

# 探索的手法による impure リファクタリング検出

堤 祥吾<sup>1</sup> 吉田 則裕<sup>2</sup> 崔 恩瀾<sup>3</sup> 井上克郎<sup>1</sup>

**概要:** 現在, impure リファクタリングを検出することは困難である. 林らは複数のリファクタリングを検出するため, 探索的手法を用いた検出手法に関する研究を行った. 本研究では, 林らの手法に加えソースコード差分検出技術を組み合わせた手法を提案し, impure リファクタリングの検出を可能にした. オープンソースプロジェクトの Apache Xerces, Apache Ant のリポジトリから impure リファクタリングを含むコミットを選択し, 提案手法を適用した.

**キーワード:** リファクタリング, メソッド抽出, ソースコード差分検出, 探索的手法

## Search-Based Impure Refactoring Detection

SHOGO TSUTSUMI<sup>1</sup> NORIHIRO YOSHIDA<sup>2</sup> EUNJONG CHOI<sup>3</sup> KATSURO INOUE<sup>1</sup>

### 1. まえがき

リファクタリングとは, ソフトウェアの外部的振る舞いを保ったままで, 内部の構造を改善していく作業をいう [2]. ソースコードが設計の欠陥や度重なる変更, 追加により複雑化してしまった場合に, リファクタリングを適用することで, 外部から見た振る舞いを変更せずにソースコードを修正することができる. リファクタリングは Fowler によっていくつかのパターン [2] に分類され, 多くの研究で各パターンを 1 つのリファクタリングとみなしている.

現在, リファクタリングがソースコードの品質に与える影響について関心が持たれている. [13][14][9] そのため, バージョン管理システムのソースコード変更履歴からリファクタリングの検出を行うことが必要である. 調査において効率的にリファクタリングを検出するためには, 自動的にリファクタリング検出を行うことができるツールが必要である. そこで, リファクタリング検出手法について多くの研究がなされている [12].

林らは, 探索的手法を用いることで, 複数のリファクタリングが同時に適用されていても検出する手法を提案した [4]. しかし林らの手法では, リファクタリング以外の操作が適用された場合に検出結果を出すことはできない. そこで本研究では, 林らの手法に加え, ソースコードの差分検出とテストを用いることで, そのようなリファクタリングを検出することを可能とする.

本研究では, Java のソースコードを対象とし, 提案手法を Eclipse プラグインとして実装した. また, オープンソースプロジェクトである Apache Ant<sup>\*1</sup>, Apache Xerces<sup>\*2</sup> に対して提案手法を適用し, 複数のリファクタリングやリファクタリング以外の変更が適用された impure リファクタリングを検出できることを確認した.

以下, 2 章では本研究の前提知識となるリファクタリングを説明し, 3 章では, 提案手法を説明する. 4 章では適用結果を示し, 5 章ではまとめと今後の課題について述べる.

### 2. リファクタリング

この章では, リファクタリングの説明と例を紹介し, impure リファクタリングについて説明する. そして, 既存のリファクタリング検出手法の問題点と, 先行研究である

<sup>1</sup> 大阪大学  
Osaka University

<sup>2</sup> 名古屋大学  
Nagoya University

<sup>3</sup> 奈良先端科学技術大学院大学  
Nara Institute of Science and Technology

<sup>\*1</sup> <http://ant.apache.org>

<sup>\*2</sup> <http://xerces.apache.org>

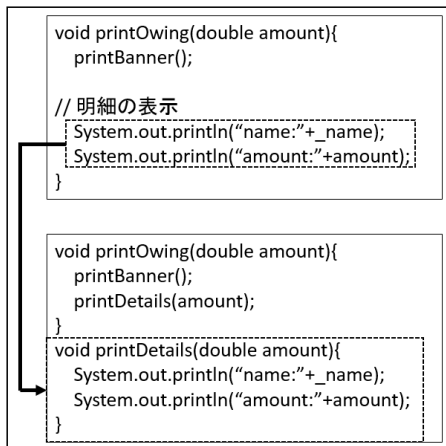


図 1 メソッド抽出の例

Fig. 1 An example of extract method refactoring

探索的手法を用いたリファクタリング検出について述べる。

## 2.1 リファクタリングのメリット

Fowler は著書 [2] でリファクタリングのメリットを 3 つ説明している。

- リファクタリングはソフトウェアを理解しやすくする
- リファクタリングはバグを見つけ出す
- リファクタリングでより速くプログラミングできる

リファクタリングはコードを読みやすい形に修正する作業を言う。そのため、リファクタリングによってコードを他者が理解しやすくなるのは明白である。また、リファクタリングを行う際にコードを理解し、その理解をリファクタリングによってコードに反映することで、自身によるコードの理解も深まる。リファクタリングによってコードが単純で理解しやすくなることで、バグの発見やプログラミング速度の上昇にもつながる。このような理由により、リファクタリングが行われている。

