特別研究報告

題目

抽象構文木を利用した API に対する変更の検出

指導教員

楠本 真二 教授

報告者

入山 優

令和 3 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 2 年度 特別研究報告

抽象構文木を利用した API に対する変更の検出

入山 優

内容梗概

ライブラリがアップグレードされると，API も変更される．API の変更は様々で，それらを分類することはコードレビューやリリースノートの作成に役立つ．ライブラリから API の変更点を手動で調べるには負担が大きいため，その負担の軽減を目的とした API の変更の自動検出に関する研究が行われている．API の変更を自動で検出し，その種類ごとに分類するツールとして APIDiff がある．APIDiff は Java ライブラリの 2 つのバージョンを入力として受け取り，静的解析とコードの類似性に基づいて API の変更を検出して分類する．しかし，コードの類似性の閾値を適切に設定することは難しく，APIDiff はリファクタリングとして分類すべき API の変更を誤って分類する場合がある．そこで提案手法では，コードの類似性の閾値に依存せず抽象構文木を用いてリファクタリングを検出する RefactoringMiner を利用し，API の変更を分類する．8 個のオープンソースソフトウェアに対して実験を行った結果，既存手法と比べて API の変更をより高い精度で分類できることを確認した．また，既存手法と比較し新たに 8 種類のリファクタリングを検出可能であることを確認した．

主な用語

ソフトウェアリポジトリマイニング，API の進化，リファクタリング

目次

図目次

表目次

1　はじめに

多くのソフトウェアにおいてライブラリが活用されている [1]．ライブラリを活用することで開発者は無駄なプログラムを一から作る必要がなくなり．コードの記述量を減らすことができる．そのためプログラムの本質的な問題に集中することができ，生産性が向上する [2,3]．ライブラリは他のソフトウェアと同様に開発後もバグ修正や新機能の追加などメンテナンスされる [4]．Git などバージョン管理システムを活用し，ライブラリのバージョン管理が行われている．ライブラリは，アプリケーション・プログラミング・インターフェース（以下 API）を介して機能を提供している．ライブラリがアップグレードされると，API も変更される場合がある．API の変更は新機能の追加や不必要な機能の削除，保守性の向上を目的としたリファクタリングなど様々である．API の変更を分類することは，コードレビューやリリースノートの作成に役立つ [5,6]．リリースノートではどのような変更点があるのか，API の変更に対して API 利用者がどのような対処をとるべきかなど記載されている．API の変更を分類することで，その変更に応じた内容を記載することが可能である．例えば API の変更例として新たなメソッドの追加および既存メソッドのリネームがあげられる．新たなメソッドの追加に対して API 利用者が取るべき対処はない．しかし既存メソッドのリネームはそのメソッドを利用していた API 利用者のプログラム修正を必要とする変更であるため，API 利用者が取るべき対処をリリースノートに書く必要があることがわかる．

ライブラリから API の変更点を手動で調べるには負担が大きいため，その負担の軽減を目的とした API の変更の自動検出に関する研究が行われている．API の変更を自動で検出し分類するツールとして，APIDiff [7] が提案されている．APIDiff は Java ライブラリの 2 つのバージョンを入力として受け取ると，そのバージョン間で適用された API の変更操作のリストを出力する．APIDiff は RefDiff [8] と呼ばれるリファクタリング検出ツールを利用している．RefDiff は静的解析とコードの類似性に基づいて 2 つのバージョン間で適用されたリファクタリング操作のリストを出力する．APIDiff は静的解析を行うことでバージョンごとの API の要素（クラス，メソッド，フィールド）を検出し，変更前後の API 要素の比較結果と RefDiff によって得られるリファクタリング操作のリストをもとに，API の変更を検出してその種類ごとに分類する．

この APIDiff を利用してさまざまな研究が行われている．例えばライブラリの安定性 [9]，API の互換性が失われる変更がクライアントに与える影響 [9]，開発者が API の互換性が失われる変更を行なった理由 [10,11]，API の互換性が失われる変更の危険性に対する開発者の認識 [12] などを明らかにするための研究が行われている．

しかし，APIDiff には課題点がある．RefDiff においてコードの類似性の閾値を適切に設定することは難しく，APIDiff は API メソッドのパラメータリストの変更やフィールド名の変更といったリファ

クタリングとして分類すべき API の変更を API の削除および API の追加として誤って分類する場合がある．その結果，開発者や API の利用者が API の変更に対して誤った認識を持つ可能性がある．それによって API 利用者がその API の変更に対処するために冗長なプログラム修正や誤ったプログラム修正を行ない，生産性が下がってしまうおそれがある．

そこで本研究は，コードの類似性の閾値に依存しない方法を用いて API の変更をより高い精度で分類することを目的とする．APIDiff を改造し，RefDiff の代わりに RefactoringMiner [13] を用いて，API の変更を分類する手法を提案する．RefactoringMiner は Java ライブラリの 2 つのバージョンを入力として受け取ると，ステートメントマッピング情報と抽象構文木のノード置換に基づいてそのバージョン間で適用されたリファクタリング操作のリストを出力する．これによってコードの類似性の閾値に依存せずにリファクタリングを検出できる．提案手法では静的解析を行うことでバージョンごとの API の要素を検出し，変更前後の API 要素の比較結果と RefactoringMiner によって得られるリファクタリング操作のリストをもとに，API の変更を検出してその種類ごとに分類する．8 個のオープンソースソフトウェア（以下 OSS）に対して実験を行った結果，既存手法と比べて API の変更をより高い精度で分類できることを確認した．また，既存手法と比較し新たに 8 種類のリファクタリングを検出可能であることを確認した．

2 準備

2.1 API

API とはアプリケーション・プログラミング・インターフェースの略で，外部のライブラリを使用するためのインターフェースである．開発者は API として提供されているクラスやメソッド，フィールドを介して外部ライブラリの機能を利用することができる．すでに用意されている API を有効活用することで，開発者は無駄なプログラムを一から作る必要がなくなり．コードの記述量を減らすことができる．そのためプログラムの本質的な問題に集中することができ，生産性が向上する．

