社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

構文情報を付加したリポジトリによるメソッド抽出リファクタリングの検出

藤原　賢二†　　吉田　則裕†　　飯田　　元†

† 奈良先端科学技術大学院大学 情報科学研究科　〒 630–0192 奈良県生駒市高山町 8916–5
E-mail: †{kenji-f,yoshida}@is.naist.jp, ††iida@itc.naist.jp

あらまし　近年，ソフトウェア開発履歴を分析することでリファクタリングが品質に与える影響を明らかにすることを目的とした研究が多数実施されている．このような分析にあたっては，実施されたリファクタリングの種類と実施時期を把握する必要がある．本稿では構文情報を付加したリポジトリを用いることでリファクタリングの実施履歴を高速に復元する手法を提案する．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．その結果として，既存手法である UMLDiff と比較してより高精度にリファクタリングの実施履歴を復元可能なことを確認した．

キーワード　リファクタリング，リポジトリマイニング，細粒度分析

Detecting Extract Method Refactoring from Repository with Syntactic Information

Kenji FUJIWARA†, Norihiro YOSHIDA†, and Hajimu IIDA†

† Graduate School of Information Science, Nara Institute of Science and Technology
Takayama-cho 8916–5, Ikoma-shi, Nara, 630–0192 Japan
E-mail: †{kenji-f,yoshida}@is.naist.jp, ††iida@itc.naist.jp

Abstract　Recently, several research have tackled to reveal the effects of refactorings in the software development. In order to reveal the effects, it is important to detect performing of refactorings from the history of the software development. In this research, we propose a fast refactoring detection method which use the repository with syntactic information. The result of comparison with the UMLDiff algorithm shows our proposed method provide high recall and high precision.

Key words　Refactoring, Mining software repositories, Fine-grained analysis

1. はじめに

リファクタリングとはソフトウェアの外部的な振る舞いを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，リファクタリングの効果を実証的に調査することを目的とした研究がいくつか報告されている．例えば，Ratzinger らはリファクタリングと欠陥の関係を調べており，ある期間にリファクタリングが多く実施された場合，後続の期間において欠陥の発生量が減少すると報告している [2]．また，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 [3]．これらの研究は，いずれも 5 件未満のプロジェクトを対象に実施されているが，より一般性の高い知見を得るためには，更に多くのプロジェクトを対象に分析を実施することが望ましい．そのためには次に挙げる特徴を持ったリファクタリング検出技術の研究が重要となる．

- 細粒度な（コミット単位での）リファクタリングの検出が可能
- 大規模なプロジェクトに適用可能
- 多数のプロジェクトに適用可能な可搬性とスケーラビリティを持つ

本稿では，これらの要求を満たすリファクタリング検出手法として，構文情報を付加したリポジトリを用いたリファクタリング検出手法を提案する．また，オープンソースソフトウェアであ

る jEdit を対象に既存手法である UMLDiff との比較を行った．

2. 背　　景

2.1 リファクタリング検出

Murphy-Hill らは，ソフトウェア開発中に実施されるリファクタリングに関する情報を収集する為のアプローチとして，次の 4 つのアプローチが有効であると述べている [4]．

（1） 開発者を監視し，リファクタリングの実施を記録する

（2） リファクタリングツールの使用を自動的に記録する

（3） 版管理システムのコミットログを分析する

（4） 版管理システムに記録されたソースコードの変更を解析する

ここで，アプローチ（1）, (2）を適用するためにはプロジェクト毎に準備が必要であるため，多くのプロジェクトからリファクタリングの実施履歴を収集しリファクタリングが品質に与える影響を分析するという目的には適さない．対照的に，ソフトウェア開発において広く利用されている版管理システムを用いたアプローチ（3）, (4）であれば多くのプロジェクトを対象とすることができると考えられる．しかし，開発者がリファクタリング活動をコミットログとして記録することは，実際にリファクタリングを行う頻度に比べて少ないことが分かっている [5]．そのため，版管理システムからリファクタリングの実施状況を把握する場合はアプローチ（4）を採用する必要があり，これまで数多くの手法が提案されている．本稿ではこのアプローチをもってリファクタリングの実施履歴を復元することをリファクタリング検出と定義する．

