抽象構文木を利用したAPIの後方互換性が破壊される変更の検出

入山 優[1,a)]　肥後 芳樹[1,b)]　楠本 真二[1,c)]

概要：ライブラリが更新されると，API も変更される．API の変更は様々で，後方互換性を壊す変更と後方互換性を維持する変更に分類できる．API の変更を検出し，その変更によって API の後方互換性が維持されているかを判別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，後方互換性を壊す変更と後方互換性を維持する変更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，コードの類似度に基づいて API の変更を検出し，各変更を後方互換性を壊す変更と後方互換性を維持する変更に分類する．しかし，コードの類似度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．また検出した変更を変更の種類ごとに後方互換性を壊す変更なのかを判別しているため，ある API に対して複数の変更が行われるとそれらの変更によって API の後方互換性が維持されているかを正しく判別できない場合がある．そこで提案手法では，コードの類似度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 の変更を検出する．そして API ごとに変更をグループ化して後方互換性を壊す変更が含まれているかを調べることで，それらの変更によって API の後方互換性が維持されているかを判別する．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検出できることを確認した．また 1 つの API に対して後方互換性を維持する変更と後方互換性を壊す変更が行われた場合でも，それらの変更によって API の後方互換性が維持されているかを判別できることを確認した．

1. はじめに

多くのソフトウェアにおいて生産性の向上のためライブラリが活用されている [1, 2]．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．ライブラリが更新されると，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 [3]．これらの変更は後方互換性を壊す変更と後方互換性を維持する変更に分類が可能である [3]．API の変更を検出し，その変更によって API の後方互換性が維持されているかを判別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 [4]．

API の変更を手動で検出すること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 [5] 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 [6] 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 はコードの類似度に基づいて 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 はバージョンごとの API を検出し，変更前後の API 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 の変更を検出して後方互換性を壊す変更と後方互換性を維持する変更に分類する．

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 [7]，API の後方互換性を壊す変更がクライアントに与える影響 [7]，開発者が API の後方互換性を壊す変更を行なった理由 [8]，API の後方互換性を壊す変更の危険性に対する開発者の認識 [9] など

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] m-iriyam@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

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度の閾値を適切に設定することは難しく，APIDiff は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．また検出した変更を変更の種類ごとに後方互換性を壊す変更なのかを判別しているため，ある API に対して複数の変更が行われるとそれらの変更によって API の後方互換性が維持されているかを正しく判別できない場合がある．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．

そこで本研究は，コードの類似度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検出すること，そしてある API に対して複数の変更が行われた場合でもそれらの変更によって API の後方互換性が維持されているかどうかを正しく判別することを目的とする．RefDiff ではなく RefactoringMiner [10] を用いて，API の変更を検出し，API ごとに変更をグループ化することでそれらの変更によって後方互換性が維持されているかどうかを判別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，抽象構文木（以下 AST）を生成して文単位でマッチングを行い，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．コードの類似度を計算し閾値より大きいかどうかで一部が異なる文をマッチングさせるのではなく，3 段階で構成されるマッチングを行うことで，一部が異なる文でもマッチングさせる．また AST の対応づけにおいて類似度を計算したり閾値と比較したりせずに，マッチングに成功した文の数と失敗した文の数の比較を行う．そのためコードの類似度の閾値に依存せずにリファクタリングを検出できる．提案手法では変更前後の API 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 の変更を検出する．そして API ごとに変更をグループ化して後方互換性を壊す変更が含まれているかを調べることで，それらの変更によって後方互換性が維持されているかを判別する．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検出できることを確認した．また 1 つの API に対して後方互換性を維持する変更と後方互換性を壊す変更が行われた場合でも，それらの変更によって API の後方互換性が維持されているかを判別できることを確認した．

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称で，外部ライブラリが持つ機能を使用するためのインターフェースである．本研究では Java における，アクセスレベルが `public` または `protected` なクラス，メソッド，フィールドを API とみなす．API を利用する際に考慮すべき情報としてクラス名やメソッド名，シグネチャ，戻り値の型，修飾子，親クラス，例外などがあげられる．本研究ではこれの情報を含めて API と見なす．メソッドの本体の文がどのように実装されているか，どのようなローカル変数を利用しているのかといった情報は API 利用時に考慮する必要がないため，これらの情報は API として含めない．

2.2 API の変更

本研究ではクラス名やシグネチャ，戻り値の型，修飾子など API を利用する際に考慮すべき情報の変更に加えて新しい API の追加や既存 API の削除を API の変更とみなす．バグ修正によるメソッド本体の文の変更やローカル変数の変数名の変更などは API 利用時に考慮する必要がないため API の変更として扱わない．

2.3 API の後方互換性

本研究ではライブラリを更新したときに呼び出し側のコードの変更が一切不要である場合，新しいバージョンの API はその前のバージョンの API と後方互換性を持つとし，コードの変更が必要な場合は後方互換性を持たないとする．`public` から `protected` へメソッドのアクセスレベルが変更されたとき，そのメソッドが所属するクラスを継承したクラスからメソッドを利用していた場合はコードの変更が不要だが，そうでない場合はコードの変更が必要である．このように呼び出し側の実装によってコードの変更が必要かどうかが変わる場合，新しいバージョンの API はその前のバージョンの API と後方互換性を持たないとする．

