情報処理学会論文誌 Vol.56 No.12 2346–2357 (Dec. 2015)

構文情報リポジトリを用いた細粒度リファクタリング検出手法

藤原 賢二[1,a)] 吉田 則裕[2,b)] 飯田 元[1,c)]

受付日 2015年1月15日, 採録日 2015年9月2日

概要：本稿はソフトウェアのリファクタリング実施履歴をリビジョン単位で復元する手法を提案する．提案手法は，構文情報の変更を追跡可能なリポジトリを用いることで，計算時間の削減を行っている．提案手法をツールとして実装し，オープンソースソフトウェアである jEdit に適用した．その結果，従来手法である Ref-Finder，UMLDiff と比較してより高速かつ高精度にリファクタリングの実施履歴を復元可能であることを確認した．提案手法を用いてリファクタリングの実施履歴を分析することで，リファクタリングがソフトウェアの品質に与える影響の調査，有効なリファクタリング支援手法の検討に役立つことが期待できる．

キーワード：リファクタリング，リファクタリング検出，リポジトリマイニング，版管理システム

An Approach for Fine-grained Detection of Refactoring Instances using Repository with Syntactic Information

Kenji Fujiwara[1,a)] Norihiro Yoshida[2,b)] Hajimu Iida[1,c)]

Received: January 15, 2015, Accepted: September 2, 2015

Abstract: This paper proposes a method for fine-grained detection of refactoring instances from the software repository. The proposed method reduces computation time for the refactoring detection by tracing changes of syntactic information based on the repository that stores the results of syntactic analyses. We implemented the proposed method and applied it to the jEdit project. The result of comparison with the UMLDiff algorithm and Ref-Finder shows that the proposed method provides high preciseness for the refactoring detection. Refactoring instances that are provided by the proposed method are expected to be as valuable data for investigating the effectiveness of refactoring and to investigate refactoring support tool that should be implemented.

Keywords: refactoring, refactoring detection, mining software repositories, version control system

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舞いを変更することなく内部の構造を改善することをいう．Fowler は，リファクタリングを行うことで欠陥の混入を減少させることができ，また，開発者の生産性の向上を促すといった効果があると述べている [1]．そのため，ソフトウェアの品質を高めるための技術として注目されている．

近年，ソフトウェアの開発履歴からリファクタリングの実施履歴を復元するための手法（リファクタリング検出手法）が数多く研究されている．リファクタリング検出はソフトウェア開発者と研究者に次のような利益をもたらす．

利益 1 リファクタリングの適用事例を収集することで，研究者がリファクタリングの効果を実証的に分析・検証することができる．

利益 2 開発者がどのようにリファクタリングを実施して

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[2] 名古屋大学
Nagoya University, Nagoya, Aichi 464–8601, Japan
[a)] kenji-f@is.naist.jp
[b)] yoshida@ertl.jp
[c)] iida@itc.naist.jp

いるのかを分析することで，有効なリファクタリング支援手法の検討に役立つ．

利益 1 に関する研究では，リファクタリングがソフトウェア開発に与える影響を実証的に調査するために，ソフトウェアの開発履歴に対してリファクタリング検出手法を適用する．Kim らは 3 つのオープンソースプロジェクトを対象に，リファクタリングが欠陥修正と開発者の生産性に対してどのような影響を与えているかを調査している [2]．また，Ratzinger らと Fujiwara らはそれぞれリファクタリングが欠陥の混入に与える影響について調査を行っている [3], [4]．

利益 2 について，Choi らはコードクローンとリファクタリングの関係に着目し，開発者らがどのようなコードクローンに対してリファクタリングを実施しているのか分析している [5]．その結果として，メソッドオブジェクトによるメソッドの置き換えリファクタリングを支援するツールが必要であると報告している．

従来のリファクタリング検出手法は，ソフトウェアのリリースバージョン間において実施されたリファクタリングを検出することを暗黙に想定している．リリース間での比較では，リファクタリングの正確な実施時期や，リファクタリングの適用者，リファクタリング適用時点のソースコードなどの情報が失われる．リファクタリングに適切な時期やその効果の分析を行うためには，より細かい時間的粒度での検出が必要である．そのためには，開発履歴へ記録されたリビジョン単位でのリファクタリング検出を実施することが望ましい．本稿では，開発履歴中のすべての隣接しているリビジョンの組からリファクタリング検出を行うことを，細粒度リファクタリング検出と呼ぶ．以降，特に断りのない限り隣接する 2 つのリリースバージョン間とリビジョン間をそれぞれリリースバージョン間，リビジョン間と表記する．

先に述べたように従来のリファクタリング検出手法は，リリースバージョン間からの検出を目的としている．リリースバージョン間の変更は，差分の大きいリビジョン間の変更と見なすことができるため，従来手法を用いて細粒度リファクタリング検出を実施することは可能である．しかし，従来手法の評価はたかだか数十のバージョン間を対象とした検出の精度や速度を評価対象としているため，検出時間を短縮するための工夫が十分になされていない．そのため，数千や数万を超えるリビジョン間を対象とした細粒度リファクタリングを行うためには多大な時間を要するという問題がある．ソフトウェアの開発履歴から，リファクタリングの実施に関する情報をリビジョンの粒度で取得するためには，従来手法よりも高速なリファクタリング検出手法が求められる．そこで本稿では，細粒度リファクタリング検出を高速に実施可能なリファクタリング検出手法を提案する．

従来手法の計算時間は，リビジョン（バージョン）ごとに構文解析を行う処理と，あるリビジョンにおける構文要素が次のリビジョンにおけるどの構文要素に対応するかを求める処理が大きな割合を占めている．計算時間の観点で，これらの処理について従来手法が抱える問題点は以下のとおりである．

問題点 1 リビジョン間で変更されなかったファイルについても繰返し構文解析を行っている．

問題点 2 リビジョン間で変更されなかった構文要素についても繰返し対応付けを行っている．

問題点 3 本来不要な部分の構文情報を取得し，リビジョン間での対応付けを行っている．

問題点 3 について，リビジョン間で構文要素の対応付けを行う処理は，Origin Analysis と呼ばれている [6]．従来手法は，この技術を発展させる形で提案されている．そのため，構文解析においてあらかじめ変数の read/write，メソッドの callee/caller などの情報を解析している．しかし，あるリビジョンにおける全メソッドの情報が必要なわけではなく，特定の条件を満たすメソッド内の構文情報があればリファクタリング検出は可能である．また，これらの情報はすべてのリファクタリングパターンの検出に必要なわけではない．

これらの問題点を解決するために，提案手法では構文情報リポジトリ Historage [7] を用いてリファクタリング検出を行う．構文情報リポジトリを用いることでソースコードの構文解析結果をキャッシュすることができ，構文解析にかかる時間を削減できる（問題点 1）．また，構文情報リポジトリは，メソッド本体やクラス階層などの情報からハッシュ値を計算することで，リビジョン間で差分のある箇所を効率的に知ることができる．そのため，構文情報の対応関係を求める際に重複した計算を避けることができる（問題点 2）．また，構文情報リポジトリに記録されない情報は必要になった段階で取得する（問題点 3）．

