成したメソッドに移動させるリファクタリングである．そのため，抽出元のメソッドは抽出前と同じ振る舞いであるため検出漏れの度合いが低かったのだと考えられる．

“`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，メソッド外部から振る舞いを見た時に変化が生じないため提案手法で検出できたと考えられる．

“事前構文定義不可能なパターン” について，“`Replace Loop with Pipeline`”，“`Substitute Algorithm`”，“`Replace Inline Code with Function Call`” の全リファクタリングパターンで提

表 3　Commons Lang における種別ごとの検出精度

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

表 4　Commons Collections における種別ごとの検出精度

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

```
  public class CompositeSet<E> implements Set<E>, Serializable {
    ...
    @Override
    public Iterator<E> iterator() {
      ...
      final IteratorChain<E> chain = new IteratorChain<>();
-       for (final Set<E> item : all) {
-           chain.addIterator(item.iterator());
-       }
+       all.forEach(item -> chain.addIterator(item.iterator()));
        return chain;
    }
    ...
  }
```

(a) 各手法で検出できたリファクタリング

```
public final class CollectionSortedBag<E> extends AbstractSortedBagDecorator<E> {
    ...
    @Override
    public boolean containsAll(final Collection<?> coll) {
-       final Iterator<?> e = coll.iterator();
-       while (e.hasNext()) {
-           if (!contains(e.next())) {
-               return false;
-           }
-       }
-       return true;
+       return coll.stream().allMatch(this::contains);
    }
    ...
  }
```

(b) 提案手法でのみ検出できたリファクタリング

図 6　パイプラインを用いたループの置き換え

案手法が RefactoringMiner の検出数を上回った．

"Replace Loop with Pipeline"(パイプラインによるループの置き換え) について，両方の手法で検出できた図 6(a) のリファクタリングと提案手法でのみ検出できた図 6(b) のリファクタリングの二種類があった[*7]．前者は拡張 for 文を forEach のパイプラインで置き換えており，後者は 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．一方で，後者のリファクタリングはパイプライン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検出できなかったと考えられる．

"Substitute Algorithm" のリファクタリング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．変更が生じたコード箇所で当該リファクタリングが行わ

*7 https://github.com/apache/commons-collections/commit/29d79003

```
  import java.util.List;
+ import java.util.Objects;
  ...
  public class ComparatorChain<E> implements Comparator<E>, Serializable {
    ...
    public static boolean isEqualList(final Collection<?> list1, ... list2) {
      ...
      final Iterator<?> it1 = list1.iterator();
      final Iterator<?> it2 = list2.iterator();
-     Object obj1 = null;
-     Object obj2 = null;

      while (it1.hasNext() && it2.hasNext()) {
-         obj1 = it1.next();
-         obj2 = it2.next();
+         final Object obj1 = it1.next();
+         final Object obj2 = it2.next();

-         if (!(obj1 == null ? obj2 == null : obj1.equals(obj2))) {
+         if (!(Objects.equals(obj1, obj2))) {
              return false;
          }
      }
      return !(it1.hasNext() || it2.hasNext());
  }
    ...
  }
```

図 7　java.util.Objects ライブラリを用いたコード書き換え

れたか判断するには，記述が変わった箇所を実際に実行してみて振る舞いが変わらないか確認しなければならない．以上のことから，“Substitute Algorithm” のリファクタリングは，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”(関数呼び出しによるインラインコードの置き換え) は，インラインコード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．図 7 は提案手法で検出された変更のうち，当該リファクタリングが行われた箇所[*8]である．この例では，null チェック後に equals メソッドを実行する処理を java.utils.Objects ライブラリの equals メソッドの呼び出しで置き換えている．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，図 7 にある Objects.equals はコミット間で変更が行われていないため，事前構文定義ベースのリファクタリング検出ツールでは検出できない．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．内部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”(メソッドの引き上げ) の検出性能が RefactoringMiner に劣っていた．

[*8] https://github.com/apache/commons-collections/commit/e85e26e7

これは，手法の適用方法が原因である．“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る．しかし，本手法では変更前後でソースコード内のシグネチャが一致するメソッドを対象としており，変更後にはクラスからメソッドが親クラスへ移動されているため手法を適用できなかった．Fowler のリファクタリングパターンには，“Pull Up Method” 以外にもフィールドやメソッドを継承元のクラスや継承先のクラスに移動させるリファクタリングが存在するため，それらについても同様に検出できないことが予測される．