2.2 API 対する変更の種類

ライブラリはバグ修正や新機能の追加などアップグレードされる．それに伴って API もまた変更される場合がある．API に対する変更は新機能の追加や不必要な機能の削除， 保守性の向上を目的としたリファクタリングなど様々である．API に対する変更をいくつか紹介する（図 1）．

API の追加

新たに追加した機能を利用するための API としてクラスやメソッド，フィールドを追加する（図 1(a)）．

API の削除

既存の API のうち不必要になったクラスやメソッド，フィールドを削除する（図 1(b)）．

API のリファクタリング

保守性向上のため既存の API のクラスやメソッド，フィールドに対してリファクタリングを行う [14]（図 1(c)）．リファクタリングとは外部的振る舞いを変えることなく内部の構造を改善する行為である [15]．リファクタリングにはメソッドのリネームやパラメータリストの変更，クラスの移動やスーパークラスの抽出など様々な種類がある．

API の非推奨化

API のクラスやメソッドやフィールドを非推奨とする（図 1(d)）．ライブラリのアップグレードに伴い，より適切なクラスやメソッド，フィールドが新たに追加される．API 利用者が新しい API に移行するまで従来の API を維持するだけでなく，従来の API を利用してプログラミングを行わないよう伝える必要がある [16]．`@Deprecated` 注釈や`@deprecated Javadoc` タグを使用することで、クラスや

```
   public void setX(int i){ x = i; }
+  public void getX(){ return = x; }
```

(a) メソッドの追加

```
   public void setX(int i){ x = i; }
-  public void getX(){ return = x; }
```

(b) メソッドの削除

```
-  public void print(){
+  public void write(){
       System.out.println(“hello”);
   }
```

(c) メソッドのリネーム

```
+  @Deprecated
   public void setX(int i){ x = i; }
```

(d) メソッドの非推奨化

```
-  public void setX(int i){
+  private void setX(int i){
       x = i;
}
```

(e) メソッドの可視性修飾子の変更

図 1　API に対する変更の例

メソッド、フィールドを非推奨にし，従来の API を利用してプログラミングを行わないように API 利用者に警告をする．

API の可視性修飾子の変更

可視性修飾子を変更することで，クラスやメソッド，フィールドを API として外部に公開するのか，内部でのみ利用するのかを制御する（図 1(e)）．クラスやメソッド，フィールドの可視性修飾子が `public` または `protected` の場合，それらは API として外部に公開される．それに対してクラスやメソッド，フィールドの可視性修飾子が `default` または `private` の場合，それらは API として外部に公開されず，内部でのみ利用可能である．

2.3 RefDiff

RefDiff とはリファクタリングを検出するツールの 1 つである．Java ライブラリの 2 つのバージョンを入力として受け取り，静的解析とコードの類似性に基づいてそのバージョン間で適用されたリファクタリング操作のリストを出力する．このリストには API に対して行われたリファクタリング操作だけでなく，プライベートメソッドに対するリファクタリングのように API 以外のクラスやメソッド，フィールドに対して行われたリファクタリング操作も含まれている．コードの類似性の閾値は，10 個の OSS からランダムに選択された 10 個のコミットに対する実験結果をもとに決定されている．

2.4 APIDiff の検出方法

APIDiff は Java ライブラリの 2 つのバージョンを入力として受け取り，静的解析と RefDiff を用いてそのバージョン間で API に対して行われた変更操作のリストを出力する．

API の変更の検出は次の 5 つのステップで構成される．その概要を図 2 に示す．

Step 1 2 つのバージョンのライブラリインスタンスを作成
Step 2 バージョン間で行われたリファクタリングの検出
Step 3 変更前後の API 要素の対応付け
Step 4 API の要素とリファクタリング操作の対応付け
Step 5 API の変更操作のリストを作成

Step 1 では入力として受け取った Java ライブラリの 2 つのバージョンに対して静的解析を行い，バージョンごとのライブラリのインスタンスを作成する．このインスタンスは可視性修飾子が `public` または `protected` で外部に公開されているクラス，メソッド，フィールドといった API の要素の情報を保持している．

Step 2 では APIDiff の入力として与えられた Java ライブラリの 2 つのバージョンを RefDiff の入力として与え，そのバージョン間で適用されたリファクタリング操作のリストを得る．

Step 3 では Step 1 で得られた変更前後のライブラリインスタンスを比較し，変更前後の API 要素の対応付けを行う．クラスについては完全限定名が一致するクラスを，メソッドについてはそのメソッドが所属するクラスの完全限定名とメソッド名，パラメータの並びの 3 種類全てが一致するメソッドを，フィールドについてはそのフィールドが所属するクラスの完全限定名とフィールド名が一致するフィールドを対応付ける．

Step 4 では，Step 3 で対応付けることができなかった API の要素と Step 2 で得られたリファクタリング操作の対応付けを行う．対応付けられなかった API の要素がリファクタリング操作のリストに含まれていれば，リファクタリングされた要素，そうでなければ削除された要素または追加された要素

として分類する．

Step 5 では，Step 3 と Step 4 で行われた API 要素の対応付けの結果と API 要素の情報を参照し，どの API が削除・追加・リファクタリングされたのか，どの API が非推奨化されたのか、どの API の可視性修飾子が変更されたのか，どのフィールドが型を変更されたのか，どのメソッドが戻り値の型を変更されたのか，といった情報を得る．それらの情報をもとに API の変更操作のリストを作成する．

図 2 において，Step 3 で対応付けができなかった API の要素は変更前のメソッド b とメソッド c，変更後のメソッド B とメソッド e である．RefDiff によって得られるリストよりメソッド b がメソッド B にリネームされたことがわかるので，メソッド b とメソッド B が対応付けられ，それ以外のメソッド c は削除された API の要素，メソッド e は追加された API の要素だと判断される．