2.4 RefDiff

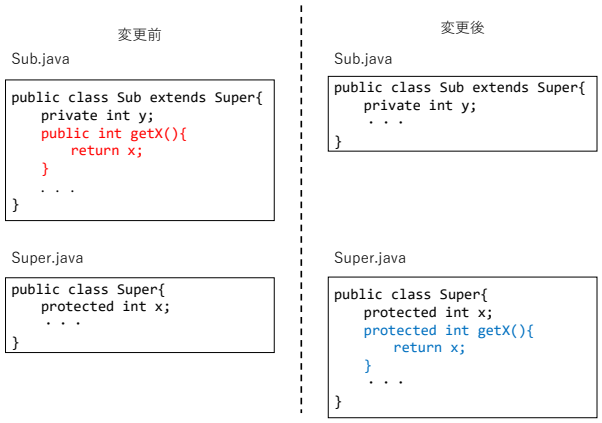
RefDiff とはリファクタリングを検出するツールの 1 つである [6]．Java ライブラリの 2 つのバージョンを入力として受け取ると，コードをトークンに分解し，出現頻度が低いトークンの重みが大きくなるように各トークンの重みを設定する．トークンごとに変更前の重みと変更後の重みを比較し，2 つのうち大きい重みの合計に対する小さい重みの合計の比を類似度として算出する．算出した類似度が閾値より大きければバージョン間でクラス，メソッド，フィールドを対応づける．対応づけの結果を元にリファクタリング操作のリストを生成する．類似度の閾値は，10 個の OSS からランダムに選択された 10 個のコミットに対する実験結果をもとに決定されている．

2.5 APIDiff における API の変更の検出および分類方法

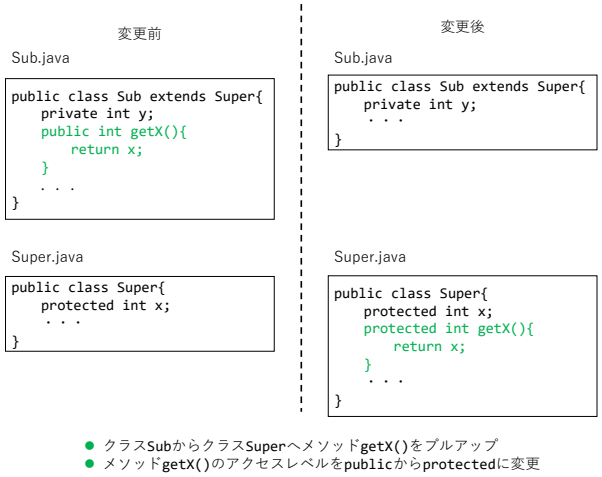
APIDiff は Java ライブラリの 2 つのバージョンを入力として受け取り，RefDiff を用いてそのバージョン間で API に対して行われた変更操作のリストを出力する [5]．API の変更の検出と分類は次の 5 つのステップで構成される．

Step 1 クラス，メソッド，フィールドの抽出
Step 2 類似度の計算によるリファクタリングの検出
Step 3 クラス，メソッド，フィールドの抽出
Step 4 バージョン間で API をマッピング
Step 5 API の変更の検出
Step 6 後方互換性を壊す変更かどうかを判別

Step 1 および Step 2 は RefDiff によって行われる処理である．Step 1 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，バージョンごとにクラス，メソッド，フィールドを抽出する．Step 2 ではコードの類似度を計算し，バージョン間で行われたリファクタリング操作のリストを得る．このリファクタリング操作のリストには API に対するリファクタリングだけでなくプライベートメソッドのメソッド名の変更のように API 以外に対するリファクタリングも含まれている．そのため API 以外に対するリファクタリング操作をフィルタリングする．Step 3 では APIDiff の入力として与えられた Java ライブラリの 2 つのバージョンからバージョンごとにクラス，メソッド，フィールドを抽出する．Step 1 のクラス，メソッド，フィールドの抽出は RefDiff 内部で行われる処理であるのに対して，Step 3 の処理は APIDiff 内部で行われる処理である．Step 4 ではバージョン間での API の比較および Step 2 で得られたリファクタリング操作のリストを元に API をマッピングする．まず 2 つのバージョン間で，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 3 つ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づける．対応づけることができなかった API をリファクタリング操作のリストをもとにリファクタリングされた API，削除された API，追加された API に分類する．Step 5 では，Step 4 で行われた API のマッピングの結果と API の修飾子やアノテーションなどを参照し，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といった情報を得る．Step 6 では検出した変更を変更の種類ごとに後方互換性を壊す変更と後方互換性を維持する変更に分類し，変更の種類や変更前後の API，後方互換性を壊す変更かどうかの判別結果などの情報を参照可能な API の変更操作のリストを作成する．



(a) APIDiff の出力



(b) 実際の API の変更操作

図 1 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 をリファクタリングされた API，削除された API，追加された API に分類している．しかしコードの類似度の閾値を適切に設定することは難しいため，RefDiff で検出および分類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および分類できなかった場合は，そのリファクタリングされた API を削除された API または追加された API として分類してしまう．また APIDiff は変更を主体として考え，検出した変更を後方互換性を壊す変更と後方互換性を維持する変更に分類している．API に対して複数の変更が行われる場合, 一部の変更は後方互換性を壊す変更として分類されるにもかかわらず，他の変更は後方互換性を維持する変更として分類されるため，それらの変更によって API の後方互換性が維持されているかを正しく判別できない．