また，提案手法では Fowler のリファクタリングパターンに該当しないリファクタリングについても検出できた．例を挙げると，冗長な括弧の除去や if 文の三項演算子への置換などがある．リファクタリングは，Fowler が定義したリファクタリングパターン以外にも数多く存在しており，それらを有限個の構文定義で全て検出することはできない．そのため，提案手法は事前構文定義ベースのリファクタリングでは検出しきれない多くのリファクタリングを検出できる可能性を持っていると考えられる．

各手法で誤検出した事例として図 8 のソースコード[*9]を紹介する．この事例では，() -> getMessage(message, values) のコード箇所を nonNull メソッドから toSuppier メソッドへ抽出しているので，一見すると “Extract Method” というリファクタリングが適用されているように見える．しかし，このコミットでは getMessage メソッドも同時に書き変わっており，変更前後で null 判定の仕様が追加されているためリファクタリングの定義から外れる．両手法で誤検出した理由としては，nonNull メソッドの変更のみに着目して getMessage メソッドの処理内容も網羅的に考慮できなかったことが挙げられる．従来手法は構文定義と一致する変更パターンをリファクタリングとして検出する．そのため，パターンと一致する変更が起きていれば振る舞いの変化の有無を問わずリファクタリングとして検出する．Herzig らは，リファクタリングや機能追加など複数のタスクを伴った変更を “もつれた変更” と呼んでいる [15]．もつれた変更はソフトウェアの開発を効率よく行う上での妨げとなるが [15]，リファクタリング検出手法を適用する際に誤検出を引き起こす要因にもなりうることが本結果から分かった．

提案手法を適用することで，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．提案手法でのみ検出できたリファクタリングの例として 図 9 の変更を示す． 図 9 の変更は，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である．変更内容が複雑であることから，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．一方で，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．

[*9] https://github.com/apache/commons-lang/commit/756daad8

```
public class Validate {
  ...
  public static <T> T notNull(final T object, final String message, ...) {
-   return Objects.requireNonNull(object, () -> getMessage(message, values));
+   return Objects.requireNonNull(object, toSupplier(message, values));
+ }
+ private static Supplier<String> toSupplier(final String message, ...) {
+   return () -> getMessage(message, values);
  }
  ...
  private static String getMessage(final String message, final Object... values) {
-   return values.length == 0 ? message : String.format(message, values);
+   return ArrayUtils.isEmpty(values) ? message : String.format(message, values);
  }
}
```

(a) 変更があった箇所

```
public class ArrayUtils {
  ...
  public static int getLength(final Object array) {
    return array != null ? Array.getLength(array) : 0;
  }
  ...
  public static boolean isEmpty(final Object[] array) {
    return isArrayEmpty(array);
  }
  ...
  private static boolean isArrayEmpty(final Object array) {
    return getLength(array) == 0;
  }
  ...

}
```

(b) 呼び出されたライブラリメソッド

図 8　両手法で誤検出した変更箇所

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” のリファクタリングパターンに対しては顕著な成果を示した．

```
public class StringUtils {
  ...
  public static String join(final Iterator<?> iterator, final String separator) {
-
      // handle null, zero and one elements before building a buffer
      if (iterator == null) {
          return null;
      }
      if (!iterator.hasNext()) {
          return EMPTY;
      }
-     final Object first = iterator.next();
-     if (!iterator.hasNext()) {
-         return Objects.toString(first, "");
-     }

      // two or more elements
      final StringBuilder buf = new StringBuilder(STRING_BUILDER_SIZE); // ...
-     if (first != null) {
-         buf.append(first);
-     }

      while (iterator.hasNext()) {
-         if (separator != null) {
-             buf.append(separator);
-         }
          final Object obj = iterator.next();
          if (obj != null) {
              buf.append(obj);
          }
+         if (separator != null && iterator.hasNext()) {
+             buf.append(separator);
+         }
      }
      return buf.toString();
  }
  ...
}
```