リファクタリングには，コードの抽出を行うものと，クラス間でメソッドの移動を行うものが多い．本稿では，それらのリファクタリングで代表的な「メソッド抽出」と「メソッドの引き上げ」の 2 種類について提案手法を実装した．そして，オープンソースソフトウェアである jEdit を対象に従来手法である Ref-Finder，UMLDiff との比較を行い，提案手法の有効性を評価した．評価の結果，提案手法は従来手法より高速かつ高精度にリファクタリングを検出できることを確認した．

以降，本稿は 2 章で背景を，3 章で構文情報リポジトリの構築手順と，そのリポジトリを用いたリファクタリング検出について述べる．4 章と 5 章では従来手法との比較結果とその考察を述べ，6 章にて本稿をまとめる．

2. 背景

本章では，本稿におけるリファクタリング検出の定義を説明するとともに，既存のリファクタリング検出手法を紹介する．また，本稿で検出の対象とするメソッド抽出およびメソッドの引き上げリファクタリングについて説明し，提案手法が利用する Historage について述べる．

2.1 リファクタリング検出

本稿で扱うリファクタリング検出は，始点および終点となるリビジョンにおけるソースコードのスナップショット S_x，S_y を入力として与えると，始点 S_x から終点 S_y において行われた変更を解析し，その間に実施されたリファクタリングの集合を返す．

本稿で扱うソースコードの変更解析に基づくリファクタリング検出は，次の手順で実施される．

手順 1 ソースコードの構文解析
手順 2 構文要素の対応付け
手順 3 リファクタリングパターンにマッチングする変更をリファクタリングとして検出する

検出の手順としてはまず，S_x と S_y のすべてのファイルに対して構文解析を行い，クラスやメソッドなどの構文要素を得る．次に，構文要素の対応付けを行い，S_x から S_y への変更において構文要素に対して行われた変更（追加，削除，修正）を求める．最後に，構文要素およびそれらの変更に対して各種リファクタリングパターンとのマッチングを行う．そして，リファクタリングの種類と，適用対象のメソッドやクラスなどの情報を出力する．

以上のような手順でリファクタリング検出を行う既存の手法として，Xing らは UMLDiff アルゴリズムを用いたもの [8] を，Prete らは Ref-Finder [9] を提案している．Xing らの UMLDiff アルゴリズムは構文要素の階層構造と，各要素の名称についてそれぞれ類似度を求め，それらを基に対応付けを行う [10]．このアルゴリズムの適用結果をデータベースに格納し，そこからリファクタリングの検出を行うためのクエリが提案されている [8]（以降，本稿では UMLDiff アルゴリズムを用いたリファクタリング検出を UMLDiff と表記する）．Prete らの Ref-Finder は，手順 1，手順 2 により得られる情報を論理プログラミングの述語として表現する [9]．そして，手順 3 に対応するパターンを述語の組合せとして定義するのが特徴である．彼らは文献 [11] において Fowler のリファクタリングカタログに記載されているリファクタリング 72 件のうち 65 件のパターンを記述している．

2.2 メソッド抽出リファクタリング

メソッド抽出リファクタリングとは，あるメソッドから機能的に意味のある単位でコード片を選択し，新たなメソッドとして抽出することをいう [1]．このリファクタリングが実施される主な目的として，重複したコードをメソッドとして抽出することでコードの保守性を向上させることがあげられる．提案手法では，以下に示す条件を満たすものをメソッド抽出リファクタリングとして定義する．

条件 A-1 抽出対象のメソッドと抽出したメソッドは同一クラス内に定義されている．
条件 A-2 メソッド抽出時に，抽出対象のメソッドに抽出したメソッドの呼び出しが追加されている．

2.3 メソッドの引き上げリファクタリング

メソッドの引き上げリファクタリングとは，継承関係にあるクラス間において，子クラスに存在するメソッドを親クラスへ移動することをいう [1]．提案手法では Prete らによる定義 [11] を基に，以下に示す条件を満たすものをメソッドの引き上げリファクタリングと定義する．

条件 B-1 リファクタリング対象となるメソッドが，そのメソッドを保持するクラスの親クラスへ移動されている[*1]．
条件 B-2 リファクタリングの対象となるメソッドを保持するクラスは，変更前の時点で移動先のクラスを継承している．
条件 B-3 メソッドの移動後も移動元のクラスが存在している．

2.4 Historage

Git や Subversion などに代表される従来の版管理システムは，ソースコードの変更をファイル単位で管理する．そのため，ファイルの追加，削除，変更に関する情報は容易に取得できるが，メソッドの追加や削除に関する情報を得るためには各リビジョンにおけるソースコードを解析し，その対応を求める必要があった [12]．Hata らはメソッド，コンストラクタ，フィールドの単位でソースコードの変更を管理するための Historage を提案している [7]．Hitorage は各リビジョンにおけるソースコードを構文解析し，得られた構文木を Git リポジトリ上のディレクトリ階層に対応付けて記録する．具体的には，クラスがメソッドを保持することをディレクトリ上の親子関係で表現し，メソッドの本体をファイルとして表現する．このように構文情報を版管理システムに記録することで，通常のファイル変更を追跡するのと同様の操作でメソッド本体に対する変更を追跡できる．本稿で提案する手法は，リファクタリング検出に必要な構文情報とその差分を求めるにあたり，Historage を構文情報リポジトリとして利用する．しかし Historage は，メソッド

[*1] Fowler は複数の子クラスから単一の親クラスへメソッドを引き上げる場合をメソッドの引き上げリファクタリングとしている．しかし，本稿では UMLDiff と同様に単一の子クラスからメソッドを引き上げる場合もメソッドの引き上げリファクタリングとして扱う．

の引き上げリファクタリングの検出に必要なクラスの継承に関する情報を保存しない．提案手法では Historage の拡張を行うことで検出に必要な情報を取得できるようにする．

3. 構文情報リポジトリを用いた細粒度リファクタリング検出

提案手法は，版管理システムに記録された開発履歴（リポジトリ）を入力として，リポジトリ中の全リビジョン間からリファクタリング検出を実施する．手順として，まず，分析対象のリポジトリから構文情報リポジトリを構築する．次に，構文情報リポジトリから得られる構文情報とその差分を用いて，各リビジョン間で実施されたリファクタリングを検出する．

以降，本章では 3.1 節で構文情報リポジトリを構築する手順について説明し，3.2 節と 3.3 節で構文情報リポジトリが提供する情報からメソッド抽出およびメソッドの引き上げリファクタリングを検出する手順について説明する．

3.1 構文情報リポジトリの構築

提案手法は，Historage を用いてメソッド抽出，メソッドの引き上げリファクタリングの検出に必要な構文情報およびその差分を取得する．それぞれのリファクタリングを検出するのに必要な情報を**表 1** に示す．メソッド抽出リファクタリングの検出においては，メソッドの変更を行った箇所についてのみ知ることができれば十分である．そこで，提案手法ではメソッド呼び出しの追加を除いた情報を取得できるように構文情報リポジトリを構築し，メソッド呼び出しの追加については必要になった段階で取得する（次節で詳述）．