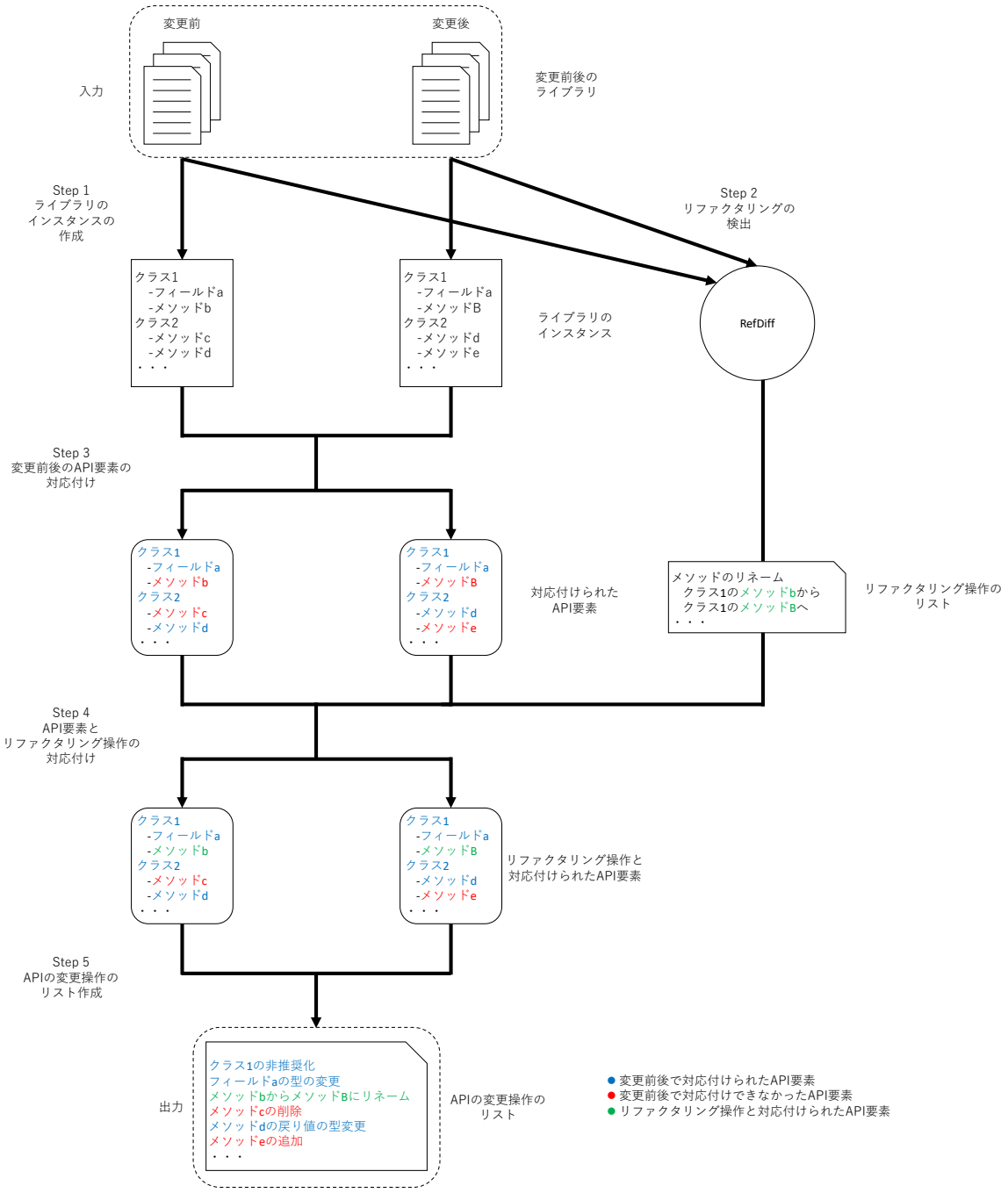
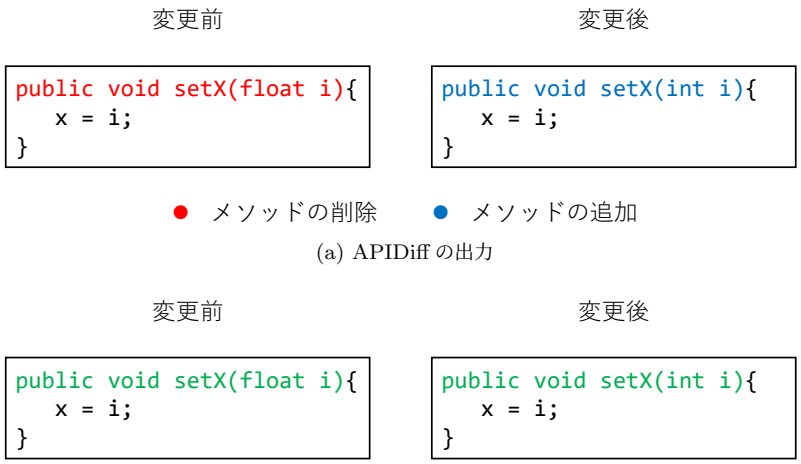


図 2　APIDiff の概要



(a) APIDiff の出力

(b) 実際の API の変更操作

図 3　実際の API の変更操作と APIDiff の出力が異なる例

3　研究目的

APIDiff は RefDiff によって得られるリファクタリング操作のリストをもとに API の要素をリファクタリングされた要素，削除された要素，追加された要素に分類している．しかしコードの類似性の閾値を適切に設定することは難しく，RefDiff で検出できないリファクタリングが存在する．そのため RefDiff が API に対するリファクタリングを検出できなかった場合は，そのリファクタリングされた API の要素を削除された要素または追加された要素として分類してしまう．

図 3 に実際の API の変更操作と APIDiff の出力が異なる例を示す．変更前後で APIDiff を用いて得られる変更操作は，変更前のメソッド `setX(float i)` の削除，変更後のメソッド `setX(int i)` の追加となる (図 3(a))．しかし，実際に行われた API の変更操作はメソッド `setX` のパラメータを `float` から `int` に変更するリファクタリング操作である（図 3(b)）．この違いは，RefDiff の類似性の閾値が高く設定されていることが原因であり，APIDiff は API のパラメータの変更として検出することができなかった．`setX(float i)` を利用していた API 利用者はパラメータの型を変更するだけで，その API を利用し続けることができるのにもかかわらず，`setX(float i)` が削除されたため同様の機能を持つ API を探す必要があると誤った認識を持ち，生産性が下がってしまうおそれがある．

そこで，本研究では類似性の閾値に依存しない方法で API の変更をより高い精度で分類する手法を提案する．

4 提案手法

提案手法では，ステートメントマッピング情報と抽象構文木のノード置換に基づいてリファクタリングを検出する RefactoringMiner を利用し，API の変更を分類する．これによってコードの類似性の閾値に依存せず，API の変更を分類することが可能である．