Xing らは任意のバージョン間におけるソースコードの設計情報の差分を求める UMLDiff アルゴリズム [6] を提案している．また，その差分情報を解析することでリファクタリングを検出する手法も提案しており，そのためのツールも公開されている [7]．以降，本稿では特に断りの無い限り 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．

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，あるメソッドから意味のある単位でコードを選択し，新たなメソッドとして抽出することを言う [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挙げられる．また，長すぎるメソッドを対象にコードの可読性を向上させることを目的とすることもある．

本稿で提案する手法は，メソッド抽出リファクタリングが以下の手順に沿って実施されることを前提としている．

手順 1　抽出の対象とするメソッドとコードを選択する．

手順 2　選択したコードを新たなメソッドとして抽出する．なお，メソッドは抽出対象のメソッドと同一のクラス内に作成する．

手順 3　必要に応じて抽出したメソッドに修正を加える．

手順 4　選択したコードを抽出元のメソッドから削除し，新たに作成したメソッドの呼び出しに置き換える．

3. 構文情報を付加したリポジトリからの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を入力として，各コミット間で実施されたメソッド抽出リファクタリングを検出する．初めに，分析対象となるソフトウェアの開発履歴が記録されたリポジトリを構文情報を付加したリポジトリに変換する．通常のリポジトリは，ソースコードファイル毎の変更情報（行の追加や削除）を提供するが，構文情報を付加したリポジトリはメソッドやクラス単位での変更情報を提供する．次に，これらの情報を基に各種リファクタリングの検出を行う．

以降，本節ではリポジトリに構文情報を付加する手順と，構文情報を付加したリポジトリからメソッド抽出リファクタリングを検出する手順について詳しく説明する．

3.1 構文情報を付加したリポジトリの構築

提案手法では，Hata らが提案している Historage と同様の方法で構文情報を付加したリポジトリを構築する [8]．Historage は Java 言語で記述されたソースコードの変更を，メソッド単位で追跡するためのリポジトリである．版管理システムの一種である Git(注1)はファイルおよびディレクトリ構造の差分を計算する機能を持つ．Historage はこの仕組みを利用して，コミット間における構文情報の差分を自動的に計算する．そのために，各コミットにおけるソースコードを構文解析し，得られた構文木を一定の規則に従ったディレクトリ構造に変換する．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の階層関係で表現し，メソッドの本体をファイルとして表現する．例として，図 1 に変換前のディレクトリ構造を，図 2 に変換後のディレクトリ構造を示す．

構文情報を付加したリポジトリの構築手順としては，各コミットにおけるソースコードに対してそれぞれ構文解析を行い，ディレクトリ構造の変換を行い，コミットする．この際，コミットには以下の情報が付加されている．

（1） クラスのパッケージ情報

（2） クラスが保持するメソッド

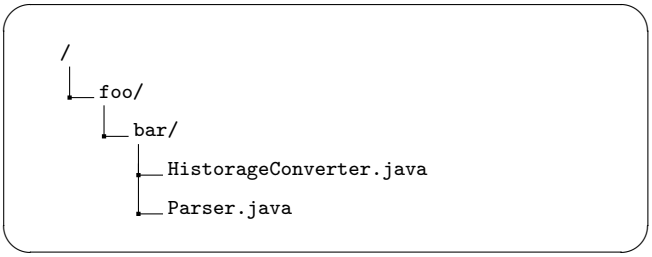

図 1　変換前のディレクトリ構造

(注1)：http://git-scm.com/

（3） クラスが保持するフィールド

（4） メソッドの識別子

（5） メソッドの本体

（6） メソッドのパラメータ

構文木を変換したディレクトリ構造は，構文木の基になったソースコードファイルと一対一に対応するため，同一のソースコードファイルからは同一のディレクトリ構造が得られる．そのため，各コミットを処理する際は，変更のあったソースコードファイルのみ新規に構文解析を行い，変更の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流用することが可能である．これにより，提案手法は既存の手法と比較して大幅な計算時間の短縮に成功している．