## 2.2 リファクタリングパターン

リファクタリングは Fowler によっていくつかのパターン [2] に分類され、多くの研究で各パターンを 1 つのリファクタリングとみなしている。ツールや手動でリファクタリングを行う場合、パターン単位でリファクタリングが行われている。リファクタリングパターンの例として、以下ではメソッド抽出とメソッド引き上げを紹介する。

### 2.2.1 メソッド抽出

メソッド抽出は、長い処理やひとまとまりの処理を取り出して、新しいメソッドを作成するリファクタリングである。メソッド抽出を行うことの利点として、単一メソッドの長さを短くすることで、ソースコードの見通しをよくする点や、メソッド化されたソースコードを再利用しやすくなる点が挙げられる。

図 1 は、支払いの明細を表示する “printOwing” メソッ

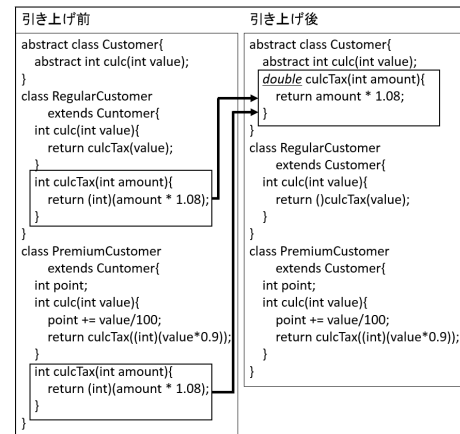


図 2 メソッド引き上げの例

Fig. 2 An example of pull up method refactoring

ドにメソッド抽出リファクタリングを適用した例である。ひとまとまりの処理である明細の表示部分を新しく作成した “printDetails” メソッドに移すことで、メソッドの機能を細かく分割している。また、処理自体に名前がつくことで “printOwing” メソッドの概要を素早く把握することができるようになる。

### 2.2.2 メソッド引き上げ

メソッド引き上げは、サブクラスのメソッドをスーパークラスに移動するリファクタリングである。

図 2 は顧客を表す “Customer” クラスとそのサブクラスに対し、メソッド引き上げを適用した例である。両方のサブクラスに税込み価格を計算する “culcTax” メソッドが存在し、どちらも処理内容は同じソースであるため、スーパークラスである “Customer” クラスに移動させている。これにより、コードの可読性を上げ、バグ発生の可能性を下げている。

複数のサブクラスに同じメソッドが存在する場合、あるメソッドに対して行った変更がほかのメソッドに対しても同様に行われる保証はない。もし変更漏れがあればバグとなりうる。また、同じメソッドが別の場所に存在することで、コードの可読性も減少してしまう。メソッド引き上げは、これらの問題を解消することが可能である。

### 2.2.3 impure リファクタリング

版管理システムにおいて、ソースコードに複数のリファクタリングが組み合わせて適用されている場合や、バグ修正や機能の追加など、リファクタリング以外の変更 (以下、非リファクタリング変更) が同時に適用されている場合がある。Gorg らは、このような変更を impure リファクタリングと呼んでいる [3]。impure リファクタリングでは、かならずしも外部的振る舞いが保存されるわけではない。

図 3 は Apache Ant のリポジトリにおける、あるコミットの変更の抜粋である。処理の詳細は割愛するが、コンストラクタ内でファイル名の正規化と確認が行われており、メソッド抽出と非リファクタリング変更が混在している。

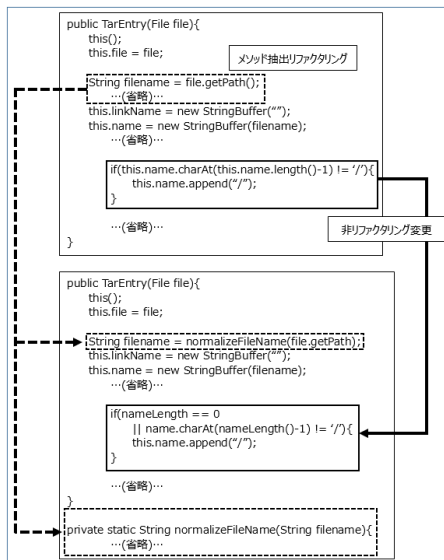


図 3 impure リファクタリングの例  
Fig. 3 An example of impure refactoring

点線枠で表された部分はメソッド抽出であり、fileName を生成する一連の処理を“normalizeFileName”メソッドとして抽出している。また、下の実線枠内のコードは上の赤い枠内のコードに非リファクタリング変更が加えられたコードである。