4.1 RefactoringMiner

RefactoringMiner は Java ライブラリの 2 つのバージョンを入力として受け取ると，メソッド本体のステートメントに関するマッピング情報と抽象構文木のノード置換に基づいてそのバージョン間で適用されたリファクタリング操作のリストを出力する．これによってコードの類似性の閾値に依存せずにリファクタリングを検出できる．ステートメントマッピングは，メソッドに含まれる各文を変更前後で対応付けた情報である．抽象構文木はソースコードを構文解析して得られる木構造のデータである．リファクタリング操作のリストには API に対して行われたリファクタリング操作だけでなく，API 以外のクラスやメソッド，フィールドに対して行われたリファクタリング操作も含まれている．

4.2 API の変更の分類方法

既存手法と同様に提案手法における API の変更の分類は次の 5 つのステップで構成される．その概要を図 4 に示す．

Step 1 2 つのバージョンのライブラリインスタンスを作成
Step 2 バージョン間で行われたリファクタリングの検出
Step 3 変更前後の API 要素の対応付け
Step 4 API の要素とリファクタリング操作の対応付け
Step 5 API の変更操作のリストを作成

Step 1 は既存手法の Step 1 の処理を再利用した．

Step 2 では APIDiff の入力として与えられた Java ライブラリの 2 つのバージョンを RefactoringMiner の入力として与え，バージョン間で適用されたリファクタリング操作のリストを得る．

Step 3 では既存手法と同様に変更前後の API 要素の対応付けを行う．ただしメソッドについてはそのメソッドが所属するクラスの完全限定名とメソッド名，パラメータの並びに戻り値の型を加えた 4 種類全てが一致するメソッドを，フィールドについてはそのフィールドが所属するクラスの完全限定名とフィールド名にフィールドの型を加えた 3 種類全てが一致するフィールドを対応付けるように変更した．これは既存手法のように変更前後で対応付けた API 要素の情報を比較することでフィールドの型

の変更やメソッドの戻り値の型変更を検出するのではなく，RefactoringMiner によってフィールドの型の変更やメソッドの戻り値の型変更を検出し，そのリファクタリング操作と API 要素と対応付けるためである．これによって変更前後で API 要素が所属するクラスの完全限定名や API 要素の名前などが一致しない場合でも，フィールドの型の変更やメソッドの戻り値の型変更を検出できるようになった．

Step 4，Step 5 では既存手法と同様の方法で API の要素とリファクタリング操作の対応付けと API の変更操作のリスト作成を行う．

図 4 の例において，Step 3 で対応付けができなかった API の要素は変更前のフィールド a とメソッド b，メソッド c，メソッド d と変更後のフィールド a とメソッド B，メソッド d，メソッド e である．RefactoringMiner によって得られるリストよりフィールド a の型が変更されたこと，メソッド b がメソッド B にリネームされたこと，メソッド d の戻り値の型が変更されたことががわかるので，変更前後のフィールド a，メソッド b とメソッド B，変更前後のメソッド d が対応付けられる．それ以外のメソッド c は削除された API の要素，メソッド e は追加された API の要素だと判断される．