図 1 (a) に変換前のディレクトリ構造を，図 1 (b) に変換後のディレクトリ構造を示す．ここで，変換前のディレクトリ構造に存在する `foo/bar/HistorageConverter.java` の構文情報は，変換後の `foo_bar_HistorageConverter.java` ディレクトリ以下に保存されている．この Java ファイルのパッケージ情報は，変換後のディレクトリ直下にある `package` ファイルに保存されている．また，図中の `[CN]` および `[MT]` ディレクトリには，それぞれクラス，メソッドに関する情報が格納されている．たとえば，図 1 (b) からは，`HistorageConverter` クラスには `convert(Parser)` メソッドが定義されていることが分かる．ここで，`convert(Parser)` ディレクトリ直下の `body` ファイルにはメソッドの本体が，`parameter` ファイルにはメソッドのパラメータ情報が記録されている．

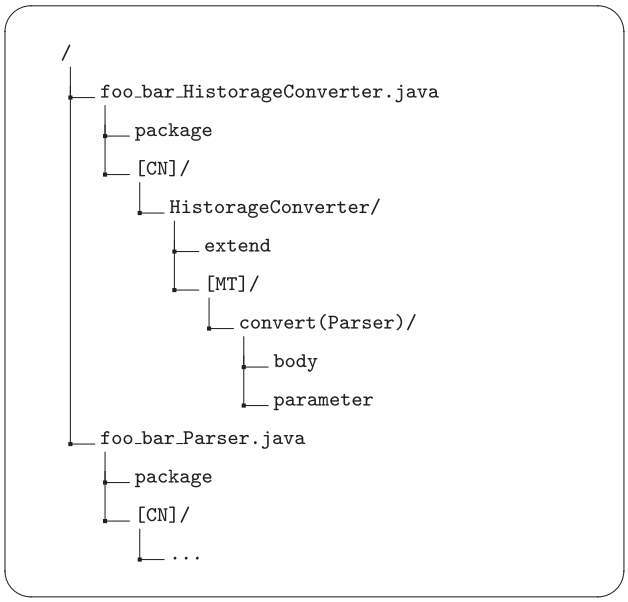
メソッドの引き上げリファクタリングの検出には，あるクラスが継承しているクラスを知る必要がある．しかし，Hata らが提案している Historage ではクラスの継承関係を記録していない．そこで，親クラスの名前をクラスに対応するディレクトリの直下にテキストファイルとして記録し（図 1 (b) における `extend`），メソッドの引き上げリファクタリングの検出に利用する．

各コミットを処理する際には，変更されたソースファイルのみ新規に構文解析を行い，変更されなかった分については直前のコミットで変換されたディレクトリ構造を再利用する．これは，構文木から変換したディレクトリ構造は，基になったソースファイルと一対一に対応するため，同一のソースファイルからは同一のディレクトリ構造が得られるからである．これにより，提案手法は既存の手法と比較して計算時間の大幅な短縮ができる．

表 1 リファクタリング検出に必要な構文情報
Table 1 Syntactic information for refactoring detection.

		メソッド抽出	メソッドの引き上げ
構文要素	パッケージ情報	✓	✓
	クラス	✓	✓
	メソッドのシグネチャ	✓	✓
	メソッドの本体	✓	✓
	クラスの継承		✓
差分情報	メソッド呼び出しの追加	✓	
	メソッド本体の変更	✓	
	メソッドの追加	✓	✓
	メソッドの削除		✓

(a) 変換前



(b) 変換後

図 1 ディレクトリ構造の変換
Fig. 1 Tranformation of directory structure.

3.2 メソッド抽出リファクタリングの検出

前節で説明した手順で構築した，構文情報リポジトリのリビジョン間の差分を分析することで，メソッド抽出リファクタリングの検出を行う．提案手法が検出するメソッド抽出リファクタリングは，メソッド抽出の対象となったメソッド（$targetMethod$）とリファクタリングにより新たに作成されたメソッド（$extractedMethod$），$targetMethod$ から抽出されたコードと $extractedMethod$ の類似度（3.4 節参照）に関する情報で構成される．

図 2 にメソッド抽出リファクタリングの候補を検出する手続きを示す．検出はメソッドの追加および変更が実施されたクラスごとに実施される（6 行目）．まず，新規に作成されたメソッドを見つけ，$extractedMethod$ の候補とする（7 行目）．次に，行の追加および削除が行われたメソッドを見つけ，$targetMethod$ の候補とする（9 行目）．そして，$targetMethod$ に追加された行に $extractedMethod$ の呼び出しが存在すれば（12 行目），メソッド抽出リファクタリングの候補として，$targetMethod$ と $extractedMethod$ の類似度を計算する（13，14 行目）．なお，前節で述べたように構文情報リポジトリにはメソッド呼び出しに関する構文情報は記録されていない．そこで，$targetMethod$ に追加された行に対して部分的な構文解析を行うことでメソッド呼び出しに関する情報を取得する．これにより，提案手法はメソッド呼び出しに関する構文解析を行う対象を削減し，計算時間を短縮できる．これらの手続きによりメソッド抽出リファクタリングの候補を検出し，最終的に類似度が閾値以上のものをリファクタリングとして検出する．

3.3 メソッドの引き上げリファクタリングの検出

提案手法により検出されるメソッドの引き上げリファクタリングは，変更前に子クラスに存在した引き上げ対象のメソッド（$sourceMethod$），親クラスに作成された引き上げ後のメソッド（$destinationMethod$）とこれらメソッド間の類似度に関する情報で構成される．

図 3 にメソッドの引き上げリファクタリングの候補を検出する手続きを示す．メソッドの引き上げリファクタリングは，追加されたメソッドと変更されたメソッドの組に対して，追加されたメソッドを保持するクラスが削除されたメソッドを保持するクラスの親クラスである場合に（11 行目）それぞれのメソッドの類似度を計算し，リファクタリングの候補とする（13–15 行目）．

3.4 類似度の計算

本節ではメソッド抽出および，メソッドの引き上げリファクタリングの検出に使用するメソッド間の類似度を計算する方法を説明する．提案手法は，自然言語処理の分野で文書間の類似度を計算するのに利用される $2-shingles$ [13] を用いてメソッド間の類似度を計算する．Biegel らはコー

```
1  Input  : C, ;  /* a set of classes modified in r_{i+1} */
2  r_i, ;          /* a revision before the change */
3  r_{i+1} ;       /* a revision after the change */
4  Output: R ; /* a set of extract method candidates */
5  R := ∅;
6  foreach c ∈ C do
7      foreach m_a ∈ addedMethod(c, r_i, r_{i+1}) do
8          body_a := getBody(m_a) ;
9          foreach (m_{c,i}, m_{c,i+1}) ∈ changedMethod(c, r_i, r_{i+1}) do
10             added := getAddedLines(m_{c,i}, m_{c,i+1}) ;
11             deleted := getDeletedLines(m_{c,i}, m_{c,i+1}) ;
12             if ∃ m_a ∈ getCallee(added) then
13                 sim := calculateSimilarity(body_a, deleted) ;
14                 R.push((m_a, m_{c,i+1}, sim)) ;
15             end
16         end
17     end
18 end
```