このように、リファクタリングと非リファクタリング変更が同時に適用されている変更や、複数のリファクタリングが適用されている変更を impure リファクタリングと言う。

## 2.3 リファクタリング検出手法

リファクタリングがソースコードの品質に及ぼす影響について調査するため、行われたリファクタリングを検出することが必要である。現状、リファクタリングを検出するために、バージョン管理システムのソースコード変更履歴の解析が行われているが、大規模なプロジェクトでのリファクタリングを目視で検出することは現実的ではない。そこで、リファクタリングの実施を自動的に検出する手法が数多く提案されている [12]。

### 2.3.1 検出規則を用いた手法

Prete らの手法では、版間の差分があらかじめ定められたパターン (検出規則) に一致する場合はリファクタリングが実施されたとする、検出規則を用いた手法をとっている [8]。以下に例としてメソッド引き上げの検出規則を示す [7]。

メソッド引き上げ (pull up method) の検出規則

```

move_method(fShortName,
            tChildFullName, tParentFullName)
^ before_subtype(tParentFullName,
                tChildFullName)
→ pull_up_method(fShortName,
                 tChildFullName, tParentFullName)
    
```

上の例によると、メソッド引き上げは tChildFullName クラスにある fShortName メソッドが tParentFullName クラスに移動されたという条件 (move\_method) と、tChild-FullName クラスが tParentFullName のサブクラスであるという条件 (before\_subtype) を満たしているリファクタリングである、ということを示している。このように、あらかじめ定義されている基本的なソースコード変更パターンを検出し、それら基本的パターンの組み合わせがリファクタリングパターンと一致している場合に、リファクタリングが検出されたとする。検出規則を定義しておけばそれに応じたリファクタリングを検出することができる。一方、検出規則の検討が不十分である場合に検出精度が低下する欠点がある。また、非リファクタリング変更が存在する impure リファクタリングの検出には向かない。

### 2.3.2 コードクローン検出を用いた手法

Weissgerber らは、版間の変更において、類似したコード片がどこに移動、抽出されたかを追跡することでリファクタリングを検出した [10]。

追跡にはコードクローン検出ツール [1][5] を用いるため、移動、抽出されたコード片が完全に一致していなくとも追跡することができる。非リファクタリング変更が存在してもリファクタリング検出が行える可能性があるが、複数の種類の変更やリファクタリングが適用されている場合の検出には向かない。

### 2.3.3 探索を用いた検出手法

上記、検出規則、コードクローン検出を用いた手法は、同一箇所に複数のリファクタリングが適用されている場合に、リファクタリングを検出することができない。しかし、探索を用いた手法により、そのような場合でもリファクタリングを検出することが可能となる。

林らは、探索的手法を用いてリファクタリングを検出する手法を提案している [4]。提案手法では、プログラムを状態、リファクタリング操作を状態遷移とみなし、版間の変更と一致するリファクタリング操作列を探索する。これにより、複数のリファクタリングが同時に適用されている場合においてもリファクタリング操作列を検出することを可能としている。

以下、林らの論文をもとに手法を説明する。

概要: 林らは、プログラム  $P_{old}$ ,  $P_{new}$  間に行われたり

ファクタリング操作の発見を、以下の探索問題として表現している。

- 状態：プログラム
- 初期状態： $P_{old}(=n_0)$
- 目標状態： $G = \{P_{new}(=n_m)\}$
- オペレータ：リファクタリング操作
- 経路コスト：それまでに行ったりファクタリング操作の回数

この手法では、経路探索として A\*探索 [11] が用いられている。探索中の状態  $n$  を含むコストは、評価関数  $f(n) = g(n) + h(n)$  で与えられ、 $g(n)$  は経路コスト、 $h(n)$  は  $n$  から目標状態 ( $=n_m$ ) までの予想コストを経験的に与えるヒューリスティックである。この  $h(n)$  が、探索の効率に直結するため、ヒューリスティックを工夫する必要がある。

**探索の流れ：** 探索の各状態で、以下を繰り返す。

- (1)  $n, n_m$  間の差分をもとに  $h(n)$  を計算する。
- (2) 差分をもとにリファクタリング操作候補を選出し、順位付けを行う。
- (3) 順位をもとに、 $n$  にリファクタリング操作を適用していき、得たソースコード群を新たに  $n$  とする。