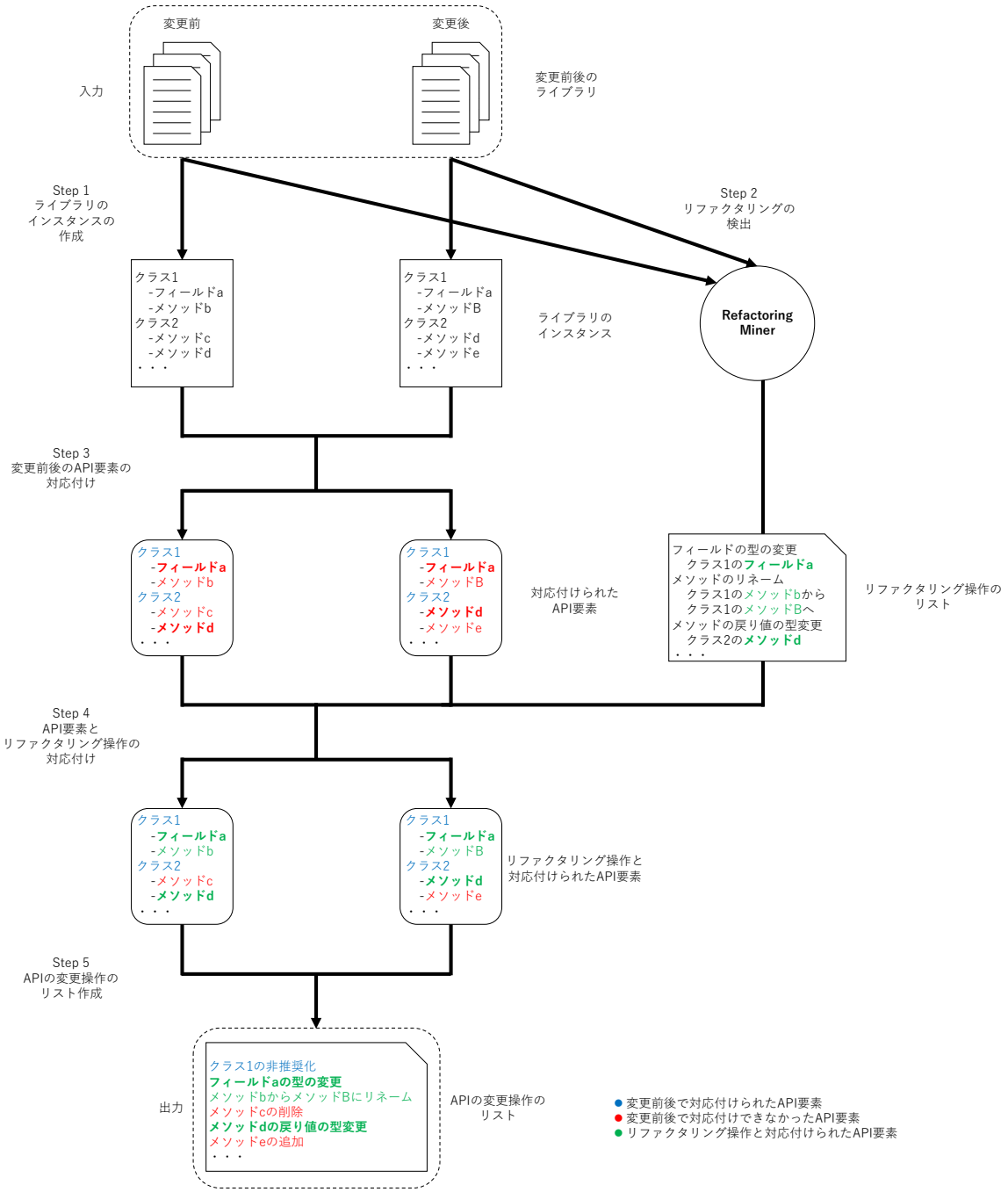

変更前
変更後
入力
変更前後の
ライブラリ
Step 1
ライブラリの
インスタンスの
作成
Step 2
リファクタリングの
検出
クラス1
-フィールドa
-メソッドb
クラス2
-メソッドc
-メソッドd
・・・
クラス1
-フィールドa
-メソッドB
クラス2
-メソッドd
-メソッドe
・・・
ライブラリの
インスタンス
Refactoring
Miner
Step 3
変更前後のAPI要素の
対応付け
クラス1
-フィールドa
-メソッドb
クラス2
-メソッドc
-メソッドd
・・・
クラス1
-フィールドa
-メソッドB
クラス2
-メソッドd
-メソッドe
・・・
対応付けられた
API要素
フィールドの型の変更
クラス1のフィールドa
メソッドのリネーム
クラス1のメソッドbから
クラス1のメソッドBへ
メソッドの戻り値の型変更
クラス2のメソッドd
・・・
リファクタリング操作の
リスト
Step 4
API要素と
リファクタリング操作の
対応付け
クラス1
-フィールドa
-メソッドb
クラス2
-メソッドc
-メソッドd
・・・
クラス1
-フィールドa
-メソッドB
クラス2
-メソッドd
-メソッドe
・・・
リファクタリング操作と
対応付けられたAPI要素
Step 5
APIの変更操作の
リスト作成
出力
クラス1の非推奨化
フィールドaの型の変更
メソッドbからメソッドBにリネーム
メソッドcの削除
メソッドdの戻り値の型変更
メソッドeの追加
・・・
APIの変更操作の
リスト
変更前後で対応付けられたAPI要素
変更前後で対応付けできなかったAPI要素
リファクタリング操作と対応付けられたAPI要素

図 4　提案手法の概要

5 実験

本章では，提案手法を用いて行なった実験とその結果について述べる．

5.1 評価項目

提案手法を評価するために 3 つの評価項目を設定した．

リファクタリングの検出数

提案手法と既存手法からそれぞれ得られる検出結果を比較し，API に対して行われたリファクタリングの検出数にどのような変化が見られるのかを調べる．またリファクタリングの種類別に検出結果を比較し，どのような変化が見られるかを調べる．

分類精度

提案手法によって API の変更をより高い精度で分類できるかどうかを検証する．提案手法によって改善が期待される変更は API に対するリファクタリングであるため，検出されたリファクタリングが実際に正しいのかを確認する．

実行時間

提案手法の実行時間と既存手法の実行時間を比較し，提案手法の処理速度を評価する．またどの処理に時間を要するのかを確認する．

5.2 実験対象

関連研究 [17] で用いられている OSS のうち，コミット数が 20,000 以下でスター数が多い OSS から順に 8 個の OSS を実験対象とした．実験対象の OSS を表 1 に示す．コミット数に制限を設けた理由は，大規模な OSS に対して提案手法，既存手法のどちらを適用しても実行時間が長くなり評価が難しいためである．リファクタリング別の検出数の比較と分類精度の比較では 8 個の OSS のうち MPAndroidChart のみを対象とした．これは APIDiff の適用事例でも利用されていた OSS である．

5.3 既存手法とのリファクタリングの検出数の比較

5.3.1 全リポジトリに対しての検出数の比較

実験対象の 8 個の OSS の master ブランチ[*1]上に存在する全てのコミットに対して提案手法と既存手法を適用した．提案手法と既存手法からそれぞれ得られる検出結果を比較し，API に対して行われたリファクタリングの検出数にどのような変化が見られるのかを調べた．その結果が表 3 である．既存手法では実験対象のリポジトリから 1,985（=1,626+359）個の API に対するリファクタリングしか検出できなかったのに対して，提案手法では 4,581（=1,626+2,955）個の API に対するリファクタリングを検出できた．すべてのプロジェクトにおいて提案手法が検出した API に対するリファクタリングの数が，既存手法が検出した API に対するリファクタリングの数を上回った．また提案手法のみが検出した API に対するリファクタリングの数は，提案手法または既存手法によって検出された API に対するリファクタリングの総数の 31.3%〜76.6% を占めており，その平均は 59.8% であった．それに対して既存手法のみが検出した API に対するリファクタリングの数は，提案手法または既存手法によって検出された API に対するリファクタリングの総数の 3.8%〜17.3% を占めており，その平均は 7.3% であった．

5.3.2 リファクタリング別の検出数の比較

8 個の OSS の中から MPAndoridChart を選択し，提案手法と既存手法からそれぞれ得られる検出結果をリファクタリングの種類別に比較した．その結果が表 3 である．提案手法では既存手法で検出され

表 1　実験対象の OSS

プロジェクト名	LOC	コミット数	対象の最終コミット日
OkHttp	72,061	4,779	2021 年 1 月 9 日
Retrofit	26,995	1,862	2020 年 12 月 10 日
MPAndroidChart	25,232	2,068	2020 年 10 月 30 日
LeakCanary	26,207	1,554	2021 年 1 月 6 日
Hystrix	50,510	2,108	2018 年 11 月 20 日
iosched	23,550	2,757	2020 年 6 月 26 日
Fresco	98,427	2,768	2021 年 1 月 12 日
Logger	1,441	144	2018 年 4 月 10 日