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，構文情報を付加したリポジトリのコミット同士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．提案手法は変更前のコミットと変更後のコミットを入力として，検出した各リファクタリングについて，以下の情報を出力する．

$$extractMethod := (commit, targetMethod, extractedMethod, simirality)$$

ここで，$commit$ はメソッド抽出が実施されたコミットを一意に特定する値，$targetMethod$ はメソッド抽出の対象となったメソッド，$extractedMethod$ はリファクタリングにより新たに作成されたメソッドを表す．また，$simirality$ は 3.3 節にて述べる方法で算出される，$targetMethod$ から抽出されたコードと $extractedMethod$ の類似度を表す．2.2 節で述べたように，複数のメソッドに跨がって存在する重複したコードを除去するために，メソッド抽出リファクタリングを行う場合がある．提案手法ではこのような場合，それぞれ独立したリファクタリングが実施されたものとする．仮に，二つのメソッドに跨がって存在する重複したコードが一つのメソッドとして抽出された場合，メソッド抽出リファクタリングが二回実施されたと判断する．

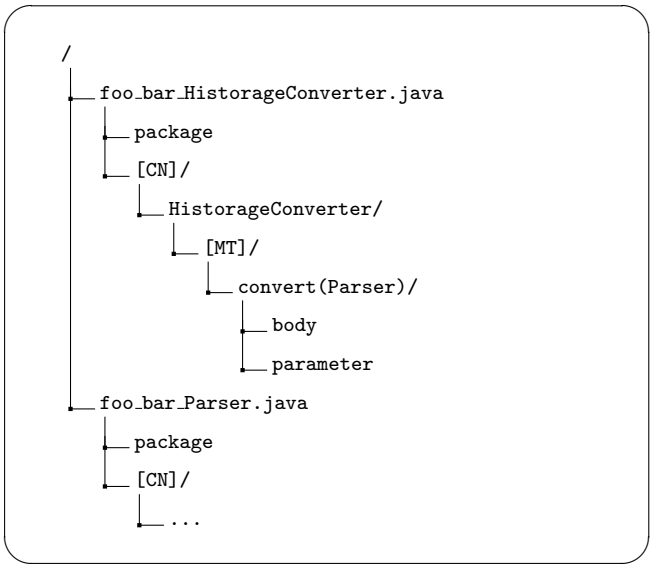

図 2 変換後のディレクトリ構造

変更前のコミットと変更後のコミットからメソッド抽出リファクタリングを検出する手順としては，まず，変更において新規に作成されたメソッドを見つけ，$extractedMethod$ の候補とする．次に，変更において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．そして，$targetMethod$ に追加された行に $extratedMethod$ が存在すれば，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．これらの手順を変更のあったクラス毎に行い，最後に閾値以上の類似度を持つ候補をリファクタリングとして検出する．Algorithm 1. にこれらの手順を擬似コードとして示す．

3.3 類似度の計算

本節では $targetMethod$ と $extractedMethod$ 間の類似度を計算する方法を説明する．提案手法では，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 shingles [9] を応用してメソッド間の類似度を計算する．shingles とは文章 s の先頭から単語を順に 2 個ずつ取り出し，取り出した単語の組を要素とする集合 $SH(s)$ ことを言う(注2)．shingles を用いて，文書 s_1，s_2 間の類似度 $similarity(s_1, s_2)$ は次の式で計算される．

