情報処理学会論文誌 Vol.59 No.4 1192–1202 (Apr. 2018)

多粒度コードクローンの検出と評価

幸 佑亮[1] 肥後 芳樹[1,a)] 楠本 真二[1]

受付日 2017年8月2日, 採録日 2018年1月15日

概要：コードクローンはソフトウェアの保守性を低下させる原因の 1 つとされており，コードクローンがソフトウェア中にどの程度存在しているか，およびどこに存在しているかを理解することはソフトウェア保守の観点から重要である．そのため，これまでに多くのコードクローン検出手法が提案され，自動的にコードクローンを検出するツールが開発されている．既存のコードクローン検出手法は，ファイル単位やコード片単位など単一の粒度でコードクローンを検出する．一般的に，検出の粒度が粗いほど検出時間は短くなるが，検出可能なコードクローンは少なくなる．一方，検出対象の粒度が細かいほど検出可能なコードクローンは多くなるが，検出時間は長くなる．そこで本論文では，より多くのコードクローンをより短時間で検出することを目的として，粗粒度から細粒度へ段階的にコードクローンを検出する手法を提案する．段階的にコードクローンを検出する過程において，ある粒度でコードクローンとして検出されたコードをそれよりも細粒度なコードクローンの検出対象から除外することで，細粒度な検出手法と比較してより高速に検出できる．また，粗粒度な検出手法と比較してより多くのコードクローンを検出できる．提案手法をコードクローン検出ツール **Decrescendo** として実装し，複数のオープンソースソフトウェアに適用した．そして，提案手法を粗粒度な検出手法および細粒度な検出手法と比較して評価を行った．実験の結果より，細粒度な（コード片単位の）検出手法と比較して，多粒度な検出手法が約 10∼20 倍高速にコードクローンを検出できることを示した．また，粗粒度な（メソッド単位の）検出手法と比較して，多粒度な検出手法が約 10∼30 倍のコードクローンを検出した．この検出数は細粒度な（コード片単位の）検出手法とほぼ同数であった．

キーワード：コードクローン，ソースコード解析，大規模実験

Detection and Evaluation of Multi-grained Code Clones

Yusuke Yuki[1] Yoshiki Higo[1,a)] Shinji Kusumoto[1]

Received: August 2, 2017, Accepted: January 15, 2018

Abstract: The presence of code clones is one of the factors that makes software maintenance more difficult. Understanding the number of code clones and their distribution is helpful for developers. Before now, various kinds of code clone detection techniques and tools have been proposed and developed. Existing techniques detect code clones with a single granularity such as file, method, or code fragment. Coarser granularity detection requires shorter time, but it finds the smaller number of code clones. In this paper, we propose a multi-grained clone detection technique, which detects a larger number of code clones with short time. In the proposed technique, firstly coarse grained detection is performed. Detected coarse-grained code clones are eliminated from the target of finer-grained clone detections. Such a multi-grained detection is effective to shorten detection time and detects a large number of code clones as well as fine-grained detection techniques. The proposed technique has been implemented by a tool, **Decrescendo**. Decrescendo has been applied to a large set of open source software to compare it with existing detection techniques. As a result, the proposed technique finished detection 10∼ times shorter than fine-grained techniques. The proposed technique detected code clones 10∼30 times more than coarse-grained techniques. The number of code clones detected by the proposed technique is approximately the same as fine-grained techniques.

Keywords: code clone, source code analysis, large-scale experiment

1. はじめに

コードクローン（以下，クローン）とは，ソースコード中に存在する互いに同一，あるいは類似したコード片であ

[1] 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
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] higo@ist.osaka-u.ac.jp

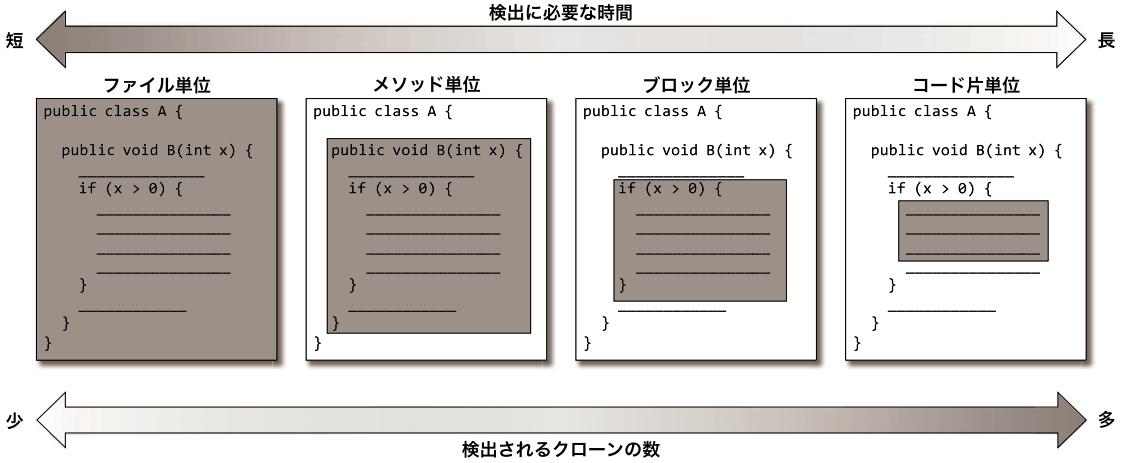


図 1 各検出手法の特徴
Fig. 1 Characteristics of each detection technique.