[*1] LeakCanary についてはデフォルトのブランチが main ブランチであるため，main ブランチに対して実験を行なった．

た 14 種類のリファクタリングに加えて，Change in Parameter List や Rename Field など新たに 8 種類のリファクタリングを検出した．また両手法で検出された 14 種類のリファクタリングについて提案手法は既存手法以上の数のリファクタリングを検出した．

5.4 既存手法との分類精度の比較

8 個の OSS の中から MPAndoridChart を選択し，提案手法と既存手法からそれぞれ得られる検出結果を目視確認することで分類の精度の比較を行なった．提案手法のみで検出された API に対するリファクタリングと既存手法のみで検出された API に対するリファクタリングを目視確認した．提案手法のみで検出された API に対するリファクタリングの数が多かったため，許容誤差 5%，信頼度 95% となるように 289 個サンプリングを行った．またリファクタリングの種類ごとに可能な限り均等になるようにサンプリングを行なった．既存手法のみで検出された API に対するリファクタリングは全て目視確認を行った．目視確認を行なった結果が表 3 の通りである．Change in Return Type Method と Move Method については既存手法の適合率が提案手法の適合率より高くなったが，全体としては既存手法の適合率が 79.5% であったのに対して提案手法の適合率が 88.6% と高い結果となった．

5.5 既存手法との実行時間の比較

実験対象の 8 個の OSS の master ブランチ上に存在する全てのコミットに対して提案手法と既存手法を適用し，実行時間を比較した．実験に用いた計算機の性能を表 4 に示す．実験結果を図 5 に示す．図 5 においてチェックアウトはライブラリを別のバージョンに切り替える処理を表す．その他には対象の OSS がローカルに存在するのかを確認する処理やコミットグラフをたどって一致するコミットを順番に生成する処理などが含まれる．

表 2　API に対するリファクタリングの検出数

プロジェクト名	両手法が検出したリファクタリングの数	提案手法のみが検出したリファクタリングの数	既存手法のみが検出したリファクタリングの数	合計
OkHttp	286（51.4%）	174（31.3%）	96（17.3%）	556
Retrofit	154（48.7%）	135（42.7%）	27（ 8.5%）	316
MPAndroidChart	709（36.5%）	1,161（59.8%）	73（ 3.8%）	1,943
LeakCanary	15（16.1%）	66（71.0%）	12（12.9%）	93
Hystrix	128（24.4%）	342（65.1%）	55（10.5%）	525
iosched	91（32.7%）	143（51.4%）	44（15.8%）	278
Fresco	226（19.2%）	901（76.6%）	50（ 4.2%）	1,177
Logger	17（32.7%）	33（63.5%）	2（ 3.8%）	52
合計	1,626（32.9%）	2,955（59.8%）	359（ 7.3%）	4,940

Fresco を除く 7 個の OSS において提案手法の実行時間は既存手法の実行時間より長くなった．主な原因はリファクタリングの検出時間が長くなったためである．また提案手法は既存手法と比較してチェックアウトやライブラリのインスタンスの作成に要する時間も長くなった．その原因を調べるためにどのメソッドの実行時間が増加したのかを調査した結果，APIDiff が外部ライブラリとして使用している JGit と JDT のメソッドの実行時間が増加していた．提案手法では RefactoringMiner を利用するために JGit と JDT のバージョンをアップグレードした．それによって JGit と JDT の API の処理が変更され，その API を利用していたチェックアウトやライブラリのインスタンスの作成の実行時間が増加した．

表 3　リファクタリング別の検出数と適合率

	両手法が検出したリファクタリング	提案手法のみが検出したリファクタリング			既存手法のみが検出したリファクタリング		
リファクタリングの種類	検出数	検出数	サンプルサイズ	適合率（%）	検出数	サンプルサイズ	適合率（%）
Pull Up Method	114	93	19	100	21	21	100
Rename Method	147	51	19	78.9	22	22	66.2
Change in Return Type Method	125	42	19	94.7	3	3	100
Move Method	59	39	19	15.8	13	13	61.5
Move Field	45	18	18	100	1	1	100
Push Down Method	27	24	19	100	3	3	100
Inline Method	6	33	19	100	3	3	100
Rename Type	27	2	2	100	2	2	100
Push Down Field	6	2	2	100	1	1	100
Extract Method	0	128	20	85.0	4	4	25.0
Move Type	69	6	6	83.3	0	0	
Change in Field Type	53	8	8	100	0	0	
Pull Up Field	28	19	19	100	0	0	
Move and Rename Type	3	2	2	50.0	0	0	
Change in Parameter List	0	558	20	100	0	0	
Rename Field	0	48	19	100	0	0	
Extract Supertype	0	36	19	94.7	0	0	
Move and Rename Method	0	31	19	73.7	0	0	
Extract Type	0	13	13	92.3	0	0	
Extract Field	0	3	3	100	0	0	
Move and Rename Field	0	3	3	100	0	0	
Extract Subtype	0	2	2	100	0	0	
平均	709	1,161	289	88.6	73	73	79.5