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

提案手法ではメソッド間の類似度を計算するために，文章

```
Input  : C, /* a set of class modified in v_{i+1}      */
v_i, /* before version of the change                  */
v_{i+1} /* after version of the change                */
Output: R /* a set of extract method candidates       */
R := ∅;
foreach c ∈ C do
    foreach em ∈ addedMethod(c, v_i, v_{i+1}) do
        body = getBody(m) ;
        foreach (tm_i, tm_{i+1}) ∈ changedMethod(c, v_i, v_{i+1})
        do
            added := getAddedLines(em_i, em_{i+1}) ;
            deleted := getDeletedLines(em_i, em_{i+1}) ;
            if ∃ m ∈ getCallee(added) then
                sim := calculateSimirarity(body, deleted) ;
                R.push((m, em, sim)) ;
            end
        end
    end
end
```

Algorithm 1: Detect Extract Method Candidates

(注2)：厳密には，単語を w 個ずつ取り出し要素とする集合を $w-shingles$ と呼び，単語を 2 個ずつ取り出す場合に $2-shingles$ と呼ぶが，本稿では $2-shingles$ を shingles と呼称する．

s_1 を $targetMethod$ から削除されたコード断片，文章 s_2 を $extractedMethod$ に追加されたコード断片とし，字句解析によるトークンを単語として類似度を計算する．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 shingles が得られる．

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

4. 評　　価

提案手法をツールとして実装し(注3) 既存手法である UMLDiff との比較を行った．ツールは Eclipse JDT を用いて作成した軽量パーサを使用して Java で記述された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．ソースコードの構文解析はプロセッサ数に応じて並列計算を行うよう実装されている．また，類似度を計算する際に必要な字句解析器の作成にはリバースエンジニアリングツールのためのツールキット Remics [10] の一部である Pyrem Torq(注4) を利用した．

UMLDiff との比較はオープンソースソフトウェアである Columba (注5) と jEdit (注6) に提案手法と UMLDiff を適用し，得られた結果を比較することで行った．表 1 に Columba と jEdit の概要を示す．

4.1 閾値の決定

UMLDiff との比較を行うにあたり，まず，Columba に提案手法を適用し，その結果を基に提案手法が使用する類似度の閾値を決定した．Columba からは計 56 個のメソッド抽出リファクタリングの候補が得られた．これらの候補に対して目視でメソッド抽出リファクタリングであるかどうか確認を行ったところ，13 件のメソッド検出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．また，これらの候補が全て類似度 0.45 以上に存在することが分かった．次に，ツールが誤検出したリファクタリング候補に着目したところ，類似度 0.25 以上のリファクタリング候補が存在していなかったことが分かった．これらの情報を基に，類似度 0.25 から 0.45 の間に閾値を設定することで，誤検出である確率が極めて高いリファクタリング候補をフィルタリングできると考え，UMLDiff の際に使用する類似度を 0.3 に決定した．

4.2 UMLDiff との比較

4.1 節で決定した閾値 0.3 を使用して jEdit を分析対象とし，UMLDiff との精度（precision）と再現度（recall）の比較評価を行った．

表 2　UMLDiff を適用したバージョン

バージョン番号	タグ名
4	jedit-4-0-final
4.1	jedit-4-1-final
4.2	jedit-4-2-final
4.3	jedit-4-3
4.4	jedit-4-4-1
4.5	jedit-4-5-0

4.2.1 提案手法の jEdit への適用

Xeon E5540 2.53GHz を 2 基，メモリを 12GB 搭載した計算サーバを用いて jEdit のリポジトリを提案手法を実装したツールに入力として与えた．なお，ツールの実行は Xeon E5540 2.53GHz を 2 基，メモリ 12GB 搭載した計算サーバにて行った．その結果，構文情報を付加したリポジトリの構築に 1 時間 20 分，構築後のリポジトリからリファクタリングを検出するのに 6 分要した(注7)．