る．クローンの主な発生要因はコピーアンドペーストである [10], [11], [16], [19]．一般的に，クローンはソフトウェアの保守性を低下させる原因になるとされている．たとえば，あるコード片にバグが存在した場合，そのクローンに対しても同様のバグが存在する可能性があり，同様の変更を検討する必要がある．そのため，クローンがソフトウェア中にどの程度存在しているか，およびどこに存在しているかを理解することはソフトウェア保守の観点から重要であり，これまでに多くのクローン検出ツールが提案されている．

近年，単一のソフトウェアのみでなく，複数のソフトウェアからなる大規模なソースコードの集合に対してクローン検出ツールを適用し，ライブラリの候補や修正漏れを検出する研究が行われている [17]．複数のソフトウェアに存在する同一の処理をライブラリ化することは開発効率の向上の観点から有益であり，先行研究においてそのようなクローンの存在が確認されている．

既存のクローン検出手法は，ファイル単位やコード片単位など単一の粒度でのみクローンを検出している．既存のクローン検出手法は，大きく分けて以下の 2 つの検出手法に分類できる [20]．

粗粒度な検出手法 類似したクラス・メソッド・ブロックをクローンとして検出する手法．

細粒度な検出手法 任意のコード片をクローンとして検出する手法．細粒度な検出手法はクラス・メソッド・ブロックの一部が類似するコード片をクローンとして検出できる．

これらの手法は以下の一長一短な特徴を持つ．ファイル単位，メソッド単位，ブロック単位，およびコード片単位の検出手法の特徴を図 1 にあげる．

検出に要する時間 検出の粒度が粗いほど検出時間が短く，検出の粒度が細かいほど検出時間が長い．ソースコードの行数，ソースコード中の字句数，ソースコードを抽象構文木で表現した場合の頂点数はソースコード中のクラス数，メソッド数，ブロック数と比較して多い．そのため，粗粒度な検出手法と比較して細粒度な検出手法は検出時間が長い．

検出可能なクローンの数 検出の粒度が粗いほど，検出可能なクローンの数が少なく，検出の粒度が細かいほど，検出可能なクローンの数が多い．粗粒度な検出手法はファイル単位，メソッド単位，もしくはブロック単位でクローンを検出するが，それらの内部の一部のみが類似しているコード片をクローンとして検出できない．そのため，細粒度な検出手法と比較して粗粒度な検出手法は検出可能なクローンの数が少ない．一方で細粒度な検出法は，大きな重複コードを小さな単位で検出してしまうという課題がある．たとえば，ファイル全体が重複している場合であっても，メソッド単位のクローン検出法を使った場合は，ファイル全体が 1 つのクローンとしては検出されず，そのファイル内に含まれているメソッドが複数のクローンとして検出されてしまう．

つまり，粗粒度な検出手法は「小さい重複コードを検出できない」というデメリットがあり，細粒度な検出手法は「検出に長い時間を必要とする」および「大きな重複コードを小さい単位のクローンとして検出してしまう」というデメリットがある．本論文では，粗粒度な検出手法と細粒度な検出手法のデメリットを低減させるために，粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する手法を提案する．具体的にはファイル単位・メソッド単位・コード片単位の順にクローンを検出する．段階的にクローンを検出する過程において，ある粒度でクローンとして検出されたコードをそれよりも細粒度なクローンの検出対象から除外していくことで，細粒度な検出手法と比較してより高速に検出できる．

また，粗粒度で検出できるクローンは粗粒度で検出する

ことにより，大きな重複コードが小さい単位のクローンとして検出されることを防げる．たとえば，ファイル全体が重複コードになっている場合には，複数のメソッド単位のクローンとして検出するのではなく，1 つのファイル単位のクローンとして検出する．これによって，検出されるクローンの数が不必要に多くならなくなる．また，提案手法は細粒度でも検出を行うため小さい重複コードも検出可能である．このように，提案手法は粗粒度な検出手法や細粒度な検出手法と比較してより適切な粒度でクローンを検出できる．

提案手法をクローン検出ツール **Decrescendo** として実装し，複数のオープンソースソフトウェアに適用した．そして，提案手法を粗粒度な検出手法および細粒度な検出手法と比較した．

本論文の貢献は以下のとおりである．

- 粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する手法を提案した．
- 細粒度な検出手法と比較して多粒度な検出手法が高速にクローンを検出できること，および粗粒度な検出手法と比較して多粒度な検出手法がクローンの検出数が多いことを示した．
- 細粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できることを示した．

以降，2 章では本研究の位置づけについて述べる．3 章では提案手法について述べる．4 章では実装したツールの詳細について述べる．5 章では評価実験について述べる．6 章では実験の妥当性について述べる．最後に，7 章で本論文をまとめる．

2. 本研究の位置づけ

コード片単位のクローン検出を行う前にファイルのハッシュ値に基づいたクラスタリングを行う，という Choi らの先行研究がある [3]．このクラスタリングはコード片単位のクローン検出時間を短縮することが目的である．ファイルのハッシュ値に基づいたクラスタリングの例を**図 2** に示す．ファイル中の数値はハッシュ値を表しており，ファイル外の枠はハッシュ値が等しいファイルのグループを表す．クラスタリング後，各グループから 1 つのファイルを選択する．そして，選択されたファイルの集合に対してコード片単位のクローン検出を行う．

本論文の提案手法は，コード片単位のクローン検出を行う前にファイル単位およびメソッド単位のクローン検出を行うことにより，先行研究に比べてより短い時間でコード片単位のクローン検出を行うことができる[*1]．さらに，本提案手法は検出した各クローンに粒度の情報を付加する．

[*1] 本研究ではブロック単位のクローン検出は行わない．その理由については 3 章を参照されたい．

図 2 ファイルのハッシュ値に基づいたクラスタリング
Fig. 2 Clustering based on file hash.

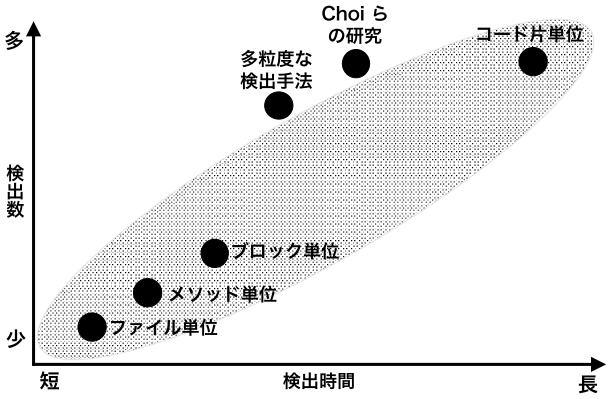


図 3 本研究の位置づけ
Fig. 3 Relatioship between this research and existing ones.