図 1 に実際の API の変更操作と APIDiff の出力が異な

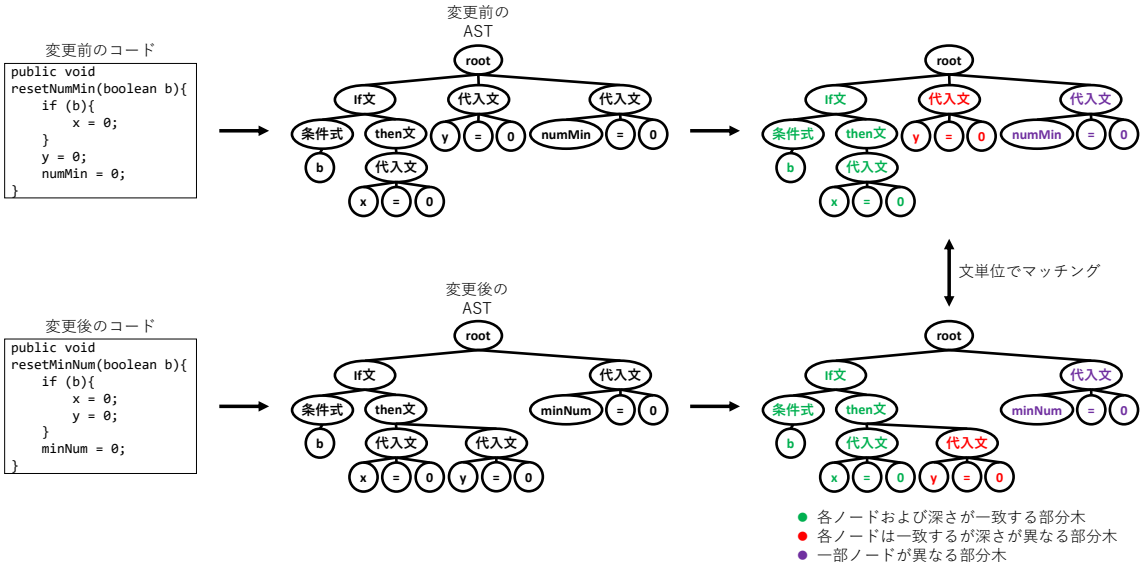


図 3 RefactoringMiner の概要

```
+ @Deprecated
- public int getY(){
+ protected int getY(){
    return y;
  }
```

図 2 複数の変更が同時に行われた API の例

る例を示す．変更前後で APIDiff を用いて得られる変更操作は，変更前のクラス Sub のメソッド getX() の削除，変更後のクラス Super のメソッド getX() の追加となる (図 1(a))．しかし，実際に行われた API の変更操作はクラス Sub からクラス Super へメソッド getX() のプルアップおよび public から protected にアクセスレベルの変更である (図 1(b))．この違いは，RefDiff の類似度の閾値が高く設定されていることが原因であり，APIDiff はメソッドのプルアップとして検出することができない．クラス Sub を継承したクラスからメソッド getX() を利用していた API 利用者は，コードを変更せずに API を利用し続けることができるにもかかわらず，クラス Sub のメソッド getX() 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．

また後方互換性を維持する変更と後方互換性を壊す変更が同時に行われた API の例を**図 2** を示す．メソッド getY() に対してメソッドの非推奨化と public から protected にアクセスレベルの変更が行われている．APIDiff は非推奨化を後方互換性を維持する変更として，public から protected への変更を後方互換性を壊す変更として分類する．1 つの API に対して後方互換性を壊す変更と後方互換性を維持する変更が行われており，それらの変更によって API の後方互換性が維持されているかを判別するには粒度が細かい．

そこで，本研究では類似度の閾値に依存しない方法で API の変更をより高い精度で検出する手法を提案する．そしてある API に対して複数の変更が行われた場合でもそれらの変更によって API の後方互換性が維持されているかを判別する手法を提案する．

4. 提案手法

提案手法では，AST の文単位のマッチングによりリファクタリングを検出する RefactoringMiner を利用し，API の変更を検出する．これによってコードの類似度の閾値に依存せず，API の変更を分類することが可能である．そして API を主体として考え，API ごとに変更をグループ化し，後方互換性を壊す変更が含まれているかを調べることで，それらの変更によって API の後方互換性が維持されているかを判別する．

4.1 RefactoringMiner

図 3 のように 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，クラス、メソッド、フィールドを抽出する．メソッドの AST を生成してバージョン間で部分木を文単位でマッチングし，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．文単位のマッチングは 3 段階で行われる．1 段階目は各ノードおよび深さが一致する部分木をマッチングさせる．葉ノードについては変数名や定数などの値を比較し，一致すれば同じノードとしてみなす．それ以外のノードについては if 文や代入文などラベルが一致すれば，同じノードとみなす．2 段階目は文の移動などに対応するために各ノードは一致するが深さが異なる部分木をマッチングさせる．3 段階目は変数名や型変更などに対応するために一部ノードが異なる部分木をマッチングさせる．このマッチングで