図 9　提案手法で検出できた複雑なリファクタリングの例

5.4 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

本調査では，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，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内に 表 5 で示したパターン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パターン [1] 以外にもリファクタリングを実施したことを記述するためのパターンがないか調査した．

調査対象のコミットメッセージとメッセージから発見したパターンを 表 6 及び表 7 にまとめた．

コミットメッセージ内にパターンを含んでいたのは全体のうち約 35.5% である．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る．

次に，コミットメッセージで頻出している単語の特徴を調査する．この調査の意図は，表 5 のパターン以外にリファクタリングの実施を言及する際に使われやすいパターン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．単語の出現数を数えて順位づけした結果を図 10 に示す．なお，前置詞とメソッド名は調査の対象から除外した．前置詞はリファクタリングに限らずどのコミットメッセージにも用いられることや，メソッド名は一般化できないことが理由である．図 10 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

本調査でコミットメッセージを確認したところ，“use” の目的語はリポジトリ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単語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 `StreamAPI` を利用して `for` ループ処理と置き換えられる．そして，“stream” をコミットメッセージ内に含んでいるコミットでは `StreamAPI` を利用した “`Replace Loop with Pipeline`” が実行されている．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単語ではないかと考えられる．

表 5　リファクタリングを含むコミットメッセージに頻出するパターン [1]

Refactor	Removed poor coding practice	Change design
Mov	Improve naming consistency	Modularize the code
Split	Removing unused classes	Code cosmetics
Fix	Pull some code up	Moved more code out of
Introduc	Use better name	Remove dependency
Decompos	Replace it with	Enhanced code beauty
Reorganiz	Make maintenance easier	Simplify internal design
Extract	Code cleanup	Change package structure
Merg	Minor Simplification	Use a safer method
Renam	Reorganize project structures	Code improvements
Chang	Code maintenance for refactoring	Minor enhancement
Restructur	Remove redundant code	Get rid of unused code
Reformat	Moved and gave clearer names to	Fixing naming convention
Extend	Refactor bad designed code	Fix module structure
Remov	Getting code out of	Code optimization
Replac	Deleting a lot of old stuff	Fix a design flaw
Rewrit	Code revision	Nonfunctional code cleanup
Simplif	Fix technical debt	Improve code quality
Creat	Fix quality issue	Fix code smell
Improv	Antipattern bad for performances	Use less code
Add	Major/Minor structural changes	Avoid future confusion
Modif	Clean up unnecessary code	More easily extended
Enhanc	Code reformatting & reordering	Polishing code
Rework	Nicer code / formatted / structure	Move unused file away
Inlin	Simplify code redundancies	Many cosmetic changes
Redesign	Added more checks for quality factors	Inlined unnecessary classes
Cleanup	Naming improvements	Code cleansing
Reduc	Renamed for consistency	Fix quality flaws
Encapsulat	Refactoring towards nicer name analysis	Simplify the cod

“reimplement” キーワード

“再実装” 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” という単語よりも “add” や “fix” という単語を用いる方

表 6　Commons Lang のコミットメッセージを解析した結果

発見したパターン	ハッシュ値	コミットメッセージ
	19612d41	[LANG-1691] ClassUtils.getShortCanonicalName doesn’t use the canonicalName #949
	8b75877f	CPD: Re-implement deprecated code to use new code
	b6739ab9	PMD: Avoid using a branching statement as the last in a loop.
	49ef6b53	PMD: Implement equals()
.*mov.*	7cb81aa2	PMD: Remove extra parens
.*simplif.*	4deabadb	PMD: Ternary operators that can be simplified with || or &&
.*fix.*	cec7857a	Fix PMD issue: Either refer to method with static import or class
	6de0fc1a	Use Stream
	11b1bc19	Deprecate ThreadUtils code that defines custom function interfaces in favor of stock java.util.function.Predicate usage.
	cb0fbd4c	Use Stream.
	1297d7c3	Use Stream.
.*refactor.*	b6df0af2	Refactor internals and use Stream
.*refactor.*	c2960c48	Refactor internals and use Stream
	ea2e58fc	Use Stream.
	4b63f240	Use Stream.
	b0b3a46b	Use Stream.
.*extend.*	1d66289a	Extends clauses are redundant as java.lang.Object is a supertype for all classes. (#937)
	30a14e94	In-line single use local variable
.*add.*	717f163a	Add Streams.of(Iterator)
	ec93f3b7	StringUtils.join(Iterator
	39f08ac6	StringUtils.join(Iterator
.*add.*	62910e4f	Add LangCollectors