これにより，検出した各クローンの粒度が明らかになり，先行研究よりも分析しやすいクローンの検出結果を生成できる．

既存のクローン検出手法と本論文で提案する多粒度な検出手法の位置づけを**図 3** に示す．粗粒度な検出手法（ファイル単位・メソッド単位・ブロック単位の検出手法）は，検出に要する時間が短く，検出されるクローンの数が少ない．一方，細粒度な検出手法（コード片単位の検出手法）は，検出時間が長く，検出数が多い．そこで，本論文で提案する多粒度な検出手法は，検出時間を短く，かつ検出数を多くすることを目的とする．また，本論文の提案手法は，ファイル単位のクローンに加えて，メソッド単位のクローンも検出する．13,000 個のソフトウェア群に存在するメソッド集合のうち約 49%がメソッドクローンであったという研究報告がある [17]．複数のソフトウェア間にメソッドクローンが多数存在するため，メソッド単位の検出の後にコード片単位の検出を行うことで，コード片単位の検出に要する時間的コストが大幅に改善されることが見込める．

Kodhai らは，メソッドクローンの検出を効率的に行う手法を提案している [8]．彼らの手法では，メソッドクローンの検出を行う前に，各メソッドに対してメトリクス計測

を行い，メトリクス値が同じか類似しているメソッドのみに対してメソッドの各行を比較する処理を行うというものである．この前処理によってクローン検出が短時間で終わることが期待されているが，前処理の有無による検出時間の比較はなされていない．Deepali らはソースコードではなくバイトコードからメトリクスを計測し，十分にメトリクス値が近いクラスについて，そのソースコードを詳細に比較することによりクローンを検出する方法を提案している [4]．こちらについてもメトリクス計測を導入することによる検出速度改善については評価されていない．これらどちらの研究も，計算に時間を要するコード片単位の検出の対象となるソースコードを減らすという点では本研究と類似している．しかし，これらの研究では，重複コードをできるだけ大きな単位として検出するということは達成されておらず，その点で本研究とは異なる．

3. 提案手法

提案手法は対象ソースコードに対して粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する．具体的にはファイル単位，メソッド単位，コード片単位の順にクローンを検出する．段階的に検出する過程において，ある粒度でクローンとして検出されたコードをそれよりも細粒度なクローンの検出対象から除外していく．以降，ファイル単位のクローンをファイルクローン，メソッド単位のクローンをメソッドクローン，コード片単位のクローンをコード片クローンと呼ぶ．

現在のところ，提案手法ではブロック単位でのクローン検出を行わない．その理由は以下の 2 つである．

- ブロックの大きさはメソッドよりも小さく，ブロック単位の検出を導入することによる高速化が限定的であるため．
- コードクローンは 1 つのブロック内で閉じているとは限らないため，検出の処理が煩雑になるため．

3.1 検出手順

図 4 に提案手法の概要を示す．段階的にクローンを検出する過程において，ファイル単位の検出でクローンとして検出されたファイルを，次のメソッド単位の検出の入力から除外する．図 4 では，ファイル単位の検出において，以下の 2 つのファイルクローングループが検出されている．

- A.java，C.java，E.java
- D.java，F.java

次のメソッド単位の検出では各ファイルクローングループから 1 つのファイルを選ぶ（各グループ内の他のファイルはメソッドクローンの検出対象から除外する）．

選択されたファイルとファイルクローンになっていないファイル内に存在しているメソッドに対してメソッドクローンの検出処理が適用される．この例では，メソッドクローン検出処理により以下の 3 つのメソッドがクローンとして検出されている．

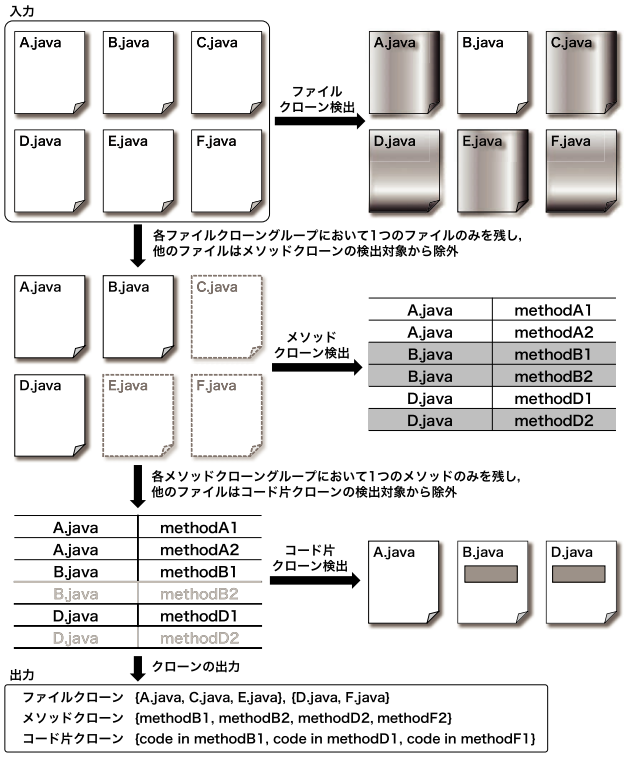


図 4 提案手法の概要

Fig. 4 Overview of the proposed technique.

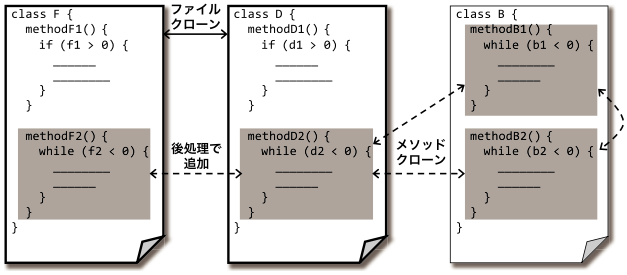


図 5 後処理の例

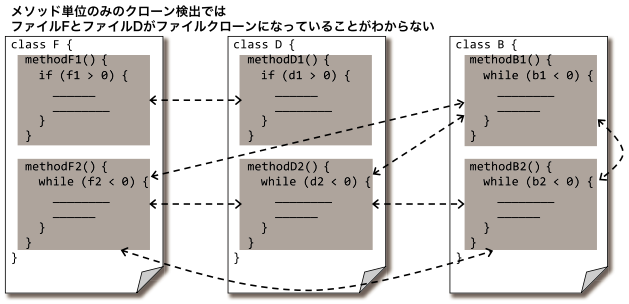
Fig. 5 Example of the post processing.