これを、 $n$  と  $n_m$  との差分がなくなったときの  $n$  か、一定回数繰り返した後で最も  $h(n)$  が小さい  $n$  を取り出し、その状態に至るまでに適用したリファクタリング操作列を結果として得る。

**ヒューリスティックの計算：** この手法で使われたヒューリスティックは以下の式で表される。

$$h(n, o) = h'(n) / \alpha(o)$$

ここで  $h'(n)$  は状態の差分をもとに計算した距離のヒューリスティックである。また、 $\alpha : O \rightarrow (0, 1]$  は選出したリファクタリング操作候補の評価値であり、その候補が実際に行われた可能性が高いほど大きい値をとる。そのため  $h(n, o)$  は、評価の低いリファクタリングを適用する場合は、ソースコードの差分から考える距離  $h'(n)$  よりも大きな値をとるようになっている。

**手法の問題点：** 林らの手法では、非リファクタリング変更が版間に適用されている場合は完全な操作列の検出が不可能である。本研究では、林らの手法に加え、ソースコード差分検出技術を用いることで、非リファクタリング変更が適用されている場合でも操作列を検出することを可能にする。

### 3. 提案手法

本研究では、探索的手法によるリファクタリング操作列検出とソースコード差分検出技術の組み合わせにより、impure リファクタリングを検出する手法を提案する。

提案手法の概要は図 4 の通りである。

- (1) 対象プロジェクトからあるコミット (旧版コード) とそれより一つ新しいコミット (新版コード) を取り出す
- (2) 探索的手法により検出したリファクタリング操作列を適用し、新版コードに近づいた旧版コード (探索後コードとする) を得る
- (3) リファクタリング操作列を適用した後正常に実行できるか動作確認を行う
- (4) 探索後コードと新版コードの差分を検出してそれを非リファクタリング操作列とし、探索後コードに適用する

上記手法を適用後、リファクタリング操作列と非リファクタリング差分を結果として得る。

#### 3.1 探索的手法によるリファクタリング検出

探索の手法は林らの手法 [4] を参考に実装を行った。検出の流れは以下の通りである。

- (1) 旧版コード (初期状態) を、評価値 (3.1.1 にて説明) とともに優先度つきキューに入れる
- (2) キュー内で最も評価値の優れた状態 ( $s_i$  とする) を取り出す
- (3)  $s_i$  から新版コードへの変更で、適用されたと考えられるリファクタリングを列挙する ( $\delta_0, \delta_1, \dots$  とする)
- (4)  $s_i$  に対して各  $\delta_0, \delta_1, \dots$  を適用し、新たな状態  $\delta_0(s_i), \delta_1(s_i), \dots$  を得た後、各評価値を計算し、キューに入れる
- (5) 終了条件を満たしていなければ 2 に戻る。満たしていれば、探索中最も評価値の優れた状態を探索結果として得る

終了条件とは、

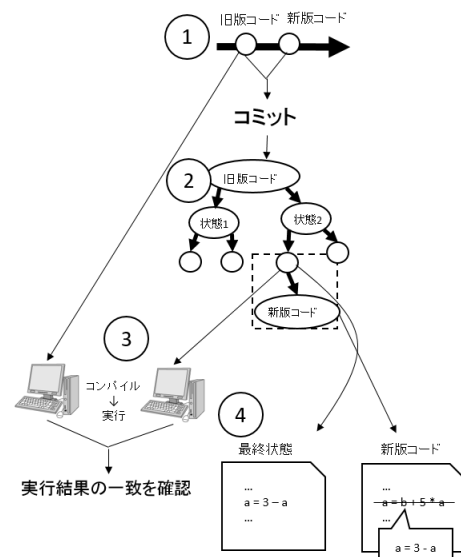


図 4 提案手法の概要

Fig. 4 An overview of the proposal technique

- キューが空になる
- 探索中のコードと新版コードが一致する
- 一定時間 (本研究では 600 秒としている) が経過するのいずれかである。

探索的手法に基づくリファクタリング検出の段階は以上であり、次の段階で、上記手法で得られた最終状態のテストを行う。

### 3.1.1 評価関数による評価値の算出

探索中に取り出される状態 (以下、探索状態) の優先順位をつけるために評価関数を用いて評価値を計算している。本研究では、目的状態に早くたどり着くため、3.3.3 で説明するレーベンシュタイン距離 [6] を利用し、評価値を算出している。