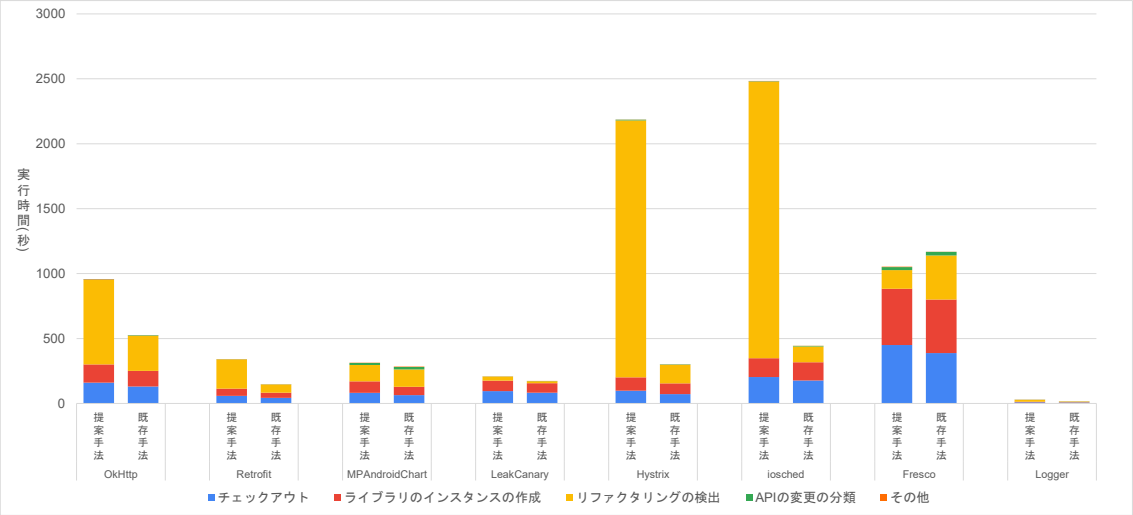


図 5　API に対する変更の検出に要する時間

6　考察

表 3 において提案手法の適合率より既存手法の適合率が高い Change in Return Type Method および Move Method について考察する．また Extract Supertype および Extract Method について，検出数に違いが見られた要因をコードの類似性の閾値以外の観点から考察する．Extract Supertype は既存手法でも仕様上検出可能とされている種類だが，実際には検出されなかった変更である．Extract Method は両手法で検出可能なリファクタリングの種類のうち両手法が検出した数が 0 であった変更である．

6.1　Change in Return Type Method

表 3 において既存手法のみが検出したリファクタリングは 3 個のみであったのに対して提案手法のみが検出した数は 42 個となっており，提案手法の方がより多くのリファクタリングを検出している．また提案手法の適合率は 94.7% であり，既存手法の適合率 100% と比べると低いがその差は小さい．

表 4　実験に用いた計算機の性能

OS	macOS Big Sur
CPU	Intel Core i5（1.4GHz，4 コア）
GPU	Intel Iris Plus Graphics 645
メモリ	16GB

変更前

```
protected int getXIndex(float x) {
    // create an array of the touch-point
    float[] pts = new float[2];
    pts[0] = x;
    // take any transformer to determine the x-axis value
    mChart.getTransformer(YAxis.AxisDependency.LEFT).pixelsToValue(pts);
    return (int) Math.round(pts[0]);
}
```

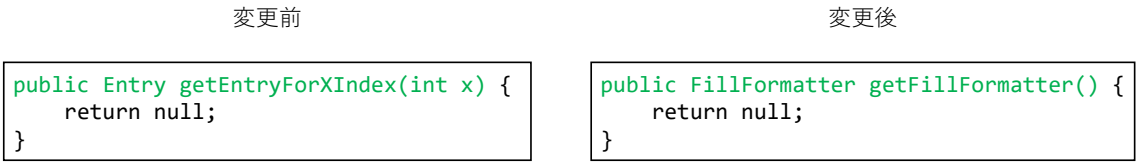
変更後

```
protected float getXForTouch(float x) {
    // create an array of the touch-point
    float[] pts = new float[2];
    pts[0] = x;
    // take any transformer to determine the x-axis value
    mChart.getTransformer(YAxis.AxisDependency.LEFT).pixelsToValue(pts);
    return Math.round(pts[0]);
}
```

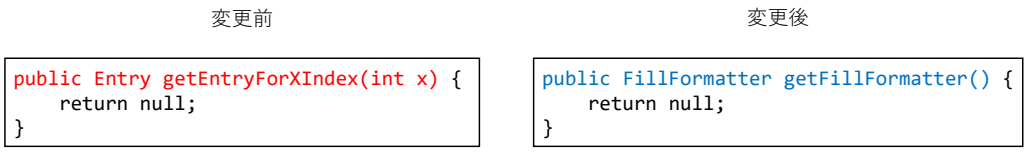
図 6　提案手法のみが検出可能な Change in Return Type Method の例

既存手法では戻り値の型に加えてパラメータリストおよびメソッド名が変更された場合，Change in Return Type Method を検出できなかった．しかし提案手法ではそのような場合でも Change in Return Type Method を検出できた．その具体例を図 6 に示す．この例ではメソッドの戻り値の型が `int` 型から `float` 型に変更されると同時に，メソッド名も `getXIndex` から `getXForTouch` に変更されている．既存手法ではこの Change in Return Type Method を検出できなかったが，提案手法では検出できた．

提案手法がメソッドの削除およびメソッドの追加を Change in Return Type Method として誤って検出した実際の例を図 7 に示す．変更前後で提案手法を用いて得られる変更操作は，メソッドの戻り値の型変更となる (図 7(a))．しかし実際に行われた API の変更操作は `getEntryForXIndex(int x)` の削除および `getFillFormatter()` の追加である（図 7(b)）．この違いは，メソッド本体の `return` 文だけではメソッド `getEntryForXIndex(int x)` と `getFillFormatter()` が異なるメソッドであると RefactoringMiner が判別できなかったことが原因である．これを解決するためにはメソッド本体の文が一致するかどうかだけでなく，一致する文がいくつあるのかも考慮する．



(a) 提案手法の出力



(b) 実際の API の変更操作

図 7　提案手法が Change in Return Type として誤って検出した例

6.2　Move Method

表 3 において提案手法の検出数は既存手法の検出数より多いが，適合率が 15.8% と低い値となっている．これは，RefactoringMiner が一部の Pull Up Method と Push Down Method を Move Method として出力してしまうためである．提案手法では RefactoringMiner の検出結果を利用しているため，本来 Pull Up Method や Push Down Method として分類すべき変更を Move Method として誤って分類してしまう．これを解決するためには RefactoringMiner による誤検出を改善する必要がある．

6.3　Extract Supertype および Extract Method

表 3 において Extract Supertype および Extract Method について検出数に違いが見られた．これは提案手法と既存手法で API のクラスおよびメソッドとリファクタリング操作の対応付けの実装が異なるためである．API のクラスおよびメソッドは可視性修飾子が `public` または `protected` であり外部に公開されているクラスおよびメソッドである．

まず Extract Supertype について述べる．Extract Supertype とはクラスの一部を切り出してスーパークラスを作成する変更である．以降では Extract Supertype によって作成されたスーパークラスを抽出されたスーパークラス，一部を切り出されるクラスを抽出元のクラスと呼称する．両手法においてまず変更前後でクラスの完全限定名が一致する API のクラスを対応付ける．対応付けることができなかった API のクラスが RefDiff または RefactoringMiner によって得られるリファクタリング操作のリストに含まれていれば，リファクタリングされたクラス，そうでなければ削除されたクラスまたは追加されたクラスとして分類する．変更前後で対応付けることができなかった API のクラスと Extract