が自然だと思われる．コメントを付与する変更を「実装」という単語で表現するとも考えづらい．このことから，再実装という単語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深い単語ではないかと考えられる．

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単語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パターンの候補を発見した．

表 7　Commons Collections のコミットメッセージを解析した結果

発見したパターン	ハッシュ値	コミットメッセージ
	29d79003	Use for-each loop
	e85e26e7	When possible use java.lang.Objects#equals; eliminated a couple of nulls (#307)
	e30b4d37	Use java 8 and method reference. (#274)
.*creat.*	2431972c	Simple syntax for array creation.
	505722c3	Use final.
.*mov.*	01413dc5	Remove unnecessary Casting
.*creat.*	ffd2a02d	[COLLECTIONS-796] SetUniqueList.createSetBasedOnList doesn't add list elements to return value
	c1f26230	Reimplement SortedProperties#keys() a la Java 8.
	a18086f2	Reimplement SortedProperties#keys() a la Java 8.

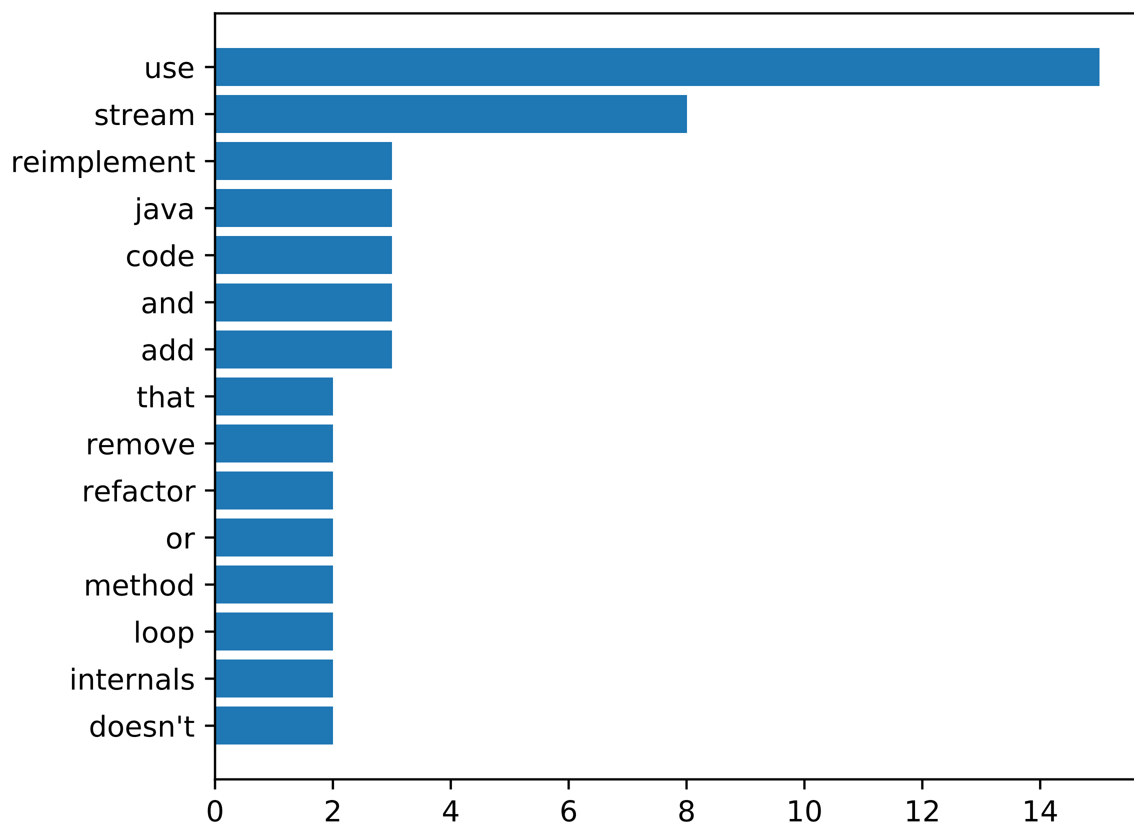

図 10　コミットメッセージの単語出現数

6 妥当性の脅威

本研究の実験対象は 2 つのリポジトリである．そのため，他のリポジトリで実験すると今回の調査とは異なる結果になる可能性がある．本研究の評価実験では，テストの生成に EvoSuite [11] 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．自動テスト生成ツールはメソッドの検証を目的とした単体テストを生成することが理由である．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．