評価値の求め方を説明する。レーベンシュタイン距離は名前的一致するメンバごと、トークン単位で求めている。ここで、探索状態のメンバ ( $A_1, A_2, \dots, A_n$ ) とマッチング (後述) する目的状態のメンバを ( $B_1, B_2, \dots, B_3$ ) とする。各  $A_i$  と  $B_i$  ( $i=1, 2, \dots, n$ ) とのレーベンシュタイン距離を  $d_i$ 、 $A_i, B_i$  のトークン数をそれぞれ  $a_i, b_i$  とする。このとき、探索状態の評価値は、

$$\frac{\sum_{i=1}^n d_i}{\sum_{i=1}^n \max(a_i, b_i)}$$

として表される。分母である総トークン数は分子である差分より大きくなるため、評価値は  $[0, 1]$  の値をとる。

### 3.1.2 重複状態での探索打ち切り

たとえば、メソッド A, B があったとして、A にメソッド引き上げを適用した後に B にメソッド引き上げを適用した場合と、B にメソッド引き上げを適用した後に A にメソッド引き上げを適用した場合に、同じ状態にたどり着いてしまう場合がある。同じ状態に対して探索を実行するのは無駄であるので、2 回目以降で同じ状態にたどり着いた場合、そこで探索を打ち切ることが必要である。

本研究では、ソースコードのハッシュ値と評価値両方が一致した場合に探索打ち切りを行うことにした。評価値の求め方は上で説明した通りなので、ハッシュ値の求め方について説明を行う。探索状態のハッシュ値を求めるために、プロジェクト全体のソースコードを文字列として得た後、Java の String クラスの hashCode メソッドと同じアルゴリズムを用いることでハッシュ値を得た。

文字列の各文字を

$$(s_0, s_1, \dots, s_{n-1})$$

で表すとき hashCode の値は、

$$31^{n-1}s_0 + 31^{n-2}s_1 + \dots + s_{n-1}$$

で表される。Java で標準的に用いられておりハッシュの衝突が少ないアルゴリズムのため、この手法を用いた。

## 3.2 探索後の状態の動作確認

探索によってリファクタリング操作列が適用されたソースコードを得る。ここで、探索後の状態 (探索後コード) の動作と、リファクタリング操作列を適用する前の状態 (旧版コード) の動作が一致するかどうかを確認する。リファクタリングは、正しく適用された場合は動作を変えることはないが、失敗した場合思わぬ動作変更を起こす場合がある。正しく適用されたことを確認するために動作確認を行う。

本研究では、プロジェクト付属のテストケースを利用する。あらかじめ旧版コードのテスト結果を保持しておき、探索後コードとのテスト結果を比較する。結果が一致しない場合はリファクタリング失敗として扱い実行を終了する。結果が一致する場合は非リファクタリング差分検出に移る。

## 3.3 非リファクタリング差分検出

探索において、すべてのリファクタリングを検出できたものと仮定する。この場合、探索的手法によって得られた最終状態のソースコード (以下、探索後コード) と、新版コードとの差分は非リファクタリング差分となる。

非リファクタリング差分検出手法の流れは以下の通りである。プロジェクトに複数のファイルがある場合は、同一名のファイルごとに以下を適用する。

- (1) 探索後コードと新版コードをそれぞれトークン列に変換する
- (2) トークン列同士のレーベンシュタイン距離を算出する差分を求める際に、空白やコメントを無視したいので、ソースコードはトークン列として扱う。トークン列同士の差分は、一般に文字列に使われるレーベンシュタイン距離算出のアルゴリズムをもとに求めた。

差分を取得した後、各ファイルごとに得られたリファクタリング操作列と非リファクタリング変更について表示を行う。

### 3.3.1 ソースコードをトークン列に変換

Java ソースコードを、スペースや記号で区切ることでトークン列に変換している。例として、以下のようにソースコードからトークン列への変換が行われる。

元となる Java ソースコード

```
class Example{
    void example(){
        int a = 3 + 4;
    }
}
```

トークン列 (カンマ区切り)

```
[class, Example, {, void, example, (, ),
{, int, a, =, 3, +, 4, ;, }, ]
```