図 2 メソッド抽出リファクタリングの候補の検出
Fig. 2 Detection of candidates for Extract Method.

```
1  Input  : M_a, ;   /* a set of methods added in r_{i+1} */
2  M_d, ;            /* a set of methods deleted in r_{i+1} */
3  C_{r_{i+1}} ;     /* a set of classes exists in r_{i+1} */
4  Output: R ; /* a set of pull up method candidates */
5  R := ∅;
6  foreach m_a ∈ M_a do
7      c_a := getClass(m_a) ;
8      body_a := getBody(m_a) ;
9      foreach m_d ∈ M_d do
10         c_d := getClass(m_d) ;
11         if c_a is super class of c_d then
12             if c_d ∈ C_{r_{i+1}} then
13                 body_d := getBody(m_d) ;
14                 sim := calculateSimilarity(body_a, body_d) ;
15                 R.push((m_a, m_d, sim)) ;
16             end
17         end
18     end
19 end
```

図 3 メソッドの引き上げリファクタリングの候補の検出
Fig. 3 Detection of candidates for Pull Up Method.

ド間の類似度を 3 種類定義し，リファクタリングの検出性能を比較している [14]．そのうち 2 種類の類似度はトークン列および抽象構文木を用いたコードクローン検出手法に基づいてそれぞれ定義され，コードクローン検出ツールを用いて計測されている．また，残り 1 種類の類似度の定義中には $2-shingles$ が用いられている．彼らの調査によると，いずれの類似度を用いた場合も精度，再現度ともに大きな差異はない．そこで提案手法では，これらの類似度のうち計算速度が最も速いと報告されている $2-shingles$ を採用した．

$2-shingles$ とは文書の先頭から単語を順に 2 個ずつ取り出し，それらの組を要素とする集合のことをいう[*2]．メソッド間の類似度を計算する際は，コード断片を字句解析した結果得られるトークン列を文書，各トークンを単語として扱う．具体的な例として，次に示すコード断片を考える．

```
int a = i++ + this.getConst();
```

このコード断片は字句解析を行うと以下のトークンに分解される．

$$\{int, a, =, i, ++, +, this, ., getConst, (,), ;\}$$

そして，これらのトークンから次の $2-shingles$ が得られる．

$$\{\{int, a\}, \{a, =\}, \{=, i\}, \{i, ++\}, \{++, +\}, \{+, this\},$$
$$\{this, .\}, \{., getConst\}, \{getConst, (\}, \{(,)\}, \{), ;\}\}$$

ある文章 s の $2-shingles$ を $SH(s)$ と定義する．この $2-shingles$ を用いて，文書 s_1，s_2 間の類似度 $similarity(s_1, s_2)$ は 0 から 1 の実数値として次の式で計算される．

$$similarity(s_1, s_2) = \frac{|SH(s_1) \cap SH(s_2)|}{|SH(s_1) \cup SH(s_2)|}$$

提案手法では，メソッド抽出リファクタリングにおいては文書 s_1 を $targetMethod$ から削除されたコード断片，文書 s_2 を $extractedMethod$ に追加されたコード断片として類似度を計算する．メソッドの引き上げリファクタリングでは，$sourceMethod$ および $destinationMethod$ 本体をそれぞれ s_1，s_2 とし，類似度を計算する．

4. 適用実験

提案手法の有効性を検出精度および，検出時間の観点から評価するために，提案手法をツールとして実装し（以降，提案ツールと表記する）[*3]，従来手法との比較を行った．提案ツールは Eclipse JDT を用いて作成した構文解析器を使用して Java 言語のソースコードを構文解析し，構文情報をディレクトリ構造へ変換する．また，類似度の計算およびメソッド呼び出し情報の取得に必要な字句解析器と構文解析器の作成には Pyrem Torq[*4]を利用した．提案手法の比較対象として，以下に示す理由から Ref-Finder および UMLDiff を採用した．

- 実装がツールとして公開されており，入手可能．
- 提案ツールと同じく Java 言語を対象としている．
- リファクタリング検出結果を CSV や XML などのデータ形式に書き出し可能．

適用実験に用いたオープンソースプロジェクトの Columba[*5]と jEdit[*6]の概要を**表 2** に，実験の概要を**図 4** に示す．適用実験は以下の手順で行った．

準備：Columba への適用結果を用いた閾値の決定
手順 1：jEdit のリリースバージョンを対象としたリファクタリング検出
手順 2：jEdit の全リビジョンを対象としたリファクタリング検出

準備として，提案ツールを Columba に適用し，その結果を用いて手順 1，手順 2 にて提案ツールが使用する類似度の閾値を決定した（4.1 節にて詳述）．

手順 1 においては jEdit のバージョン 4 から 4.5 までの 6 つのリリースバージョンを対象として，提案ツール，Ref-Finder，UMLDiff を用いてそれぞれリファクタリング検出を行った[*7]．また，各ツールの検出時間を計測した．提案ツールはリポジトリを入力として構文情報リポジトリを構築し，リファクタリングを検出する．そのため，対象リリースバージョンに対応するリビジョンのみを含んだリポジトリを作成し，入力とした．以降，提案ツール，Ref-Finder，UMLDiff の検出結果をそれぞれ REF_{k1}，REF_r，REF_u と表記する．次に，得られた検出結果の統合（和集合の作成）を行った．その際，$targetMethod$ と $extractedMethod$ それぞれについて，メソッドが属するクラスの完全修飾名，メソッド名，引数が一致するものを同一の結果として扱った[*8]．その後，すべての検出結果について手作業で誤検出かどうか確認し，各ツールの精度と再

[*2] 単語を w 個ずつ取り出し，要素とする集合を $w-shingles$ と呼ぶ．$2-shingles$ は $w=2$ における集合である．
[*3] 提案ツールは以下のページから入手可能である．
https://github.com/niyaton/kenja
[*4] https://github.com/tos-kamiya/pyrem_torq
[*5] http://sourceforge.net/projects/columba/
[*6] http://www.jedit.org
[*7] 提案ツールおよび Ref-Finder の実行には Xeon E5650 2.67 GHz を 2 基，メモリを 16 GB 搭載した OS X 計算機を使用した．また，UMLDiff の実行には Xeon E5540 2.53 GHz を 2 基，メモリを 12 GB 搭載した Linux サーバを使用した．
[*8] Ref-Finder はメソッドの引数に関する情報を出力せず，メソッド名のみを出力する．そのため，Ref-Finder の結果については，第 1 著者が手作業で引数情報の対応付けを行った．その際，リリースバージョン間の差分情報と出力結果のクラスの完全修飾名およびメソッド名を基に文献 [9] に記述された論理式を満たすメソッドの組を探索した．なお，対応すると考えられるメソッドのオーバロードが複数存在した場合は対応付け不可能と判断し，検出結果から除外した．

表 2 対象プロジェクトの概要
Table 2 Summary of target projects.