本研究で使用した自動テスト生成ツールの EvoSuite は，テスト生成時に何らかのエラーが発生すると生成を中断する．本研究においても，本来あるはずのクラスをクラスパスとして読み取ることに失敗したことで手法の適用に失敗するケースがあった．

また，本研究で提案した手法では，生成されるテストの品質によって検出精度が異なってくる．例えば，新たな条件分岐を追加する変更があるとする．この場合，変更後のソースコードから生成されるテストに追加された条件分岐の内容を検証するテストが生成されていれば，変更前のソースコードに対して上記のテストを実行した際に異なる振る舞いとなることが分かるため，変更内容がリファクタリングでないと判断できる．しかし，カバレッジの低いテストが生成されると，条件分岐を検証しない可能性がある．その場合，変更前後での振る舞いの変化が検出できないため，変更内容をリファクタリングと誤検出する恐れがある．

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 [16]．5.3 節の実験において，テストの実行結果が一意に定まらない FlakyTest が生成された．そのため，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，各コミットでリファクタリングが行われたか筆者が目視で確認した．そのため筆者の主観に大きく依存する．

7 関連研究

7.1 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最大限網羅できる入力値を持つ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技術が研究されている．例えば，ランダムな入力から単体テストを生成する Randoop [17] や遺伝的アルゴリズムを用いる EvoSuite [11]，記号実行を用いる SUSHI [18, 19] など様々な手法が提案されている．本研究においては，ステップ 1 で使用する自動テスト生成の技術は，これら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 [20]．回帰テスト用にテストスイー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 [20]．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おり，本研究においても手法を実現させるための有効な手段として活用した．

7.2 リファクタリング

リファクタリングとは，外部から見た時のソースコードの振る舞いを変えずに内部構造を整理する作業を指す [21]．近年では，Martin Fowler 氏がリファクタリングパターンとしてリファクタリングを種別ごとに分類しており [2]，本研究のリファクタリング検出においても検出性能を測る 1 つの指標として用いている．

7.3 リファクタリング検出ツール

リファクタリング検出は，ソフトウェアのバージョン間で変更が生じたソースコードからリファクタリングの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．具体例を紹介すると，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 [12, 13, 22, 23] や，コードクローンをベースとした手法 [24]，グラフマッチングベースの手法 [25, 26] などが挙げられる．他にも，動的解析を用いたリファクタリング検証ツールとして，SafeRefactor [27] がある．SafeRefactor は，自動リファクタリングツールで記述を書き換えたコード箇所に関して Randoop [17] で自動生成したテストの実行を通じて振る舞いの変化が生じないか検証するツールである．私の知る限りでは，自動テスト生成ツールの 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ツールに対する優位差を示したのは本研究が初となる．

8 あとがき

本研究では，既存のリファクタリング検出ツールが検出できないリファクタリングを検出することを目的として，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交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手法であるRefactoringMiner では検出できなかったリファクタリングパターンの検出に成功した．また，提案手法で検出したリファクタリングが行われたコミットのメッセージを解析して，これまでリファクタリングを含むコミットメッセージに頻出するとされていたパターンに含まれない単語が頻出していることが分かった．

今後の研究課題として，本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パターンの検出に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 2 つのリポジトリに対してのみ実験を実施したが，それ以外のリポジトリ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，それらの変更も提案手法で検出可能にすることが挙げられる．また，5.4 節においてリファクタリングを示唆する新たなパターンとして提案した単語が本当にリファクタリングを示唆する単語であるか，大規模な実験により検証する必要がある．