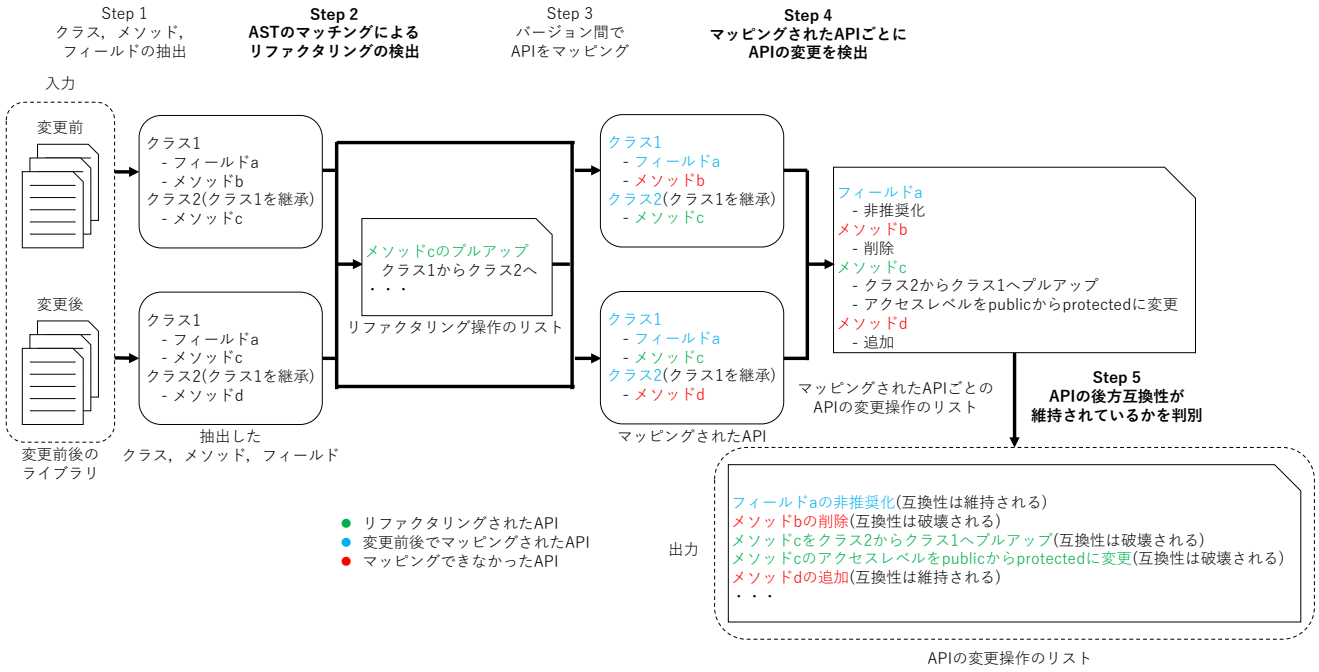


図 4　提案手法の概要

は部分木の深さ考慮しない．マッチングに成功した文の数が失敗した文の数よりも多ければ AST を比較した 2 つのメソッドを変更前後のメソッドとして対応づける．マッチングに成功したメソッドの組み合わせが複数ある場合，一致した文の数や完全一致した文の数などでソートし，もっとも値が高い組み合わせを変更前後の組み合わせとみなす．そして変更前後のメソッドをその後に行うマッチング対象のメソッドから除外することで，処理量が膨れ上がらないようにマッチングの回数を減らす．このマッチングの結果を元にリファクタリング操作のリストを生成する．3 段階で構成されるマッチングを行うことで，一部が異なる文のマッチングでも文の類似度を計算したり閾値と比較したりする必要がない．AST の対応づけではマッチングに成功した文の数とマッチングに失敗した文の数を利用し，コードの類似度の計算や閾値との比較は行わない．そのため類似度の閾値に依存せずにリファクタリングを検出できる．RefactoringMiner は RefDiff が検出可能な変更の種類に加えて変数名の変更やメソッドの戻り値の型変更，フィールドの型変更など様々な種類が検出できる．

4.2　API の変更の検出および分類方法

提案手法において API の変更の検出と分類は次の 5 つのステップで構成される．提案手法の概要を**図 4** に示す．

Step 1 クラス，メソッド，フィールドの抽出
Step 2 AST のマッチングによるリファクタリングの検出
Step 3 バージョン間で API をマッピング
Step 4 マッピングされた API ごとに API の変更を検出
Step 5 API の後方互換性が維持されているかを判別

既存手法と大きく異なる点は 3 つある．1 つ目は，Step 2 でコードの類似度の計算によってリファクタリングを検出するのではなく，AST のマッチングによってリファクタリングを検出する点である．これによりコードの類似度に依存せずに API の変更を検出できる．2 つ目は，既存手法の Step 3 で APIDiff の内部でもクラス，メソッド，フィールドを抽出し，その結果を利用して API をバージョン間でマッピングしているのに対して，提案手法では実行時間の短縮のため Step 1 で抽出したクラス，メソッド，フィールドを再利用している点である．3 つ目は，既存手法において他の変更を考慮せずに変更の種類ごとに後方互換性を壊す変更なのかどうかを判別していたのに対して，提案手法では Step 4 でマッピングされた API ごとに API の変更を検出し，Step 5 でマッピングされた API ごとに後方互換性を壊す変更が含まれているかを調べることで API の後方互換性が維持されているかを判別している点である．これにより API に対して複数の変更が行われた場合でも，それらの変更によって API の後方互換性が維持されているかを判別できる．