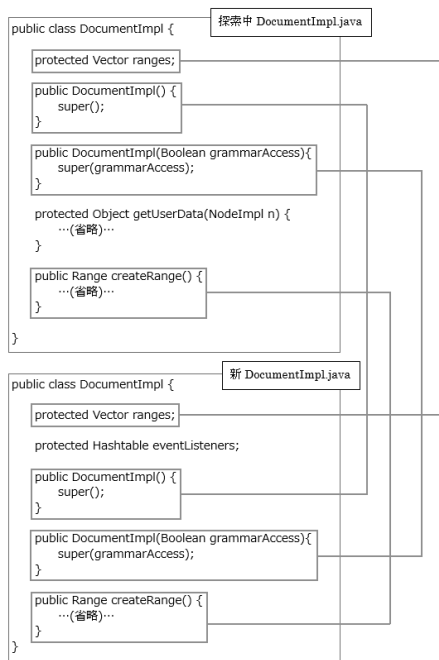


図 5 メンバ間のマッチング例

Fig. 5 An example of matching between members

差分検出の方法としては、文字単位の差分、トークン単位の差分、行単位の差分が考えられたが、このうちトークン単位の差分を採用した理由としては次の通りである。

ソースコードを変更する際、文字単位で変更を行うことは少ない。さらに、差分検出単位を細かくしてしまうと計算時間が多くかかってしまうという理由もあり、文字単位での差分検出を行わなかった。また、差分検出において空白やコメントなどは無視している。行単位で差分をとる場合、改行の位置によっては同じ処理が異なるものと判定される可能性がある。このため、本研究ではトークン単位の差分を取得し利用している。

### 3.3.2 メンバのマッチング

各探索段階においてクラス内のメンバの順序は不定である。本研究の差分検出においてメンバの順序は考慮しないため、マッチングしたメンバごとに差分をとっている。メンバをソートする方法もあるが、計算時間と実装量の問題からこちらの方法を選択した。

マッチングの手法について説明を行う。図 5 は、Apache Xerces プロジェクトの DocumentImpl.java ファイルの一部であり、探索中の DocumentImpl.java(以降ファイル A)と探索の最終目標となる DocumentImpl.java(以降ファイル B)でのマッチングを図に表したものである。実線で結ばれたメンバ同士がマッチングしていて、そうでないメンバはマッチング相手となるメンバがなかったということを示している。マッチングしなかったメンバについては、空白トークン列とマッチングしたものとして扱う。

マッチング相手の選び方だが、フィールドについては名前的一致、メソッドについては名前と引数型的一致によ

て行っている。またインナークラスに関して、本来は再帰的に内部のメソッドについてマッチングを行うべきであるが、本研究ではそれを省略し、クラス名でマッチングした後、メソッドと同様にインナークラスごと差分をとるという方法を行っている。

### 3.3.3 レーベンシュタイン距離を用いたトークン列の差分検出

編集距離の一つであるレーベンシュタイン距離 [6] は、2つの文字列がどの程度異なるかを示す距離である。1文字の削除、挿入、置換の3つの操作を組み合わせ、一方の文字列からもう一方の文字列に変換するための最小操作回数をいう。一般には文字列に用いられる距離であるが、本研究では文字をソースコードのトークンに置き換えて考えることで、トークン列同士の距離を算出している。

## 4. 適用実験

オープンソースプロジェクトである Apache Xerces<sup>\*3</sup> (SVN リポジトリ)、Apache Ant<sup>\*4</sup> (Git リポジトリ) のリポジトリから impure リファクタリングが含まれる連続する2つのコミットを選択し、本研究の手法を適用した。また、プロジェクトからリファクタリングが含まれるファイルを選択し、コミット間に適用されたリファクタリングを対象として実験を行った。実験データは表 1 と表 2 の通りである。表中左の ID は、付録の CommitID と対応している。括弧内は、適用範囲が親クラスに及んだことを示している。表中のファイルに対して、メソッド引き上げ、フィールド引き上げ、メソッド抽出のリファクタリングを対象として実験を行った。

### 4.1 結果

図 6 は、実験データに対して探索を行った際の状態遷移を表すグラフである。ここでは、DocumentImpl.java へ本

表 1 Apache Xerces での実験対象ファイル

Table 1 File for the experiment from the Apache Xerces

ID	対象ファイル
1	DocumentImpl.java(CoreDocumentImpl.java)

表 2 Apache Ant での実験対象ファイル

Table 2 Files for the experiment from the Apache Ant

ID	対象ファイル
2	Launcher.java
3	ComponentHelper.java
4	AntTypeDefinition.java
5	Main.java
6	Project.java
7	TarEntry.java