- methodB1，methodB2，methodD2

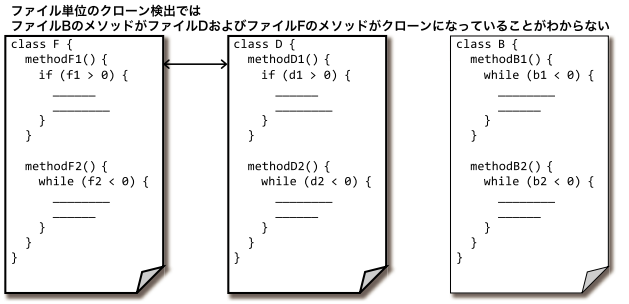
このあと除外したファイルに対するメソッドクローンの検出漏れを防ぐための後処理が行われる．具体的には，メソッドクローン methodD2 が存在している D.java は F.java とファイルクローンであるため，F.java に含まれる methodF2 もこのメソッドクローングループに加える（**図 5**）．結果として，以下の 4 つのメソッドが 1 つのメソッドクローングループとして検出される．

- methodB1，methodB2，methodD2，methodF2

次のコード片単位の検出では，各メソッドクローングループから 1 つのメソッドを選び，メソッドクローンとなっていないメソッドとともにコード片クローン検出処理が適用される．コード片クローン検出後はメソッドクローン検出後の後処理と同様に，除外したメソッド内に含まれ



(a) メソッドクローンのみの検出



(b) ファイルクローンのみの検出

図 6 単粒度におけるクローン検出例
Fig. 6 Example of single granularity detections.

るコード片クローンの検出漏れを防ぐための後処理が行われる．

3.2 多粒度クローンの提示方法

提案手法を利用することにより，重複コードをできるだけ大きい単位のクローンとして検出できる．例を**図 6** にあげる．この例ではメソッド単位のみで検出した場合，3 つのファイルからは以下の 2 つのメソッドクローングループが検出される．

- methodD1，methodF1
- methodB1，methodB2，methodD2，methodF2

この検出結果からは，D.java と F.java がファイルクローンとなっていることが分かりにくい．

ファイル単位のみで検出した場合，以下のファイルクローングループが検出される．

- D.java，F.java

この結果からは，B.java が D.java および F.java とメソッドクローンを共有していることが分からない．

多粒度で検出することで以下のクローングループが検出される．

- D.java，F.java
- methodB1，methodB2，methodD2，methodF2

このように，多粒度で検出を行うことにより，小さいクローンを見逃すことなく，重複コードはできるだけ大きなクローンとして検出できる．

4. 実装

ここでは，著者らが実装した多粒度クローン検出ツール **Decrescendo** について述べる．なお，このツールにおいて，メソッドクローン検出部分は著者らの既存研究 [17] と同一であり，提案手法のファイルクローン検出部分は既存研究 [17] の検出法をファイル単位に変更したものである．コード片クローン検出については Suffix-Tree アルゴリズムおよび Smith-Waterman アルゴリズムを用いて実装した．どちらのアルゴリズムもクローン検出において有用なことが示されている [6], [18]．なお，本研究におけるコード片単位のクローン検出は，各メソッドの内部のコードのみを対象として行っている．つまり，あるファイル内の 2 つの連続したメソッドが他のファイル内の 2 つの連続したファイルと重複コードになっていた場合，それらは 1 つのコード片単位のクローンではなく，各メソッドをコード片とする 2 つのクローンとして検出される．

Decrescendo の入力は以下のとおりである．

- Java ソースファイル群
- 最小クローン長
- 最大ギャップ率

最小クローン長は検出するクローンの最小の大きさを指定するためのものである．最小クローン長に小さな値を指定すればより多くのクローンを検出できるが，検出結果には多くの誤検出が混じる．大きな値を指定すれば誤検出を減らせるが見逃すクローンも増えてしまう．

最大ギャップ率とは，Smith-Waterman アルゴリズムを利用する場合にのみ指定する値であり，クローンとして許容するギャップ（不一致部分）の割合を表す．最大ギャップ率を大きくすればより多くのクローンを検出できるが検出結果には多くの誤検出が含まれてしまう．小さい値を設定すれば誤検出を減らせるが見逃してしまうクローンが増えてしまう．

出力はクローンペアの位置情報，粒度（ファイル・メソッド・コード片），およびタイプ（Type-1・Type-2・Type-3）である．クローンのタイプを**図 7** を用いて説明する．たとえば，コピーアンドペーストしただけのコードであれば，コピー元とコピー先のコード片は完全に同一な Type-1 クローンとなる．変数名を変更した場合には字句単位の差異が生じ，Type-2 クローンとなる．文の追加・削除・変更を行った場合には字句単位よりも大きな差異（ギャップ）が生じ，Type-3 クローンとなる．

検出するクローンのタイプは以下のとおりである．

- ファイル単位およびメソッド単位では，Type-1 および Type-2 のクローンが検出される．
- Suffix-Tree アルゴリズムを用いてコード片クローンの検出を行った場合は，Type-1 および Type-2 のクローンが検出される．

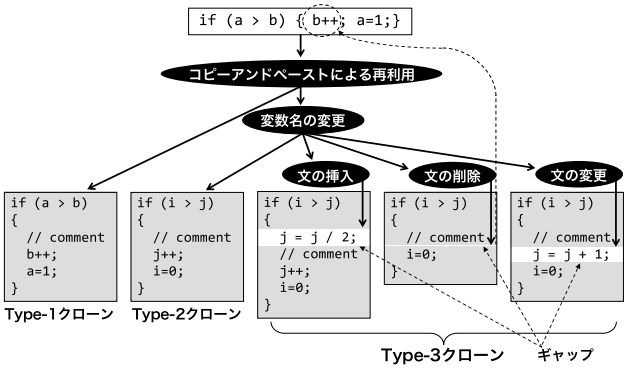


図 7 クローンのタイプ
Fig. 7 Clone types.