プロジェクト名	種類	開発期間	総リビジョン数	最終 LOC
Columba	メールクライアント	2006/7/9–2012/5/30	328	192,520
jEdit	テキストエディタ	2002/4/12–2012/1/30	4475	177,945

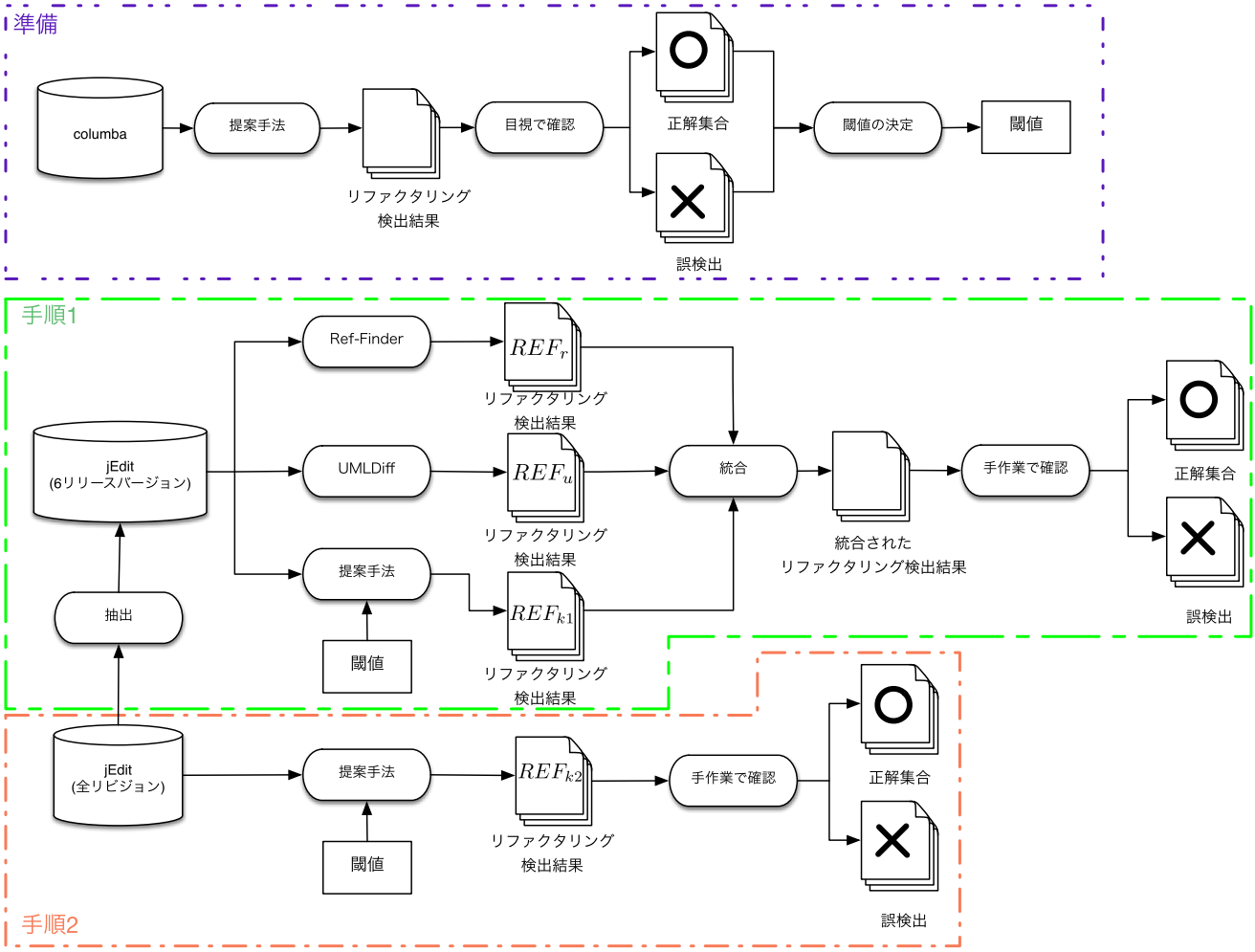


図 4 実験概要
Fig. 4 Overview of experiment.

現度を求めた．

手順 2 においては手順 1 で対象とした 6 リリースバージョン間に行われたすべての変更履歴（4,475 リビジョン）から，提案ツールを用いてリファクタリング検出を行った．また，手順 1 と同様にツールの検出時間を計測した．提案ツールの入力には jEdit プロジェクトが公開しているリポジトリを使用し[*9]，検出結果から対象期間に該当する結果のみを抽出した．そして，この検出結果（REF_{k2} と表記する）について手作業で誤検出かどうかを確認し，提案手法の精度を求めた．

[*9] http://sourceforge.net/p/jedit/jEdit.bak/にて取得可能な Git リポジトリを使用した．

4.1 閾値の決定

既存手法との比較を行うにあたり，Columba の全リビジョン間に提案ツールを適用し，その結果を用いて類似度の閾値を決定した．適用の結果，メソッド抽出リファクタリングの候補が 56 件，メソッドの引き上げリファクタリングの候補が 114 件得られた．これらの候補に対してリファクタリングであるかどうか手作業で確認を行ったところ，それぞれ 13 件と 6 件のリファクタリングが正しく検出できていることが分かった．

次に，各リファクタリング候補の類似度に着目し，最も多くの正解を検出しつつ，誤検出を少なくすることができる値を閾値とした．その閾値は，メソッド抽出リファクタリングについては 0.300，メソッドの引き上げリファクタリングについては 0.895 である．

表 4 jEdit への適用結果（リリースバージョン間でのリファクタリング検出）
Table 4 Result of refactoring detection from release versions in jEdit.

	提案手法（REF_{k1}）				Ref-Finder（REF_r）				UMLDiff（REF_u）				$REF_{k1} \cup REF_r \cup REF_u$	
	TP	FP	P	R	TP	FP	P	R	TP	FP	P	R	TP	FP
メソッド抽出	143	3	0.98	0.44	62	70	0.47	0.19	117	37	0.76	0.36	206	117
メソッドの引き上げ	0	0	0.00	0.00	0	1	0.00	0.00	0	0	0.00	0.00	0	1

表 3 リファクタリング検出にかかった時間
Table 3 Time for detecting refactoring.