提案手法では既存手法と同様に 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，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．Step 1 では提案手法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，バージョンごとにクラス，メソッド，フィールドを抽出する．論文 [10] では RefactoringMiner のバージョンは 2.0 だが，その後，著者らによってサポートするリファクタリングが拡張された 2.1 が公開されており，提案手法で

は 2.1 を利用した．Step 2 では AST を文単位でマッチングすることで，バージョン間で行われたリファクタリング操作のリストを得る．このリファクタリング操作のリストには API に対するリファクタリング操作だけでなく，ローカル変数の変数名の変更やプライベートメソッドのメソッド名の変更など API 以外に対するリファクタリング操作も含まれている．そのため API 以外に対するリファクタリング操作をフィルタリングする．Step 1 および Step 2 は RefactoringMiner によって行われる処理である．Step 3 では既存手法と同様にバージョン間で API をマッピングする．ただし変更前後の API の比較において，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づ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づけたメソッドの戻り値の型やフィールドの型を比較することでメソッドの戻り値の型やフィールドの型の変更を検出するのではなく，RefactoringMiner によってメソッドの戻り値の変更やフィールドの型の変更を検出し，そのリファクタリング操作をもとに API をリファクタリングされた API として分類するためである．これによって変更前後でクラスの完全限定名やメソッド名，フィールド名が一致しない場合でも，メソッドの戻り値の型変更やフィールドの型変更を検出でききるようになった．Step 4 では，Step 3 で行われた API のマッピングの結果と API の修飾子やアノテーションなどを参照し，API の変更を検出する．このとき変更前後の API とその API に対して行われた変更を対応付けることで，複数の変更を管理する．Step 5 ではマッピングされた API ごとに後方互換性が維持されているかどうかを判別する．まず他の変更を考慮せずに各変更を変更の種類ごとに後方互換性を壊す変更かどうか判別する．そして API ごとに，API と関連づけた変更操作の中に後方互換性を壊す変更が含まれているかを確認する．後方互換性を壊す変更が含まれている場合，それらの変更によって API の後方互換性が壊されたと判別し，後方互換性を壊す変更が含まれていない場合は，変更によって後方互換性が維持されていると判別する．その後，変更の種類や変更前後の API，その変更によって API の後方互換性が維持されているかなどの情報を参照可能な API の変更操作のリストを作成する．

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 4 つの評価項目を設定した．

項目 1：API に対する変更の検出数

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 に対する変更の検出数にどのような変化が見られるのかを調べる．また変更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．

項目 2：検出した変更の適合率

提案手法によって API の変更を既存手法より高い精度で検出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リファクタリングや修飾子の変更などであるため，検出された変更が正しいのかを確認する．

項目 3：後方互換性が維持されるかどうかの判別精度

その変更によって後方互換性が維持されているかどうかの判別結果が正しいのかを確認する．提案手法では 1 つの API に対して後方互換性を壊す変更と後方互換性を維持する変更が行われた場合に，後方互換性を壊す変更と後方互換性を維持する変更の両変更によって API の後方互換性が壊されたと判別される．そのため既存手法で常に後方互換性を維持する変更として分類される変更のうち，その変更によって API の後方互換性が壊されたと判別された変更が正しいのかを確認する．

項目 4：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．また実行時間が長くなる要因となる処理を確認する．

5.2 既存手法の実装

既存手法の実装は．既存研究 [5] の著者らによって実装された [11] を利用した．

5.3 実験対象

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および既存手法が適用可能な OSS を選出するために，RefactoringMiner を用いた縦断的研究 [12] の実験対象である OSS の中から，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．これらの OSS に対して，提案手法および既存手法が適用可能であることを確認した．実験対象の OSS を**表 1**

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	72,696	4,839	2021 年 4 月 22 日
Retrofit	26,995	1,865	2021 年 2 月 24 日
MPAndroidChart	25,232	2,068	2020 年 10 月 30 日
LeakCanary	26,269	1,609	2021 年 4 月 8 日
Hystrix	50,510	2,108	2018 年 11 月 20 日
iosched	23,550	2,757	2020 年 6 月 26 日
Fresco	97,194	2,897	2021 年 4 月 29 日
Logger	1,441	144	2018 年 4 月 10 日

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．種類別の検出数の比較と検出した変更の適合率の比較，後方互換性が維持されるかどうかの判別精度の確認では 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．

5.4 項目 1：API に対する変更の検出数の比較

評価項目 1 の API に対する変更の検出数について述べる．

5.4.1 全リポジトリに対しての検出数の比較

実験対象の 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，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た結果が**表 2** である．既存手法では実験対象のリポジトリから 3,774（=2,582+1,192）個の API の変更しか検出できなかったのに対して，提案手法では 7,376（=2,582+4,794）個の API の変更を検出できた．すべてのプロジェクトにおいて提案手法が検出した API の変更の数が，既存手法が検出した API の変更の数を上回った．また提案手法のみが検出した API の変更の数は，提案手法または既存手法によって検出された API の変更の総数の 30.0%〜70.6%を占めており，その平均は 54.3%であった．それに対して既存手法のみが検出した API の変更の数は，提案手法または既存手法によって検出された API の変更の総数の 2.8%〜29.8%を占めており，その平均は 13.5%であった．