- Smith-Waterman アルゴリズムを用いてコード片クローンを検出した場合は，Type-1，Type-2，および Type-3 のクローンが検出される．

コード片単位の検出手法を 2 通りの手法で実装した理由は，細粒度な検出手法のうち，Type-2 までのクローンを検出する手法と Type-3 までのクローンを検出する手法のどちらに対しても，多粒度な検出手法が検出時間と検出数の面で有用かどうかを評価するためである．Type-2 までのクローンを検出する手法の中で Suffix-Tree アルゴリズムを採用した理由は，Suffix-Tree アルゴリズムが用いられているクローン検出ツールが代表的な字句単位の検出ツールであり，様々な企業や研究で採用されているためである [1], [7], [14]．Type-3 までのクローンを検出する手法の中で Smith-Waterman アルゴリズムを採用した理由は，Smith-Waterman アルゴリズムが応用されているクローン検出ツールがその他の Type-3 クローン検出ツールと比較して，高速に検出可能なためである [18]．

また，提案手法はその性質上，Type-3 のファイルクローンやメソッドクローンは検出できない．対象ソースコード中に Type-3 のファイルクローンやメソッドクローンに該当する重複コードが存在した場合には，より細粒度なクローンとして検出される．

以降，本章では Decrescendo の処理の流れを説明する．必要に応じて図 4 を参照されたい．Decrescendo の最初の処理はファイルクローンの検出である．

STEP-1（ファイルの正規化） 入力として与えられた各 Java ファイルに対して以下に示す正規化を行う．

- インデント，改行，コメント，import 文，修飾子を削除
- 識別子名およびリテラルを特殊文字に置換

この処理によって，2 つのファイル間でコーディングスタイルが異なる場合や識別子名またはリテラルが異なる場合でも，その 2 つのファイルをクローンとして検出可能になる．

STEP-2（ファイルフィルタリング） 正規化後のファイルに含まれる字句数が最小クローン長に満たない場合，検出対象から除外する．このフィルタリングを行うことによりクローンが検出されることのないファイルに対するこれ以降の処理を省くことができる．

STEP-3（ファイルハッシュ値の計算） 各ファイルに対してハッシュ値を算出する．現在の実装ではハッシュ値の計算に MD5 を利用している．

STEP-4（ファイルグループの生成と出力） ハッシュ値が同じファイルでグループを作る．2 つ以上のファイルを含むグループがファイルクローンとして検出される．

ファイルクローンの検出の後は，メソッドクローンの検出を行う．

STEP-5（メソッドの切り出し） STEP-4 において生成された各グループからファイルを 1 つ無作為に選択する．ファイルが 1 つだけ含まれるグループも対象である．選択された各ファイルに含まれるメソッドを抽出する．図 4 では，A.java，B.java，および D.java が選択されている．

STEP-6（メソッドフィルタリング） メソッドに含まれる字句数が最小クローン長に満たない場合，検出対象から除外する．このフィルタリングを行うことにより，クローンが検出されることのないメソッドに対するこれ以降の処理を省くことができる．

STEP-7（メソッドハッシュ値の計算） 各メソッドに対してハッシュ値を計算する．ハッシュ値が等しいメソッドがメソッドクローンとなる．ファイルハッシュと同じく MD5 を用いて計算する．

STEP-8（メソッドグループの生成） ハッシュ値が同じメソッドでグループを作る．2 つ以上のメソッドを含むグループがメソッドクローンとして検出される．図 4 では，methodB1，methodB2，および methodD2 がメソッドクローンとして検出されている．

STEP-9（メソッドクローン検出の後処理と出力） STEP-4 で選択されなかったファイル（図 4 における C.java，E.java，および F.java）に含まれるメソッドクローンをとりこぼさないための処理を行う．具体的には，STEP-5 で選択されたファイルから STEP-8 においてメソッドクローンが検出された場合には，STEP-5 で選択されなかったファイルにもそのメソッドクローンが存在しているとする．図 4 では，この処理により，STEP-8 で検出されたメソッドクローンに，methodF2 が加えられている．

メソッドクローンを検出した後は，コード片クローンの検出を行う．

STEP-10（コード片クローンの検出） STEP-8 において生成された各グループからメソッドを 1 つ無作為に選択する．メソッドが 1 つだけ含まれるグループも対象

である．選択された各メソッドに対して，Suffix-Tree もしくは Smith-Waterman アルゴリズムを用いることにより，それらに含まれるコード片クローンを検出する[*2]．

STEP-11（コード片クローン検出の後処理と出力）
STEP-5 で選択されなかったファイルや STEP-10 で選択されなかったメソッドに含まれるコード片クローンをとりこぼさないための処理を行う．処理内容は STEP-9 と同様である．

なお，Decrescendo は，設定によって各粒度における検出のオン・オフを切り替えることができる．

5. 実験

5.1 準備

Decrescendo を複数のオープンソースソフトウェアに対して適用し，多粒度な検出手法と粗粒度な検出手法および細粒度な検出手法と比較した．今回の実験では，最小クローン長を 50 字句，最大ギャップ率を 0.3 としている．これらの値は先行研究 [2], [13] を参考にしている．なお，文献 [15] には，Decrescendo を他のソフトウェア群に対して適用した結果（Smith-Waterman アルゴリズムを用いた場合のみ）が記載されている．

5.2 調査項目

本実験の調査項目を以下に示す．

項目 1 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法は検出時間が短いか．

項目 2 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法は検出されるクローンの数が多いか．

項目 3 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法は分析しやすいクローンの検出結果を生成できるか．

これらの項目を調査するために，各粒度における検出のオン・オフを切り替えたすべての場合において，Decrescendo を実行した．コード片単位の検出を行う場合，Suffix-Tree アルゴリズムと Smith-Waterman アルゴリズムを用いた 2 通りにおいて，Decrescendo を実行した．

5.3 実験環境

本実験で用いた計算機の CPU は 2.9 GHz Intel Xeon CPU（16 プロセッサ）であり，メモリサイズは 352 GB である．また，実験対象のソフトウェアや検出結果を出力するためのデータベースはすべて SSD 上に配置した．

[*2] 本論文では Suffix-Tree アルゴリズムを用いたクローン検出および Smith-Waterman アルゴリズムを用いたクローン検出の手順は記述しない．それらは文献 [6], [18] のクローン検出の手順と同様である．

表 1 対象ソフトウェア
Table 1 Target software.