	ツール	検出時間（分）
手順 1	Ref-Finder	35
	UMLDiff	4,500
	提案ツール	$3^{\dagger}+4^{\ddagger}=7$
手順 2	提案ツール	$55^{\dagger}+15^{\ddagger}=70$

† 構文情報リポジトリの構築時間
‡ 構文情報リポジトリからのリファクタリング検出時間

4.2 精度と再現度

提案手法と，既存手法によるリファクタリング検出の結果を合わせることで，精度と再現度を求める．ここで，本稿における精度と再現度の定義について説明する．本稿では，Bellon らがコードクローン検出ツールどうしの性能を比較する際に用いた評価方法と同様の方法で精度と再現度を計算する [15]．説明のために，先述した REF_{k1} と REF_r，REF_u を用いてツール間の比較する場合を考える．この場合は初めに，各ツールから得られたリファクタリング検出結果の和集合 $REF_{k1} \cup REF_r \cup REF_u$ を作成する．次に，この和集合の各要素について手作業で誤検出かどうかをそれぞれ確認する．この確認結果を用いて，精度 P と再現度 R はツールごとに以下の式で計算される．

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}$$

ここで，TP（True Positive）は対象のツールが正しく検出できたリファクタリングの数を，FP（False Positive）は誤ってリファクタリングとして検出した変更の数を表す．また，FN（False Negative）は他のツールは正しく検出したが，対象のツールが検出しなかったリファクタリングの数を表す．

4.3 適用結果

4.3.1 検出時間の比較

手順 1 および手順 2 において各ツールがリファクタリング検出にかかった時間を**表 3** に示す．表中の提案ツールの検出時間については，構文情報リポジトリの構築にかかった時間，構文情報からリファクタリングを検出するのにかかった時間とそれらの合計を記載している．表 3 からは，提案手法はリリースバージョン間からのリファクタリング検出において，既存手法よりも高速にリファクタリング検出が可能なことが分かる．また，リビジョン数の多い開発履歴からも実用的な時間でリファクタリング検出を行うことができる．

4.3.2 精度と再現度の比較

リリースバージョンを対象とした検出

手順 1 により得られたリファクタリング検出の結果と各ツールの精度と再現度を**表 4** に示す．メソッド抽出リファクタリングの検出においては，提案ツールの精度と再現度は 0.98，0.44 であった．各ツールの結果と比較すると，提案手法はリリースバージョン間のリファクタリング検出を従来手法より正確かつ数多く行えることが分かる．また，メソッドの引き上げリファクタリングについては，いずれのツールにおいてもリリースバージョン間でのリファクタリング検出ができなかった．表 4 の結果について，メソッドの引き上げリファクタリングに関しては Ref-Finder から REF_r として 1 件のリファクタリングが検出されたが誤検出であった．提案ツールからは閾値を超える候補が検出されなかった．また，REF_u は UMLDiff の検出結果から，2.3 節で述べた条件を満たしていなかったものを除外した結果 0 件となった[*10]．

細粒度リファクタリング検出（全リビジョンを対象とした検出）

手順 2 により得られたリファクタリング検出の結果と提案ツールの精度を**表 5** に示す．精度はメソッド抽出リファクタリングの検出については 0.99，メソッドの引き上げリファクタリングの検出については 1.00 であった．この結果より，提案ツールはメソッド抽出リファクタリングの細粒度リファクタリング検出において，検出精度がリリースバージョン間での検出と変わらないことが分かる．また，305 件とリリースバージョン間の倍以上のリファクタリングを検出できていることが分かる．つまり，提案手法を用いて細粒度リファクタリング検出を行うことでバージョン間でのリファクタリング検出よりも多くの事例を収集可能である．加えて，細粒度リファクタリング検出は，リリー

[*10] 具体的には，UMLDiff がメソッドの引き上げリファクタリングとして検出した結果のうち，106 件が条件 B-2 に反して変更前の時点で移動先のクラスを継承していなかった．また，4 件が条件 B-3 に反して変更後に移動元になったクラスが削除されていた．条件 B-2 に反した例としては，変更後にのみ移動元と移動先のクラスに継承関係が存在した．これは，リリースバージョン間の特定のリビジョンにおいて継承関係を追加し，後のリビジョンにおいてメソッドの引き上げリファクタリングを実施した可能性を示唆している．

表 5 jEdit への適用結果（細粒度リファクタリング検出）
Table 5 Result of refactoring detection from all revisions in jEdit.

	提案手法（REF_{k2}）		
	TP	FP	P
メソッド抽出	356	5	0.99
メソッドの引き上げ	21	0	1.00

スバージョン間で検出できなかったメソッドの引き上げリファクタリングも検出することができている．

5. 考察

5.1 細粒度リファクタリング検出の計算量

適用実験の結果から，リリースバージョン間を対象とした場合に提案手法が従来手法よりも高速にリファクタリング検出を実施できることが分かる．各ツールの検出時間が検出の対象となる差分の大きさによって大きく変化しないと仮定すると，細粒度リファクタリングを行った場合もリリースバージョン間での結果と同様に提案手法が従来手法よりも高速であると考えられる．しかし，適用実験では従来手法を用いて細粒度リファクタリング検出を実施していないため，提案手法によりどの程度の高速化が実現できているのか定量的な比較ができない．従来手法の両ツールはともに Eclipse のプラグインとして実装されている．そのため，実行に際して各リビジョンを Eclipse 上に読み込んだ上で依存関係のあるライブラリの設定などの準備を手作業で行う必要がある．これにより，現実的な作業時間で細粒度リファクタリング検出を実施できない．そこで，細粒度リファクタリング検出におけるリビジョン数に対する計算量の変化について考察を行う．

計算量を議論するにあたり，総リビジョン数 R のリポジトリを分析する場合を考える．ここで，構文解析（2.1 章における手順 1）にかかる計算量を $O(X)$，構文解析の差分の解析（手順 2）にかかる計算量を $O(Y)$，パターンマッチングによるリファクタリングの検出（手順 3）にかかる計算量を $O(Z)$ とする．これらの手順には順序関係はあるが手続きはそれぞれ独立である．なお，提案手法と既存手法を比較した場合，手順 1 と手順 2 に関しては計算時間の削減するための工夫をしているが手順 3 については従来手法と同様の手順で行っている，そのため，本稿では $O(X)$ と $O(Y)$ についてのみ議論する．

まず，構文解析にかかる計算量 $O(X)$ を提案手法と従来手法それぞれの場合で求める．リファクタリング検出においては，構文解析はソースコードファイルを単位として実施される．そこで，あるリビジョン r において追加されたファイル数を返す関数 $\delta_a(r)$，変更されたファイル数を返す関数 $\delta_m(r)$ を考える．ここでは，追加されたファイルも変更されたファイルの一部として扱う．リビジョン r において変更されたファイルのうち，リビジョン $r-1$ にも存在したファイルの割合を $\eta_m(r)$ とすると，$\delta_a(r)$ は $\delta_m(r)$ と $\eta_m(r)$ を用いて次のように表される．

$$\delta_a(r) = \delta_m(r)(1 - \eta_m(r))$$

提案手法は 3 章で述べたように，各リビジョンにおいて変更されたファイルのみ構文解析を行う．そのため，総リビジョン数 R のリポジトリに対して，提案手法が構文解析する必要のあるファイル数 $\widetilde{file}(R)$ はこれらの関数を用いて次の式で表される．

$$\widetilde{file}(R) = \sum_{r=0}^{R} \delta_m(r)$$

一方，従来手法は各リビジョンにおけるすべてのファイルを逐一構文解析する．そのため，従来手法が解析する必要のあるファイル数は，リビジョン r におけるファイル数を返す関数 $file(r)$ を用いて次のように表される．

$$\sum_{r=0}^{R} file(r) = \sum_{r=0}^{R}\sum_{i=0}^{r} \delta_m(i)(1 - \eta_m(i))$$