5.4.2 種類別の検出数の比較

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 API の変更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た 28（=24+4）種類の変更に加えて，Change in Parameter List や Rename Field など新たに 9 種類の変更を検出した．また両手法で検出された 28 種類の変更のうち Lost Visibility in Method 以外の 27 種類に関しては提案手法は既存手法以上の数の変更を検出した．

[*1] LeakCanary 及び iosched 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．

5.5 項目 2：検出した変更の適合率の比較

評価項目 2 の検出した変更の適合率について述べる．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し適合率を比較した．提案手法のみで検出された API の変更と既存手法のみで検出された API の変更を目視確認した．提案手法のみで検出された API の変更の数が多かったため，許容誤差 5%，信頼度 95%となるように 311 個サンプリングを行った．また API の変更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 の変更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** の適合率の列である．Inline 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 50.0%であったのに対して提案手法の適合率が 90.0%と高い結果となった．また提案手法で検出した API の変更の数と既存手法で検出した API の変更の数が等しい 4 種類の変更についても適合率を求めた．両手法で検出された API の変更を目視確認した結果が表 3 の適合率の列である．4 種類の変更の適合率はすべて 100%となった．この結果から両手法で検出された変更の適合率は高いと考えられる．

5.6 項目 3：後方互換性が維持されるかどうかの判別精度

評価項目 3 の後方互換性が維持されるかどうかの判別精度について述べる．8 個の OSS の中から MPAndoridChart を選択し，既存手法で後方互換性を維持する変更として常に分類される変更のうち，提案手法ではその変更によって API の後方互換性が壊されたと判別された変更を目視確認し，判別結果が正しいかを確認した．その結果を**表 4** に示す．提案手法で API の後方互換性が壊されたと判別された変更は 61 個あり，そのうち 53 個が変更内容および判別結果ともに正しかった．

表 2 API に対する変更の検出数

プロジェクト名	両手法が検出した変更の数	提案手法のみが検出した変更の数	既存手法のみが検出した変更の数	合計
OkHttp	675（44.1%）	460（30.0%）	396（25.9%）	1,531
Retrofit	243（36.5%）	338（50.8%）	84（12.6%）	665
MPAndroidChart	1,120（39.4%）	1,607（56.5%）	116（ 4.1%）	2,843
LeakCanary	41（24.0%）	79（46.2%）	51（29.8%）	171
Hystrix	292（24.4%）	722（60.3%）	183（15.3%）	1,197
iosched	91（32.7%）	143（51.4%）	44（15.8%）	278
Fresco	452（19.4%）	1,514（65.1%）	359（15.4%）	2,325
Logger	29（26.6%）	77（70.6%）	3（ 2.8%）	109
合計	2,582（32.3%）	4,794（54.3%）	1,192（13.5%）	8,841

5.7 項目 4：既存手法との実行時間の比較

評価項目 4 の実行時間について述べる．実験対象の 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．各処理の CPU 実行時間を計測するためにプロファイラを利用した．実験に用いた計算機の性能を**表 5** に示す．実験結果を**図 5** に示す．図 5 においてリファクタリングの検出は RefDiff や RefactoringMiner によるクラス，メソッド，フィールドの抽出からリファクタリングの検出までの処理を表している．またクラス，メソッド，フィールドの抽出は APIDiff 内部で行われるクラス，メソッド，フィールドの抽出処理を，API の変更の検出はバージョン間での API のマッピングから API 変更操作のリスト作成までに要する処理を表している．図 5(a) は提案手法の実行時間および既存手法の実行時間がともに 5 分以内となった OSS の結果で，それ以外の OSS の結果を図 5(b) に示している．8 個の OSS のうち 5 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短くなった．残りの 3 個の OSS において，提案手法の実行時間が長くなった主な原因はリファクタリングの検出に要する時間が長くなったためである．

表 4 後方互換性が維持されるかどうかの判別結果

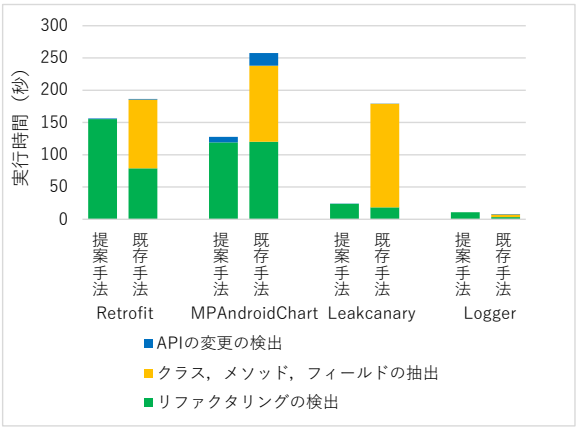
種類	個数
判別結果が正しい変更	53
変更内容が誤っている変更	2
変更内容は正しいが，判別結果が誤っている変更	6

表 5 実験に用いた計算機の性能

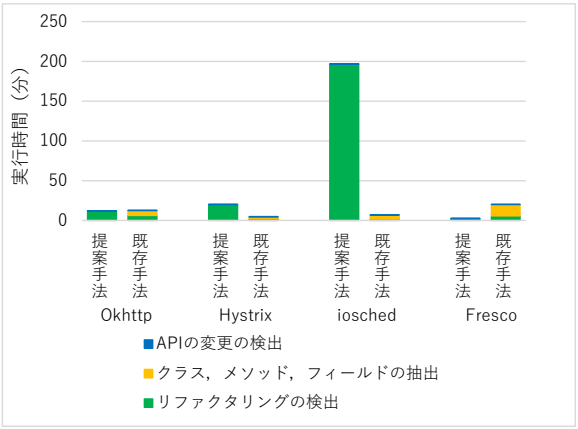
OS	macOS Big Sur
CPU	Intel Core i5（1.4GHz，4 コア）
GPU	Intel Iris Plus Graphics 645
メモリ	16GB