\*3 <http://xerces.apache.org>

\*4 <http://ant.apache.org>

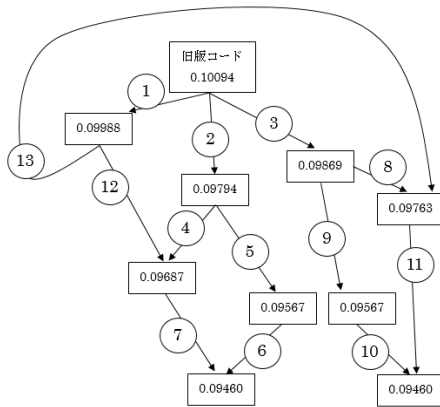


図 6 探索の状態遷移図

Fig. 6 State transition diagram on searching

手法を適用した場合を示している。各四角は探索中にたどり着いた状態を表しており、矢印は状態遷移を表している。矢印上に振ってある数字の順に探索が行われた。四角内の数字は評価値の値で、値が小さいほど新版コードとの差分が小さいことを示している。各番号が示す遷移で適用されたリファクタリングは表 3 のとおりである。ここで、適用されたリファクタリングとは、F がフィールド引き上げを表し、M がメソッド引き上げを表している。

#### 4.1.1 非リファクタリングとして検出されたトークン

探索結果であるリファクタリング操作列を適用後、リファクタリングとして検出されなかった差分を非リファクタリング差分として検出した。非リファクタリング差分として追加、削除されたトークン数は表 4 の通りである。正しく非リファクタリングであった変更については“correct add”, “correct delete”, リファクタリングとして検出されるべきであった変更については, “incorrect add”, “incorrect delete” として表している。これらは目視で確認を行った。

表 3 DocumentImpl.java に適用されたリファクタリング

Table 3 Refactorings applied to DocumentImpl.java

番号	リファクタリング	ターゲットメンバ
1	F	Hashtable userData
2	M	setUserData(NodeImpl, Object)
3	M	getUserData(NodeImpl)
4	F	Hashtable userData
5	M	getUserData(NodeImpl)
6	F	Hashtable userData
7	M	getUserData(NodeImpl)
8	F	Hashtable userData
9	M	setUserData(NodeImpl, Object)
10	F	Hashtable userData
11	M	setUserData(NodeImpl, Object)
12	M	setUserData(NodeImpl, Object)
13	M	getUserData(NodeImpl)

## 4.2 考察

DocumentImpl (pull up method と pull up field を含む) と Launcher, TarEntry(extract method を含む) に対して、正しく impure リファクタリング検出を行うことができた。ComponentHelper に対しては、3つのリファクタリングのうち2つを検出でき、1つを見逃していた。これは、見逃した extract method リファクタリングが抽出先で大きく追加があったため、似た処理とみなされずにリファクタリングが行われなかったことが原因と考えられる。AntTypeDefinition では、2つある extract method リファクタリングの両方を検出できなかった。原因として、extract method の検出では抽出処理をブロックを考慮して、連続した領域を選択するが、実際に行われたリファクタリングはブロックを越えて連続しない領域を抽出していた点が考えられる。このため、ツールによるリファクタリングの後に編集距離が大きくなり、extract method リファクタリングが採用されなかった。

いくつかのデータに対して検出できなかったリファクタリングはあったが、実験データに対して本研究の手法により、impure リファクタリングの検出を行うことができた。

**探索の状態数:** 本実験では、対象とするコミットやファイルを限定して探索を行っている。そのため、探索でとった状態数は列挙可能な大きさであったが、対象とするコミットやファイル、候補となるリファクタリングが膨大に

表 4 非リファクタリング差分として追加されたトークン

Table 4 Added tokens as non-refactoring differences

クラス	correct add	incorrect add	合計
DocumentImpl	12	0	12
CoreDocumentImpl	621	0	621
Launcher	130	0	130
ComponentHelper	313	119	432
AntTypeDefinition	123	50	173
Main	1029	127	1156
Project	34	41	75
TarEntry	27	0	27

表 5 非リファクタリング差分として削除されたトークン

Table 5 Deleted tokens as non-refactoring differences

クラス	collect delete	incorrect delete	合計
DocumentImpl	0	0	0
CoreDocumentImpl	47	0	47
Launcher	6	0	6
ComponentHelper	233	110	343
AntTypeDefinition	0	46	46
Main	2784	115	2899
Project	3	35	38
TarEntry	10	0	10

なった場合は現実的な時間内に十分な空間を探索できない可能性がある。

**探索における重複状態取り除き:** 過去にたどり着いた状態から再度探索を行うのは非効率であるため、重複状態にたどり着いた場合にそれを取り除く処理を行っている。しかし、図6の遷移5, 9と遷移6, 7, 10, 11では、評価値が完全に一致しているにも関わらず異なる状態であると判断された。これは、各ファイルのクラスメンバは一致しているが、クラスメンバの順番が異なるのが原因である。