提案手法および従来手法の計算量は，$\delta_m(r)$ と $\eta_m(r)$ が分析対象の総リビジョン数 R に対してどのように変化するかによって決まる．本稿では，**表 6** に示す関数で $\delta_m(r)$ と $\eta_m(r)$ をそれぞれモデル化し，モデルの組合せによって計算量がどのように変化するか考察する．

表 6 の各式は，ファイルの追加数と変更数が，プロジェクトの進行に従ってどのように変化するかをモデル化している．まず，プロジェクトの進行にかかわらず，変更および追加されるファイル数がほぼ一定である場合を考える．このとき，リビジョンあたりの平均変更ファイル数および平均追加ファイル数をそれぞれ α，β として，$\delta_m(r)$ と $\eta_m(r)$ をモデル化した．

次に，プロジェクトの進行に従って，変更されるファイル数が次第に減少する場合を考え，$\delta_m(r)$ を $ke^{-\lambda r}$ としてモデル化した．ここで，k および λ はプロジェクトの規模や成熟速度によってそれぞれ決まる定数である．また，プロジェクトの進行に従い，変更ファイルに占める既存ファイルの割合が増加する場合を考え，$\eta_m(r)$ を $1 - le^{-\gamma r}$ としてモデル化した．ここで，l および γ はプロジェクトの規模や成熟速度によってそれぞれ決まる定数である．このモデルはプロジェクトが成熟すると新規ファイルの作成が減少し，既存ファイルの保守が主になることを表している．

これらのモデルを用いて提案手法と従来手法がそれぞれ構文解析にかかる計算量を求めた結果を**表 7** に示す．表中の各セルはそれぞれのモデルを採用した場合の提案手法と従来手法の計算量を表している．この表より，いずれのモデルを採用した場合でも提案手法が既存手法よりも少ない計算量で構文解析を実施できることが分かる．特に，分析

表 6 ファイル変更のモデル化
Table 6 File change models.

	減少	一定	増加
$\delta_m(r)$	$ke^{-\lambda r}$	α	–
$\eta_m(r)$	–	β	$1 - le^{-\gamma r}$

表 7 提案手法および既存手法の計算量
Table 7 Orders of computation on proposed method and existing method.

$\eta_m \backslash \delta_m$	減少		一定	
	提案	既存	提案	既存
一定	$O(1)$	$O(R)$	$O(R)$	$O(R^2)$
増加	$O(1)$	$O(R)$	$O(1)$	$O(R)$

対象のリビジョン数が多く，プロジェクトが $\delta_m(r) = \alpha$，$\eta_m(r) = \beta$ のモデルに従う場合に，提案手法を用いることで従来手法より高速に細粒度リファクタリング検出を実施可能であるといえる．

本節で考察したモデルのほかに，プロジェクトの進行に従って，修正されるファイル数が増加するモデルと新規作成ファイルが増加するモデルがそれぞれ考えられる．しかし，このようなモデルに合致するソフトウェアプロジェクトは一般的でないと考えられるので本稿では扱わない．

$O(X)$ と同様に，構文要素の対応付けの計算量 $O(Y)$ について考察する．従来手法は，各リビジョンのスナップショットから得られたすべての構文要素に対して，リビジョン間での対応付けを行っている．一方，提案手法は変更されなかったファイルから同一の構文要素が得られる特性を利用し，変更の行われたファイルに関する構文要素についてのみリビジョン間での対応付けを行っている．構文要素の数は構文解析の対象となるファイル数に依存すると考えられるため，$O(Y)$ についても $O(X)$ と同様の高速化がなされていることが考えられる．

5.2 メソッド抽出リファクタリングの検出精度の向上

適用実験の結果から，提案手法はメソッド抽出リファクタリングに関して，従来手法よりも正確かつ数多くのリファクタリングを検出できることが分かる．

提案手法はメソッド間の類似度を計算する際に $2 - shingles$ を用いている．一方，従来手法である UMLDiff は抽象構文木の構造に基づく類似度を算出する．先に述べたように，Biegel らは $2 - shingles$ を用いた類似度と抽象構文木に基づく類似度のどちらをメソッド間の類似度の算出に利用したとしても，検出の精度に大きな差は生じないと報告している [14]．提案手法は，メソッド抽出の対象となったメソッド（$targetMethod$）から削除されたコード断片と抽出されたメソッド（$extractedMethod$）全体のコード断片の類似度を計算する．また，Ref-Finder は $targetMethod$ 全体のコード断片と $extractedMethod$ 全体のコード断片から類似度を計算する．そのため，従来手法は抽出したコード断片に対応するはずのないコード断片を含んで類似度を計算してしまう可能性がある．たとえば，メソッド抽出の対象となったメソッド全体の行数に対して，新たに抽出されたメソッドの行数が少ない場合，提案手法の類似度と比べて従来手法の類似度が小さくなると考えられる．このことが原因で，提案手法に比べて Ref-Finder の精度が落ちているものと考えられる．

5.3 リファクタリング検出の粒度

リファクタリング検出の粒度の妥当性について議論する．適用実験では隣接するすべてのリビジョン間からリファクタリング検出を行った結果，リリースバージョン間からの検出と比較して多くのリファクタリング適用事例を収集することができた．版管理システムを用いた開発において，1 つのコミットは単一の作業単位で行われるべきとされている [16]．そのため，jEdit の開発者らがこの方針に従って変更を行っているならば，開発中に実施されたリファクタリングを漏れなく検出できていると考えられる．一方で，複数のリビジョンをまたいでリファクタリングが実施されている例は収集できていないと考えられる．このようなリファクタリングは，検出を行うリビジョンの間隔を変更した場合の結果と組み合わせることで検出可能であると考えられるが，適切な粒度の調整や，重複する検出結果の扱いなどを考慮したうえで実験を行い確認する必要がある．

5.4 類似度の閾値

適用実験で決定した類似度の閾値を，他のプロジェクトを対象とするリファクタリング検出に用いることについて議論する．適用実験では Columba の開発履歴を用いて決定した閾値を用いて jEdit からリファクタリング検出を行った．実験結果より，メソッド抽出リファクタリングについては精度，再現度ともに他の手法と比較して高い値が得られている．また，メソッドの引き上げリファクタリングについては細粒度リファクタリング検出の場合に精度が 1.00 と高い値が得られている．このことから，適用実験で用いた閾値は今回対象としたプロジェクト以外の開発履歴からの検出においても有効である可能性がある．同一のプログラミング言語を対象とした場合，メソッド単位の類似度はプロジェクト間で特性が大きく変化することはないと考えられる．そのため，ともに Java で記述された Columba と jEdit において同一の閾値が有効であったと考えられる．一方，今回の閾値が Java 以外の言語で記述されたプロジェクトに対して有効かどうかは今後確認を行う必要がある．