表 3 種類別の検出数と適合率

	両手法が検出した API の変更			提案手法のみが検出した API の変更			既存手法のみが検出した API の変更		
API の変更の種類	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（%）	検出数	サンプルサイズ	適合率（%）
Change in Field Default Value	107	0		18	13	100	1	1	100
Change in Return Type Method	125	0		56	13	100	3	3	100
Extract Method	0	0		133	14	78.6	4	4	25.0
Inline Method	5	0		39	13	84.6	4	4	100
Lost Visibility in Method	19	0		32	13	38.5	44	44	0.0
Pull Up Method	115	0		107	14	100	20	20	100
Push Down Field	6	0		2	2	100	1	1	100
Push Down Method	28	0		29	13	100	2	2	100
Move Field	45	0		75	14	100	1	1	100
Move Method	60	0		46	13	15.4	12	12	66.7
Rename Method	147	0		67	14	100	22	22	68.1
Rename Type	27	0		2	2	100	2	2	100
Add Static Modifier in Method	1	0		3	3	100	0	0	
Change in Field Type	53	0		10	10	50.0	0	0	
Change in Supertype	132	0		2	2	100	0	0	
Deprecated Method	6	0		48	13	100	0	0	
Deprecated Type	3	0		2	2	94.7	0	0	
Gain Visibility in Field	43	0		35	13	73.7	0	0	
Gain Visibility in Method	47	0		56	13	92.3	0	0	
Gain Visibility in Type	2	0		4	4	100	0	0	
Lost Visibility in Field	8	0		18	13	100	0	0	
Move and Rename Type	3	0		2	2	100	0	0	
Move Type	69	0		8	8	100	0	0	
Pull Up Field	28	0		45	13	100	0	0	
Change in Parameter List	0	0		626	14	100	0	0	
Extract Field	0	0		3	3	100	0	0	
Extract Subtype	0	0		2	2	100	0	0	
Extract Supertype	0	0		36	13	100	0	0	
Extract Type	0	0		25	13	100	0	0	
Move and Rename Field	0	0		4	4	100	0	0	
Move and Rename Method	0	0		32	13	100	0	0	
Remove Static Modifier in Method	0	0		2	2	100	0	0	
Rename Field	0	0		58	13	84.6	0	0	
Add Final Modifier in Field	1	1	100	0	0		0	0	
Add Supertype	28	28	100	0	0		0	0	
Remove Final Modifier in Field	5	5	100	0	0		0	0	
Remove Supertype	7	7	100	0	0		0	0	
平均	1,120			1,607	311	90.0	116	116	50.0



(a) 実行時間が 5 分未満のプロジェクト



(b) 実行時間が 5 分以上のプロジェクト

図 5 API の変更の検出に要する時間

```
  public interface OnChartGestureListener {
-     public void onChartLongPressed(MotionEvent me);
+     void onChartLongPressed(MotionEvent me);
  }
```

図 6 Lost Visibility in Method の誤検出の例

6. 考察

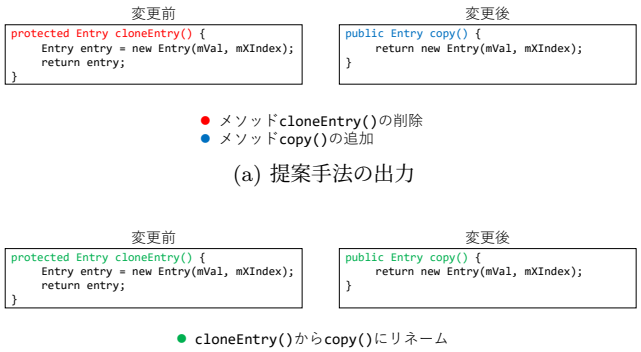
表 3 において既存手法の適合率が 0%であった Lost Visibility in Method と既存手法のみが検出可能な変更について考察する．また表 4 において変更内容は正しいが，判別結果が誤っている変更について考察する．

6.1 Lost Visibility in Method

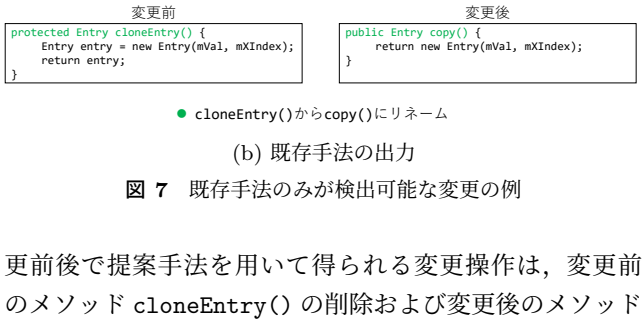
既存手法で検出した変更の例を**図 6** に示す．APIDiff はメソッド `onChartLongPressed(MotionEvent me)` のアクセスレベルを `public` から `default` への変更として検出する．しかしこのメソッドはインタフェースのメソッドでありアクセス修飾子 `public` を削除してもアクセスレベルは変更されないので．誤検出としてみなすことができる．