4.2.2 UMLDiff の jEdit への適用

UMLDiff は表 2 に示すバージョン 4 から 4.5 までの 6 リリースバージョンに適用した．表中のタグ名は Sourceforge.net にて配布されているソースコードのバージョンと対応するリポジトリのタグ名を表す．なお，UMLDiff は提案手法を実装したツールと同様の環境で独立して実行した．その結果，UMLDiff の実行には 3 日と 3 時間を要した．

4.2.3 検出結果の比較

類似度の閾値を 0.3 に設定した，提案手法を実装したツールからは 305 件，UMLDiff からは 208 件のリファクタリングが検出された．うち，76 件が両方のツールがリファクタリングとして検出され，全体としては 437 件のリファクタリングが検出された．

次に，これらの結果を用いて提案手法および UMLDiff の精度と再現度を調査した．まず，リファクタリング検出における精度と再現度について考える．提案手法が検出するリファクタリングを R_k，UMLDiff が検出するリファクタリングを R_u とすると，両ツールから得られるリファクタリング $Actual$ は次のように表される．

$$Actual = R_k \cap R_u$$

次に，$Actual$ について目視で誤検出かどうかを確認し，それぞれのツールの結果と $Actual$ の Confusion Matrix を考えると表 3 のようになる．これらの情報を用いてそれぞれのツールの精度（Precision）と再現性（Recall）は次の式で計算することができる．

(注3)：実装したツールは次のページから入手可能である：https://github.com/niyaton/kenja
(注4)：https://github.com/tos-kamiya/pyrem_torq
(注5)：http://sourceforge.net/projects/columba/
(注6)：http://www.jedit.org/

(注7)：入力には http://jedit.git.sourceforge.net/git/gitweb.cgi?p=jedit/jEdit;a=summary にて取得可能な Git リポジトリを使用した．そのため，計算時間には jEdit バージョン 4 以前とバージョン 4.5 以降の履歴からのリファクタリング検出に要した時間も含まれている

表 1　実験対象プロジェクトの概要

プロジェクト名	種類	開発期間	総コミット数	最終 LOC
columba	メールクライアント	2012/5/30 - 2006/7/9	328	192520
jEdit	テキストエディタ	2002/4/12 - 2012/1/30	4475	177945

表 3　リファクタリング検出における Confusion Matrix

		Tool Result	
		not detected	detected
Actual	not refactored	True negative (TN)	False positive (FP)
	refactored	False negative (FN)	True positive (TP)

表 4　jEdit への適用結果（ランダムに 100 件抽出）

提案手法			UMLDiff			Actual
R_k	TP	FP	R_u	TP	FP	
72	69	3	47	28	19	80

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$

今回，両手法のツールを jEdit に適用した結果得られた結果全てに対して，目視で誤検出であったかどうかを確認するのは時間の都合上現実的では無かったため，全体の結果からランダムに 100 件抽出したものを *Actual* とし，目視で誤検出かどうかの確認を行った．その結果を表 4 に示す．これらの結果から *Precision*，*Recall* についてそれぞれ提案手法は 0.96，0.86，UMLDiff は 0.60，0.35 が得られた．

5. 関 連 研 究

本研究で検出対象としたメソッド抽出リファクタリングの他に，メソッドの引き上げリファクタリングや，Template Method の形成など，メソッドへの操作を行うリファクタリングは数多くある．これらのリファクタリングを対象としたリファクタリング検出手法の多くは，メソッド同士の類似度を求め，手法の精度向上に役立てている．Biegel らはメソッド間の類似度を表す 3 つのメトリクスを用いて，リファクタリング検出を行い，各メトリクスにおけるリファクタリングの検出精度やパフォーマンスについて比較を行っている [11]．彼らの報告ではクローン検出技術を用いて計測したメトリクス，抽象構文木の情報を用いて計測したメトリクスと，文書間の類似度を求めるためのメトリクスをメソッドに対して適用したメトリクスを比較対象として使用している．3.3 節で説明した，本稿におけるメソッド間の類似度も彼らが使用している文書間の類似度を求めるためのメトリクスを基に計算している．