謝辞

輪講や中間報告といった研究活動の中で，困っている時に適切な助言をしていただいた，楠本真二教授に感謝の言葉を申し上げます．

日々の研究活動の中で生じる疑問や不安などに関して，お忙しい中でも個別に時間を設けて熱心かつ丁寧にご指導していただいた，肥後芳樹教授に深く感謝の言葉を申し上げます．本当にお世話になりました．

研究に関する新たな気づきを得る機会や，研究成果の伝え方の工夫など，今後の人生の糧になる貴重な助言を賜った，柗本真佑助教に感謝を申し上げます．

コロナが猛威を振るう中での研究室生活でしたが，事務補佐員の橋本美砂子氏の多大なる支援のおかげもあり，大事もなく過ごすことができました．誠にありがとうございます．

研究に関して困っている時に親身になって相談に乗っていただいた，コンピュータサイエンス専攻博士前期課程 2022 年卒業生の市川直人氏，同出田涼子氏，同荻野翔氏，同藤本章良氏，同前島葵氏に感謝を申し上げます．

研究活動や日々の悩みに関して相談に乗ってくれた，コンピュータサイエンス専攻博士前期課程 2 年の入山優氏，同髙市陸氏，同高木一真氏，同谷口真幸氏，同橋本周氏，同渡辺大登氏に心よりお礼申し上げます．共に研究に勤しんだ日々は私にとってかけがえのないものでした．

コンピュータサイエンス専攻博士前期課程 1 年の石野太一氏，同岩瀬匠氏，同小田郁弥氏，開地竜之介氏，同竹重拓輝氏，同吉岡遼氏，同王天豪氏，並びに基礎工学部情報科学科 4 年の皆森祐希氏，同久保光生氏，同馬渕航氏，同三原公平氏，同渡邉凌雅氏に感謝申し上げます．研究室で皆さんが頑張っている姿を見ていると，私も頑張ろうという気持ちになれました．

コンピュータサイエンス専攻博士前期課程 2 年ソフトウェア工学講座の鶴智秋氏には，研究に関する様々な悩みを相談しつつ有意義な議論をさせてもらいました．本当にありがとうございます．

最後に，遠い地から私の生活を心身ともに支えてくださった家族に心からお礼を申し上げます．

参考文献

[1] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *International Workshop on Refactoring*, pp. 51–58, 2019.

[2] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[3] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 5–18, 2012.

[4] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 132–146, 2013.

[5] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, Vol. 40, No. 7, pp. 633–649, 2014.

[6] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 858–870, 2016.

[7] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 97–106, 2002.

[8] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, Vol. 34, No. 3, pp. 321–335, 2008.

[9] Edmilson Campos Neto, Daniel Alencar da Costa, and Uir 叩 Kulesza. Revisiting and improving szz implementations. In *International Symposium on Empirical Software Engineering and Measurement*, pp. 1–12, 2019.

[10] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 751–754, 2014.

[11] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of Symposium and the European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[12] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, Vol. 48, No. 3, pp. 930–950, 2022.

[13] Danilo Silva, Jo 達 o Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, Vol. 47, No. 12, pp. 2786–2802, 2021.

[14] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pp. 201–211, 2014.

[15] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 121–130, 2013.

[16] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 643–653, 2014.

[17] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *Proceedings of Companion to the Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 815–816, 2007.

[18] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs. In *Proceedings of International Symposium on Software Testing and Analysis*, pp. 90–101, 2017.

[19] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. SUSHI: A Test Generator for Programs with Complex Structured Inputs. In *Proceedings of International Conference on Software Engineering: Companion Proceeedings*, pp. 21–24, 2018.

[20] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Commun. ACM*, Vol. 41, No. 5, pp. 81–86, may 1998.

[21] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, 1992.

[22] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pp. 371–372, 2010.

[23] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pp. 404–428, 2006.

[24] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer.

Comparison of similarity metrics for refactoring detection. In *Proceedings of the Working Conference on Mining Software Repositories*, pp. 53–62, 2011.

[25] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ACM International Conference on Automated Software Engineering*, pp. 163–172, 2011.

[26] Quinten David Soetens, Javier Perez, and Serge Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *IEEE International Conference on Software Maintenance*, pp. 384–387, 2013.

[27] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE Software*, Vol. 27, No. 04, pp. 52–57, 2010.