提案手法を他のプロジェクトに適用する場合にはリファクタリング検出の目的に応じて閾値を設定するのが望ましい．たとえば，リファクタリングの適用事例をできるだ

け多く収集し，リファクタリングが実施される条件を調査したい場合には誤検出が少ない方が良いといえる [5]．一方，ある期間に実施されたリファクタリングを，網羅的にレビューする場合には検出の漏れが少ない方が良いといえる．提案手法で用いた閾値には検出結果の精度と再現度についてトレードオフの関係がある．ここでの精度と再現度とは，閾値を 0 に設定した場合，つまり閾値による候補の絞り込みを行わなかった場合の検出結果から作成した正解集合と，閾値による絞り込みを行った結果を用いて計算したものである．閾値を 0 とすると提案手法で検出可能なすべてのリファクタリングを得ることができるが，誤検出を含むようになる．閾値を高く設定することで誤検出を除外できるが，リファクタリングとそれ以外の変更が混在する場合に類似度が低く計算され，結果としてリファクタリングとして検出されない可能性が考えられる．このようなトレードオフを考慮したうえで，対象プロジェクトの一部を用いて，閾値の変化による精度と再現度の変化を確認し，目的に応じた閾値の調整を行うのが良いと考えられる．

5.5 他のリファクタリングパターンへの適用可能性

適用実験において対象とした 2 つのリファクタリングパターン以外への適用可能性について議論する．メソッド抽出は，コードの抽出を行う代表的なリファクタリングパターンである．また，Template Method の形成のようなより複雑なパターンの一部として用いられている基本的なパターンでもある．したがって，このようなリファクタリングを検出する際の手順の一部として，メソッド抽出の検出は十分有効であると期待できる．同様に，メソッドの引き上げは，クラス間でメソッドの移動を行う代表的なリファクタリングパターンである．そのため，メソッドの引き上げに対して示された有効性は，クラス間でメソッドの移動を行う他のリファクタリングパターン（スーパクラスの抽出など）に対しても期待できると考えられる．

Prete らは，リビジョン間においてリファクタリングパターンが適用されているかどうかを判定する条件を論理式で記述している [9]．彼らは，63 個のリファクタリングパターンに対して条件を記述しており，これら条件の判定を実現するよう提案手法を拡張することで，メソッド抽出やメソッドの引き上げ以外のリファクタリングパターンに対しても検出を行うことができると考えられる．彼らが記述した条件を表す論理式に用いられている述語のほとんどは，3.1 節で述べた構文情報リポジトリの構築において記録する構文情報，および 3.2 節において用いるメソッド呼び出しに関する情報である．そのため，彼らが記述した条件の判定を行い，他のリファクタリングパターンに対応するよう提案手法を拡張することは，多くのリファクタリングパターンにおいて容易であると考えられる．ただし，現状の提案手法ではメソッド内の変数宣言に関する構文情報を取得していない．そのため，一時変数の分離などのメソッド内で行われる変数宣言に関するリファクタリングパターンの検出ついては，構文情報の取得部分に拡張が必要である．

6. おわりに

本稿では構文情報リポジトリを用いて細粒度リファクタリング検出を実施するための手法を提案した．メソッド抽出およびメソッドの引き上げリファクタリングについて手法を実装したツールと，既存手法である Ref-Finder および UMLDiff を jEdit に適用し，検出時間および，精度と再現度について比較実験を行った．その結果，提案手法を用いて細粒度リファクタリング検出を行うことで，より多くのリファクタリングを正確かつ高速に検出できることが確認できた．

本稿では精度と再現度を求めるにあたり，第 1 著者が手作業で誤検出の確認を行った．そのため，偏った判断に基づいて精度と再現度を求めている可能性が適用実験に対する妥当性への脅威として考えられる．この問題の解決策として，複数人による手作業での誤検出確認や，分析対象プロジェクトの開発者自身に検出結果が正しいかどうかを判断して貰う方法が考えられるが，いずれの場合も個人の主観を完全に排除し，客観的な判断を行うことは難しい．

今後の課題として，より多くの種類のリファクタリングを提案手法を用いて検出できるように技術を確立することがあげられる．また，提案手法により得られる細粒度なリファクタリングの適用事例に基づいて，リファクタリングの効果の定量評価や，より有用なリファクタリング支援手法の提案が行われることが期待される．

参考文献

[1] Fowler, M.: *Refactoring: Improving the design of exsiting code*, Addison Wesley (1999).

[2] Kim, M., Cai, D. and Kim, S.: An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution, *Proc. ICSE*, pp.151–160 (2011).

[3] Ratzinger, J., Sigmund, T. and Gall, H.C.: On the relation of refactorings and software defect prediction, *Proc. MSR*, pp.35–38 (2008).

[4] Fujiwara, K., Fushida, K., Yoshida, N. and Iida, H.: Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives, *Proc. PROFES*, pp.313–323 (2013).

[5] Choi, E., Yoshida, N. and Inoue, K.: What kind of and how clones are refactored?: A case study of three OSS projects, *Proc. WRT*, pp.1–7 (2012).

[6] Tu, Q. and Godfrey, M.W.: An Integrated Approach for Studying Architectural Evolution, *Proc. IWPC*, pp.127–136 (2002).

[7] Hata, H., Mizuno, O. and Kikuno, T.: Historage: Fine-grained Version Control System for Java, *Proc. IWPSE-EVOL*, pp.96–100 (2011).

[8] Xing, Z. and Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries, *Proc. WCRE*, pp.263–274 (2006).

[9] Prete, K., Rachatasumrit, N., Sudan, N. and Kim, M.: Template-based Reconstruction of Complex Refactorings, *Proc. ICSM*, pp.1–10 (2010).
[10] Xing, Z. and Stroulia, E.: UMLDiff: An algorithm for object-oriented design differencing, *Proc. ASE*, pp.54–65 (2005).
[11] Prete, K., Rachatasumrit, N. and Kim, M.: A Catalogue of Template Refactoring Rules, Technical Report UTAUSTINECE-TR-041610, The University of Texas at Austin (2010).
[12] Kim, M. and Notkin, D.: Program Element Matching for Multi-Version Program Analysis, *Proc. MSR*, pp.58–64 (2006).
[13] Broder, A.Z.: On the Resemblance and Containment of Documents, *Proc. SEQUENCES*, pp.21–29 (1997).
[14] Biegel, B., Soetens, Q.D., Hornig, W., Diehl, S. and Demeyer, S.: Comparison of similarity metrics for refactoring detection, *Proc. MSR*, pp.53–62 (2011).
[15] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol.33, No.9, pp.577–591 (2007).
[16] Berczuk, S. and Appleton, B.: *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, Addison Wesley (2002).

藤原 賢二 （正会員）

平成 22 年大阪府立工業高等専門学校総合工学システム専攻修了．平成 27 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．リファクタリングの適用履歴分析，プログラミング教育支援に関する研究に従事．

吉田 則裕 （正会員）

平成 16 年九州工業大学情報工学部知能情報工学科卒業．平成 21 年大阪大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．平成 22 年奈良先端科学技術大学院大学情報科学研究科助教．平成 26 年より名古屋大学大学院情報科学研究科付属組込みシステム研究センター准教授．博士（情報科学）．コードクローン分析手法やリファクタリング支援手法に関する研究に従事．

飯田 元 （正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 7 年奈良先端科学技術大学院大学情報科学センター助教授．平成 17 年同大学情報科学研究科教授．博士（工学）．