Prete らは論理プログラミングの概念をリファクタリング検出に導入した手法を提案している [12]．彼らの手法では，ソースコード中のクラスや，メソッド，それらの関係を論理プログラミングにおける述語として抽出し，リファクタリングを述語の組み合わせとして定義している．彼らの手法を実装した Ref-Finder は，現在最も多くの種類のリファクタリングを検出することが可能である．

6. まとめと今後の課題

本稿では構文情報を保持したリポジトリを構築することで，メソッド抽出リファクタリングを検出する方法を提案した．提案手法を実装したツールと既存のリファクタリング検出手法である UMLDiff をオープンソースソフトウェアである jEdit に適用し，精度と再現性について比較実験を行った．その結果，精度と再現性について提案手法を実装したツールが UMLDiff よりも優位であることが確認された．

また，UMLDiff の適用に要した計算時間を基に，提案手法を実装したツールと同じ粒度で UMLDiff が分析しようとした場合，単純計算で約 6 年かかることが分かる．このことから，提案手法を用いることで，UMLDiff ではできなかった細粒度での分析をより高精度に実施できることが示唆される．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利用して検出できるよう技術を確立することが挙げられる．そして，その結果を用いて様々なプロジェクトからリファクタリングの実施履歴を復元し，リファクタリングが品質に与える影響を詳細に分析できるようになると考える．

文　献

[1] M. Fowler, Refactoring: improving the design of exsiting code., Addison Wesley, 1999.

[2] J. Ratzinger, T. Sigmund, and H.C. Gall, “On the relation of refactorings and software defect prediction,” In Proc. the 5th Working Conference on Mining Software Repositories(MSR 2008), pp.35–38, 2008.

[3] M. Kim, D. Cai, and S. Kim, “An empirical investigation into the role of api-level refactorings during software evolution,” In Proc. the 33rd International Conference on Software Engineering(ICSE 2011), pp.151–160, 2011.

[4] E. Murphy-Hill, A.P. Black, D. Dig, and C. Parnin, “Gathering Refactoring Data: a Comparison of Four Methods,” Proceedings of the 2nd ACM Workshop on Refactoring Tools(WRT 2008), pp.1–5, 2008.

[5] E. Murphy-Hill, C. Parnin, and A.P. Black, “How We Refactor, and How We Know it,” In Proc. the 31st International Conference on Software Engineering(ICSE 2009), pp.287–297, 2009.

[6] Z. Xing and E. Stroulia, “UMLDiff: an algorithm for object-oriented design differencing,” In Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering(ASE 2005), pp.54–65, 2005.

[7] Z. Xing and E. Stroulia, “Refactoring Detection based on UMLDiff Change-Facts Queries,” In Proc. the 13th Working Conference on Reverse Engineering(WCRE 2006), pp.263–274, 2006.

[8] H. Hata, O. Mizuno, and T. Kikuno, “Historage: Fine-grained Version Control System for Java,” In Proc. the 12th International Workshop on Principles of Software Evolution and the 7th ERCIM Workshop on Software Evolution(IWPSE-EVOL 2011), pp.96–100, 2011.

[9] A.Z. Broder, “On the resemblance and containment of documents,” In Proc. the Compression and Complexity of Sequences 1997(SEQUENCES 1997), pp.21–, 1997.
[10] 神谷年洋, “リバースエンジニアリングツールキット remics の試作,” 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, vol.109, no.170, pp.7–11, 2009.
[11] B. Biegel, Q.D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, “Comparison of similarity metrics for refactoring detection,” In Proc. the 8th Working Conference on Mining Software Repositories(MSR 2011), pp.53–62, 2011.
[12] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, “Template-based Reconstruction of Complex Refactorings,” In Proc. the 2010 IEEE International Conference on Software Maintenance(ICSM 2010), pp.1–10, 2010.