ソフトウェア数	84
ファイル数	66,724
メソッド数	628,219
LOC	11,545,556

5.4 実験対象

表 1 に対象ソフトウェアの概要を示す．対象ソフトウェアは Apache Software Foundation のリポジトリ[*3]から取得した 2013/10/31 時点の 84 個のソフトウェアである．バージョンが異なる同一ソフトウェア間からのクローン検出を避けるために trunk 以下のファイルのみを検出対象とする．これらのソフトウェアを対象とした理由は，著者らの先行研究 [5] において利用されたデータセットであり，テストコードと自動生成コードが取り除かれているためである．テストコードと自動生成コードは流用の特定や複数のソフトウェアに存在する共通処理のライブラリ化に対しては有用でなく，クローンとして検出する必要性がない．

5.5 項目 1 の調査

表 2 にクローンの検出結果を示す．各粒度の ○ はその粒度の検出が行われたことを表す．検出 ID は，本節の以降でどの検出について言及しているのかを簡潔に表すための記号である．ST および SW はそれぞれ Suffix-Tree アルゴリズム，Smith-Waterman アルゴリズムを表す．なお，Suffix-Tree アルゴリズムを用いた場合と Smith-Waterman を用いた場合で同様の傾向が得られたので，ここでは Suffix-Tree アルゴリズムを用いた場合のみを述べる．

ファイル単位，メソッド単位で検出した場合（検出 A，検出 B）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出時間が長かった（7.3 秒，32.8 秒 →3,094.6 秒）．また，コード片単位で検出した場合（検出 C）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出速度が大幅に向上した（62,569.0 秒 →3,094.6 秒）．**表 3** に各処理に要した時間を示す．全粒度で検出した場合は，STEP-5 から STEP-7（メソッドクローンの検出）に要する時間が若干短くなった（31.7 秒 →24.0 秒）．これはファイルクローンとして検出されたファイルが除外され，メソッド単位の検出時の入力ファイル数が減少したためである．

最も注目すべきは，STEP-10（コード片クローンの検出）に要した時間である．この処理が最も実行時間が減少している（62,508.9 秒 →2,980.0 秒）．全粒度で検出した場合に実行時間が約 1/20 倍になっている．どちらの場合も検出時間の大半を STEP-10 に要しているため，重複ファイルと重複メソッドを除外することが Suffix-Tree アルゴリズ

[*3] http://svn.apache.org/viewvc/

表 2 クローンの検出結果
Table 2 Clone detection results.

検出 ID	粒度				検出されたクローンペア数				検出時間 [s]
	ファイル	メソッド	コード片 ST	コード片 SW	ファイル	メソッド	コード片	合計	
A	○	-	-	-	11,029	-	-	11,029	7.3
B	-	○	-	-	-	49,446	-	49,446	32.8
C	-	-	○	-	-	-	1,637,597	1,637,597	62,569.0
D	-	-	-	○	-	-	446,442	446,442	130,367.1
E	○	○	-	-	11,029	44,168	-	55,197	32.3
F	○	-	○	-	11,029	-	1,632,319	1,644,348	46,183.2
G	-	○	○	-	-	49,446	1,588,151	1,637,597	1,885.7
H	○	○	○	-	11,029	44,168	1,588,151	1,643,348	3,094.6
I	○	-	-	○	11,029	-	441,164	452,193	109,717.1
J	-	○	-	○	-	49,446	396,996	446,442	13,013.6
K	○	○	-	○	11,029	44,168	396,996	452,193	13,036.7

表 3 各処理に要した時間（Suffix-Tree アルゴリズム）
Table 3 Detection time (Suffix-Tree algorithm).

処理	全粒度 [s]	コード片 [s]
STEP-1 から STEP-3	6.7	6.4
STEP-4	4.0	-
STEP-5 から STEP-7	24.0	31.7
STEP-8 および STEP-9	1.2	-
STEP-10	2,980.0	62,508.9
STEP-11	68.5	16.6

ムを用いたマッチング処理に要する時間を短縮し，結果として検出速度の向上に有効であることが分かる．また，STEP-11 に要した時間を時間を比較すると，全粒度で検出した場合は後処理を行うため実行時間が長くなっている．しかし，その差は全体の検出時間と比較するとわずかな時間である．

さらに，ファイルおよびコード片単位で検出した場合（検出 F）と全粒度で検出した場合（検出 H）を比較すると，メソッドクローンを除外してコード片クローンの検出を行うことが有効であることが分かる（46,183.2 秒 →3,094.6 秒）．

なお，表 3 のコード片の列には，STEP-1 から STEP-3 および STEP-5 から STEP-7 についての実行時間も含まれている．これらの処理には，コード片単位のクローン検出のみを行う場合には必要のないものも含まれている．ツールの実装上の都合により，コード片単位の検出の場合もそのような処理を行っているが，検出時間全体に対してこれらの処理に必要な時間は非常に短いため，問題はないと著者らは考えている．

項目 1 の結論

これらの結果から多粒度な検出手法は粗粒度な検出手法よりも検出時間が長いという結論を得た．しかし，細粒度な検出手法と比較して，多粒度な検出手法が高速にクローンを検出できることを示した．大規模なソースコードの集合に対してクローン検出を行う場合，多量のファイル・メソッドクローンが検出される [9], [12], [17]．そのような場合において，多粒度な検出手法はコード片クローンを検出するための時間を短縮することに有効であるといえる．

5.6 項目 2 の調査

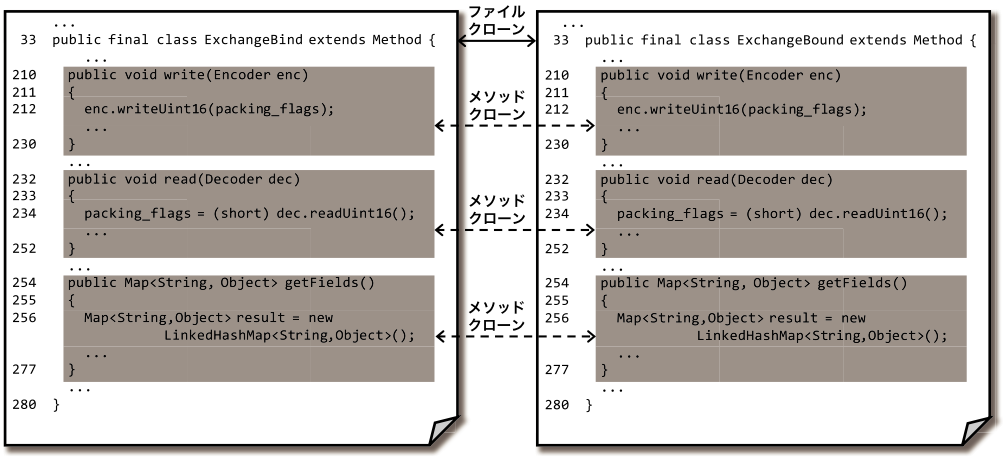
項目 2 についても，Suffix-Tree アルゴリズムを用いた場合と Smith-Waterman アルゴリズムを用いた場合で同様の傾向であったため，Suffix-Tree アルゴリズムを用いた場合のみについて述べる．