6.2 既存手法のみが検出可能な変更

既存手法のみが検出可能な変更の例を**図 7** に示す．変



(a) 提案手法の出力



(b) 既存手法の出力

図 7 既存手法のみが検出可能な変更の例

更前後で提案手法を用いて得られる変更操作は，変更前のメソッド `cloneEntry()` の削除および変更後のメソッド `copy()` となる（図 7(a)）．しかし実際に行われた API の変更操作はメソッドのリネームであり，既存手法ではこの変更操作を正しく検出することができる（図 7(b)）．この違いは，マッチングに成功した文の数とマッチングに失敗した文の数が同じ値であるため，RefactoringMiner は変更前後のメソッドとして対応づけることが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 return 文における変数 `entry` を代入文を用いて置換し，return 文のみをマッチングする．

6.3 変更内容は正しいが判別結果が誤っている変更

変更内容は正しいが判別結果が誤っている変更の例を**図 8** に示す．検出された変更操作は `protected` から `public` へアクセスレベルの変更およびパラメータリストの変更である．提案手法ではこれらの変更によって API の後方互換性が壊されたとして判別した．しかしメソッドのパラメータの型の変化に注目すると変更後の API は変更前の API と後方互換性を持っていることがわかる．そのため 2 つの変更によってメソッド `setData` の後方互換性は維持されていると判別すべきである．変更によって API の後方互換性が維持されているかを正しく判別できるようにするためには，継承関係を考慮して互換性を壊す変更なのかを判別する必要がある．

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．しかしこの結果は第一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 API の変更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．

本実験ではアクセスレベルが `public` または `protected`

```
-   public abstract class Chart extends View implements AnimatorUpdateListener {
+   public abstract class Chart<T extends ChartData<? extends DataSet<? extends Entry>>>
+           extends View implements AnimatorUpdateListener {

-       protected void setData(ChartData<? extends DataSet<? extends Entry>> data) {
+       public void setData(T data) {
            . . .
        }
    . . .
    }
```

図 8 変更内容は正しいが判別結果が誤っている変更の例

なクラス，メソッド，フィールドを API とみなしている．しかし実際には API として公開するためではなく内部処理のためにアクセスレベルを `public` または `protected` に設定している場合がある．そのため開発者が意図しないクラスなどを除くと実験結果が変化する可能性がある．

8. おわりに

本研究では，コードの類似度の閾値に依存せず AST を用いてリファクタリングを検出する RefactoringMiner を利用することで API の変更を検出し，API に対して複数の変更が行われた場合でもそれらの変更によって API の後方互換性が維持されているかを判別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 1 つの API に対して後方互換性を維持する変更と後方互換性を壊す変更が行われた場合でもそれらの変更によって API の後方互換性が維持されているか判別できることを確認した．

今後の課題としては対象の OSS を増やし，検出数が少ない API の変更について十分な数を目視確認することや継承関係を考慮して後方互換性を壊す変更かどうかを判別することでそれらの変更によって後方互換性が維持されているかどうかの判別精度を向上させることが挙げられる．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：20H04166）の助成を得て行われた．

参考文献

[1] Moser, S. and Nierstrasz, O.: The effect of object-oriented frameworks on developer productivity, *Computer*, Vol. 29, No. 9, pp. 45–51 (1996).
[2] Michail, A.: Data mining library reuse patterns in user-selected applications, *the proceedings of IEEE International Conference on Automated Software Engineering*, pp. 24–33 (1999).
[3] Dig, D. and Johnson, R.: How do APIs evolve? A story of refactoring, *Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 2, pp. 83–107 (2006).
[4] Moreno, L., Bavota, G., Penta, M. D., Oliveto, R., Marcus, A. and Canfora, G.: ARENA: An Approach for the Automated Generation of Release Notes, *IEEE Transactions on Software Engineering*, Vol. 43, No. 2, pp. 106–127 (2017).
[5] Brito, A., Xavier, L., Hora, A. and Valente, M. T.: APIDiff: Detecting API breaking changes, *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 507–511 (2018).
[6] Silva, D. and Valente, M. T.: RefDiff: Detecting Refactorings in Version Histories, *the proceedings of IEEE/ACM International Conference on Mining Software Repositories*, pp. 269–279 (2017).
[7] Xavier, L., Brito, A., Hora, A. and Valente, M. T.: Historical and impact analysis of API breaking changes: A large-scale study, *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 138–147 (2017).
[8] Brito, A., Xavier, L., Hora, A. and Valente, M. T.: Why and how Java developers break APIs, *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 255–265 (2018).
[9] Xavier, L., Hora, A. and Valente, M. T.: Why do we break APIs? First answers from developers, *the proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 392–396 (2017).
[10] Tsantalis, N., Ketkar, A. and Dig, D.: RefactoringMiner 2.0, *IEEE Transactions on Software Engineering*, pp. 1–21 (2020).
[11] Brito, A.: A tool to identify API breaking and non-breaking changes between two versions of a Java library. [Online]., Available: `https://github.com/aserg-ufmg/apidiff`.
[12] Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M. and Chávez, A.: Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects, *the proceedings of Joint Meeting on Foundations of Software Engineering*, pp. 465–475 (2017).