上記状態を同一と判断したい場合、クラスメンバのある規則に従い並べ替え、そのうえでハッシュ値の計算を行うという方法が考えられる。

**動作確認を行うタイミング:** 本研究では、状態遷移におけるリファクタリングの失敗可能性の低さ、テストの時間的コストを考慮し、動作確認を1度だけ行うこととしている。本研究で用いた実験データでは、1度のテストにおよそ60秒ほどの時間がかかり、探索における状態遷移がおよそ2秒であることにに対し30倍の時間がかかっている。本研究で最も時間がかかっているのは探索であるため、毎回の探索で動作確認を行った場合に実行時間が30倍になってしまう。ただし、テストの実行にかかるコストが十分小さい場合は、状態遷移毎に動作確認を行い、リファクタリング検出の精度向上を図ることが可能であると考えられる。

## 5. まとめ

本研究では、探索的手法とソースコード差分検出手法を組み合わせることで、impure リファクタリングの検出を行った。既存のリファクタリング検出技術では、複数のリファクタリングが同時に適用されていた場合や、リファクタリング以外の変更が適用されていた場合にリファクタリングを検出することが難しい。本研究の手法を適用することで、そのようなimpure リファクタリングを含む実験データからリファクタリングを検出することができた。

今後の課題として、複数の改善案が挙げられる。まず、今回適用していないリファクタリング検出機能の追加である。本実験ではメソッド引き上げとフィールド引き上げ、メソッド抽出を適用したが、メソッド展開等、それ以外のリファクタリング機能の追加を行うことが考えられる。また、リファクタリング検出の対象を大きくした場合に、現状では探索空間が大きくなってしまい十分な探索が行えない。本研究では実行時間や探索空間を意識せず手法を実装したが、今後リファクタリング候補の列挙や探索の枝刈りを効率化することも検討していきたい。

**謝辞** 本研究は JSPS 科研費 25220003, 26730036, 15H06344 の助成を受けた。

## 参考文献

- [1] Biegel, B. and Diehl, S.: Highly Configurable and Extensible Code Clone Detection, in *Proc. of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pp. 237–241 (2010).
- [2] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison Wesley (1999).
- [3] Gorg, C. and Weissgerber, P.: Detecting and visualizing refactorings from software archives, in *Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*, pp. 205–214.
- [4] Hayashi, S., Tsuda, Y. and Saeki, M.: Search-Based Refactoring Detection from Source Code Revisions, *IE-ICE Trans. Inf. & Syst, Vol.E93-D, No.4*, pp. 754–762 (2010).
- [5] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering, Vol.28, Issue: 7*, pp. 654–670 (2002).
- [6] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals, *Soviet Physics Doklady 10*, pp. 707–710 (1966).
- [7] Prete, K., Rachatasumrit, N. and Kim, M.: Catalogue of Template Refactoring Rules, *Technical Report UTAUSTINECE-TR-041610, The University of Texas at Austin* (2010).
- [8] Prete, K., Rachatasumrit, N., Sudan, N. and Kim, M.: Template-based reconstruction of complex refactorings, in *Proc. of the 26th International Conference on Software Maintenance (ICSM'10)*, pp. 1–10 (2010).
- [9] Soetens, Q. D. and Demeyer, S.: Studying the Effect of Refactorings: A Complexity Metrics Perspective, in *Proc. of the 7th Int'l Conf. Quality of Information and Communications Technology (QUATIC 2010)*, pp. 313–318 (2010).
- [10] Weissgerber, P. and Diehl, S.: Identifying Refactorings from Source-Code Changes, in *Proc. of the 21th International Conference on Automated Software Engineering (ASE'06)*, pp. 231–240 (2006).
- [11] Zeng, W. and Church, R. L.: Finding shortest paths on real road networks: the case for A\*, *International Journal of Geographical Information Science 23*, pp. 531–543 (2009).
- [12] 崔 恩澗, 藤原賢二, 吉田則裕, 林 晋平: 変更履歴解析に基づくリファクタリング検出技術の調査, *コンピュータソフトウェア*, vol.32, No.1, pp. 47–59 (2015).
- [13] 雑賀 翼, 崔 恩澗, 後藤 祥, 吉田則裕, 井上克郎: 組み合わせて実施されたリファクタリングの調査, *電子情報通信学会技術研究報告*, Vol.114, No.23, pp. 1–6 (2014).
- [14] 雑賀 翼, 崔 恩澗, 吉田則裕, 春名修介, 井上克郎: Code Smellの深深度がリファクタリングに与える影響の調査, *ソフトウェアエンジニアリングシンポジウム 2015 論文集*, pp. 176–181 (2015).