ファイル単位およびメソッド単位で検出した場合（検出 A，検出 B）と全粒度で検出した場合（検出 H）を比較すると，全粒度で検出した場合は検出数が多かった（11,029 個，49,446 個 →1,643,348 個）．また，コード片単位で検出した場合（検出 C）と全粒度で検出した場合（検出 H）を比較すると，検出数にあまり変化はなかった（1,637,597 個 →1,643,348 個）．

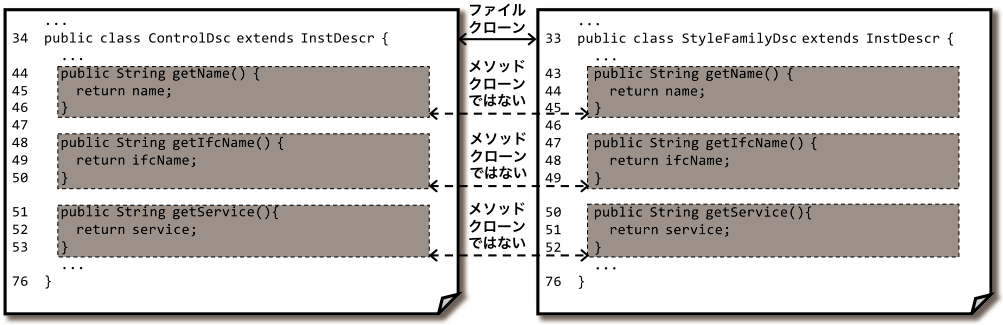
表 2 には，直感とは反した検出結果になっている数値がある．たとえば，検出 C において検出されたクローンペア数は検出 H において検出されたクローンペア数よりも少ない．この理由は，4 章において記述しているコード片単位のクローン検出の実装方針によるものである．コード片単位のクローンは各メソッド内でのみ検出処理が行われる．そのため，たとえ重複しているコードでもそれを含むメソッドが最小クローン長よりも小さい場合には，それらは検出されない．一方で多粒度での検出の場合は，そのようなメソッドであっても，それを含むファイル全体が重複していればクローンとして検出される．この理由により，多粒度による検出数が細粒度による検出数よりも多くなる場合が存在する．

項目 2 の結論

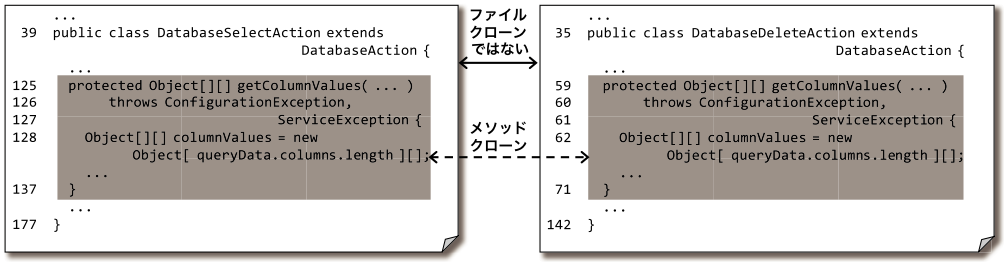
粗粒度な検出手法と比較して多粒度な検出手法は検出数が多いということを示した．また，多粒度な検出手法は細粒度な検出手法と同等数のクローンを検出することを示した．



(a) 複数のメソッドクローンを 1 つのファイルクローンとして検出した例



(b) メソッドクローンが検出されないファイルをファイルクローンとして検出した例



(c) ファイル内の一部のメソッドのみをメソッドクローンとして検出した例

図 8 分析しやすい検出結果の例
Fig. 8 Example of preferable clones.

5.7 項目 3 の調査

粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成した例を以下に 3 つ示す．

1 つ目の例を**図 8** (a) に示す．メソッド単位のみで検出した場合，write，read，getFields メソッドがそれぞれ異なるクローンペアとして検出されていた．メソッドクローンペアを 1 つ 1 つ確認することで，これらのクラス内のすべてのメソッドがクローンになっており，2 つのクラスを 1 つのクラスに集約するリファクタリングが可能であると判断できる．しかし，複数のメソッドクローンペアとしてこれら 2 つのクラスの重複が検出されるため，リファクタリングが可能かどうかの判断に時間を要する．多粒度な検出手法では，メソッド単位の検出前にファイル単位の検出を行うため，これら 2 つのクラスはファイルクローンであることが明らかになり，より容易にリファクタリングが可能かどうかを判断できる．多粒度な検出手法によって，5,278 個のメソッドクローンペアが 2,345 個のファイルクローンペアとして検出されたことを確認した．

2 つ目の例を図 8 (b) に示す．メソッド単位のみやコード片単位のみで検出した場合，getName，getIfcName，getService メソッドはクローンとして検出されなかった．この理由は，2 つのクラス内の全メソッドが最小クローン長未満のメソッドであったためである．しかし，メソッド単位の検出に加えてファイル単位でも検出を行うことで，これら 2 つのクラスがクローンとして検出された．これらのクラスがファイルクローンとして検出されたことで，2 つのクラスを 1 つのクラスに集約するリファクタリングがで

きる可能性がある．そのため，メソッド単位のみのクローン検出では気づくことができないリファクタリングが可能となる．多粒度な検出手法によって，8,684 個のファイルクローンがこのようなケースで検出されたことを確認した．

3 つ目の例を図 8 (c) に示す．ファイル単位のみで検出した場合，2 つのクラスはクローンとして検出されなかった．しかし，メソッド単位のクローン検出を行うことで，getColumnValues メソッドが検出された．DatabaseSelectAction と DatabaseDeleteAction クラスは DatabaseAction クラスを継承しており，getColumnValues メソッドに対してメソッド引き上げリファクタリングができる可能性がある．

以上のことから，多粒度な検出手法は，単一の粒度では気づくことができないリファクタリングが可能となる．ファイル，メソッド，コード片クローンに対して適用できるリファクタリングは異なるため，検出されたクローンの粒度が明確になることで，どのリファクタリングを適用できるかどうかをより容易に判断できる．