Supertype の対応付けについて提案手法では，対応付けることができなかった変更後の API のクラスが抽出されたスーパークラスなのかを調べる．それに対して既存手法では対応付けることができなかった変更前の API のクラスが抽出元のクラスなのかを調べる．提案手法で検出された Extract Supertype を確認したところ，抽出元のクラスの完全限定名は変更されていなかった．変更前の抽出元のクラスと変更後の抽出元のクラスが対応付けられるため，既存手法ではリファクタリング操作との対応付けが行われず，Extract Supertype の検出数が 0 となった．

次は Extract Method について述べる．Extract Method とはメソッドの一部を切り出して別のメソッドにする変更である．以降では Extract Method によって作成されたメソッドを抽出されたメソッド，一部を切り出されるメソッドを抽出元のメソッドと呼称する．変更前後で対応付けることができなかった API のメソッドと Extract Method の対応付けについて提案手法では，対応付けることができなかった変更後の API のメソッドが抽出されたメソッドなのかを調べる．それに対して既存手法では対応付けることができなかった変更前の API のメソッドが抽出元のメソッドなのかを調べる．提案手法または既存手法で検出された Extract Mehod の大部分で，変更前の抽出元のメソッドと変更後の抽出元のメソッドが対応付けられるため，既存手法ではリファクタリング操作との対応付けが行われず，Extract Method の検出数が少なくなった．

7　妥当性の脅威

提案手法の分類の精度を評価するために目視確認を行なった．しかしこの結果は著者の主観に依存しており，リファクタリングでないにも関わらずリファクタリングと判断したり，リファクタリングであるにも関わらずリファクタリングでないと判断したりしている可能性がある．

検出数が少ないリファクタリングについて十分な数を目視確認することができなかったため，適合率を適切に求められていない可能性がある．

8　おわりに

本研究では，コードの類似性の閾値に依存せず抽象構文木を用いてリファクタリングを検出する RefactoringMiner を利用し，API の変更を分類する手法を提案した．提案手法を用いて 8 個の OSS に対して実験を行ったところ，既存手法と比べて API の変更をより高い精度で分類できることを確認した．また，既存手法と比較し新たに 8 種類のリファクタリングを検出可能であることを確認した．

今後の課題としては次のようなものが考えられる．

サンプル数の増加　検出数が少ないリファクタリングについて十分な数を目視確認していない，そこで対象の OSS を増やしサンプル数を増加することで，より適切な適合率を求めることが考えられる．

実行時間の短縮　OSS によっては提案手法の実行時間が既存手法の実行時間の数倍になってしまった場合があるので，リファクタリングの検出時間が長くなった原因や短縮可能な処理を調査し，全体の実行時間を短縮できるようにツールを改良することが考えられる．

リファクタリング以外による変更の適合率の算出　本研究では提案手法により改善が期待されるリファクタリングの適合率のみを比較した．リファクタリング以外の変更については両手法ともに検出された変更が正しいのかを確認していない．そのためリファクタリング以外による変更に対しても適合率を求め，もし適合率が低ければその原因を調査する．

謝辞

本論文の執筆にあたり，多くの方々にお世話になりました。ここに記して感謝を申し上げます。

本研究を行うにあたって数々のご助言と有益なご意見を賜りました楠本真二教授に心より感謝申し上げます.

本研究に対する有益な御助言および発表に対するご指導等を賜りました柗本真佑助教に深く感謝申し上げます。

本研究において，研究の方針や実験方法，論文執筆など全過程を通して終始適切なご指導およびご助言を頂きました肥後芳樹准教授に深く感謝申し上げます.

本研究において，論文全体のストーリー作りやその細部まで丁寧なご助言を賜りました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博士前期課程 1 年の藤本章良氏，同前島葵氏に深く感謝申し上げます.

研究生活において，様々な場面で支えて頂きました楠本研究室の皆様に心より感謝申し上げます.

参考文献

[1] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining Trends of Library Usage. In *the proceedings of Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops*, pp. 57–62, 2009.

[2] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, Vol. 29, No. 9, pp. 45–51, 1996.

[3] A. Michail. Data mining library reuse patterns in user-selected applications. In *the proceedings of IEEE International Conference on Automated Software Engineering*, pp. 24–33, 1999.

[4] Chow and Notkin. Semi-automatic update of applications in response to library changes. In *the proceedings of International Conference on Software Maintenance*, pp. 359–368, 1996.

[5] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora. ARENA: An Approach for the Automated Generation of Release Notes. *IEEE Transactions on Software Engineering*, Vol. 43, No. 2, pp. 106–127, 2017.

[6] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *the proceedings of International Conference on Software Engineering*, pp. 47–57, 2012.

[7] A. Brito, L. Xavier, A. Hora, and M. T. Valente. APIDiff: Detecting API breaking changes. In *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 507–511, 2018.

[8] D. Silva and M. T. Valente. RefDiff: Detecting Refactorings in Version Histories. In *the proceedings of IEEE/ACM International Conference on Mining Software Repositories*, pp. 269–279, 2017.

[9] L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 138–147, 2017.

[10] A. Brito, L. Xavier, A. Hora, and M. T. Valente. Why and how Java developers break APIs. In *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 255–265, 2018.

[11] A. Brito, M. T. Valente, L. Xavier, and A. Hora. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, pp. 1458–1492,

2020.

[12] L. Xavier, A. Hora, and M. T. Valente. Why do we break APIs? First answers from developers. In *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 392–396, 2017.

[13] N. Tsantalis, A. Ketkar, and D. Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, pp. 1–21, 2020.

[14] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *the proceedings of International Conference on Software Engineering*, pp. 151–160, 2011.

[15] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[16] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. L. Traon. API Document Quality for Resolving Deprecated APIs. In *the proceedings of Asia-Pacific Software Engineering Conference*, Vol. 2, pp. 27–30, 2014.

[17] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *the proceedings of Joint Meeting on Foundations of Software Engineering*, pp. 465–475, 2017.