多数のクローンが検出されたため，すべてのクローンを確認はできてはいないが，目視で確認した範囲では，多粒度による検出が不利になると思われるクローンはなかった．

項目 3 の結論

粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できることを示した．

6. 実験結果の妥当性について

対象ソフトウェア 本論文では，Java で記述された 84 個のソフトウェアを対象にしている．しかし，他のソフトウェアに対して実装したツールを実行した場合，本論文で得られた結果と異なる可能性がある．

ハッシュ値の衝突 本論文では，ファイル，メソッド，文を構成する文字列からハッシュ値を算出している[*4]．ハッシュ値の衝突が発生した場合，誤ったクローンが検出される可能性がある．しかし，本論文では 128 ビットのハッシュ値を出力する MD5 を用いており，ハッシュ値の衝突の可能性は十分に低い．

正規化の方法 本論文では，3 章で述べた正規化を行っている．正規化により，Type-2 クローンの検出が可能になるが，誤検出も増える．現時点では，正規化によりどの程度の誤検出が増えるのかは確認できていない．

7. おわりに

本論文では，粗粒度から細粒度へ段階的に（多粒度で）クローンを検出する手法を提案した．また，提案手法をクローン検出ツール Decrescendo として実装し，複数のオープンソースソフトウェアに適用した．そして，多粒度な検出手法を粗粒度な検出手法および細粒度な検出手法と比較した．

比較の結果，以下の項目を示した．

- 細粒度な検出手法と比較して，多粒度な検出手法が高速にクローンを検出できる．
- 粗粒度な検出手法と比較して，多粒度な検出手法がクローンの検出数が多い．
- 粗粒度な検出手法および細粒度な検出手法と比較して，多粒度な検出手法が分析しやすいクローンの検出結果を生成できる．

今後はまず，被験者実験を行い多粒度な検出手法が生成した検出結果の分析のしやすさをさらに評価する予定である．検出において内部クラスを考慮するなどのさらにより良い検出が行えるよう拡張することも考えられる．また，Java 以外の言語への拡張やその他のクローン検出ツールとの比較を行う予定である．さらには，より大規模なソースコードの集合に対して適用し，ライブラリの候補となるクローンや修正漏れの発見も試みる．

謝辞 本研究は，科学研究費補助金基盤研究 (S)（課題番号：25220003）および基盤研究 (B)（課題番号：17H01725）の助成を得て行われた．

[*4] 文を構成する文字列から算出したハッシュ値は，Smith-Waterman アルゴリズムで利用している．

参考文献

[1] Barbour, L., Khomh, F. and Zou, Y.: An empirical study of faults in late propagation clone genealogies, *Journal of Software: Evolution and Process*, Vol.25, No.11, pp.1139–1165 (2013).

[2] 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).

[3] Choi, E., Yoshida, N., Higo, Y. and Inoue, K.: Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files, *IEICE Trans. Information and Systems*, Vol.98, No.2, pp.325–333 (2015).

[4] Deepali, Gupta, A. and Batra, C.: Hybrid approach for Detecting Code Clone by Metric and Token based comparison, *International Journal of Advanced Research in Computer Science*, Vol.7, No.6, pp.297–302 (2016).

[5] Higo, Y. and Kusumoto, S.: How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods, *Proc. 22nd International Symposium on the Foundations of Software Engineering*, pp.294–305 (2014).

[6] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).

[7] Kim, M., Sazawal, V., Notkin, D. and Murphy, G.: An empirical study of code clone genealogies, *Proc. ACM SIGSOFT Software Engineering Notes*, Vol.30, No.5, pp.187–196 (2005).

[8] Kodhai, E., Kanmani, S., Kamatchi, A., Radhika, R. and Saranya, B.V.: Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics, *Proc. 2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, pp.241–243

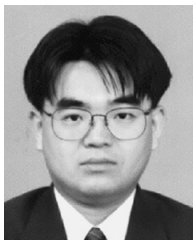
(2010).
[9] Ossher, J., Sajnani, H. and Lopes, C.: File cloning in open source java projects: The good, the bad, and the ugly, *Proc. 27th International Conference on Software Maintenance*, pp.283–292 (2011).
[10] Rattan, D., Bhatia, R. and Singh, M.: Software clone detection: A systematic review, *Information and Software Technology*, Vol.55, No.7, pp.1165–1199 (2013).
[11] Roy, C.K., Cordy, J.R. and Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming*, Vol.74, No.7, pp.470–495 (2009).
[12] Sasaki, Y., Yamamoto, T., Hayase, Y. and Inoue, K.: Finding file clones in FreeBSD ports collection, *Proc. 7th Working Conference on Mining Software Repositories*, pp.102–105 (2010).
[13] Svajlenko, J. and Roy, C.K.: Evaluating clone detection tools with BigCloneBench, *Proc. 31st International Conference on Software Maintenance and Evolution*, pp.131–140 (2015).
[14] Weissgerber, P. and Diehl, S.: Identifying refactorings from source-code changes, *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pp.231–240 (2006).
[15] 幸　佑亮，肥後芳樹，楠本真二：多粒度コードクローン検出手法の提案，電子情報通信学会技術研究報告，Vol.116, No.277, pp.73–78 (2016).
[16] 神谷年洋，肥後芳樹，吉田則裕：コードクローン検出技術の展開，コンピュータソフトウェア，Vol.28, No.3, pp.29–42 (2011).
[17] 石原知也，堀田圭佑，肥後芳樹，井垣　宏，楠本真二：大規模なソフトウェア群を対象とするメソッド単位でのコードクローン検出，情報処理学会論文誌，Vol.54, No.2, pp.835–844 (2013).
[18] 村上寛明, 堀田圭佑, 肥後芳樹, 井垣　宏, 楠本真二：Smith-waterman アルゴリズムを利用したギャップを含むコードクローン検出，情報処理学会論文誌，Vol.55, No.2, pp.981–993 (2014).
[19] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌 D，Vol.91, No.6, pp.1465–1481 (2008).
[20] 堀田圭佑，楊　嘉晨，肥後芳樹，楠本真二：粗粒度なコードクローン検出手法の精度に関する調査，情報処理学会論文誌，Vol.56, No.2, pp.580–592 (2015).

幸 佑亮

2015 年大阪大学基礎工学部情報科学科中退．2017 年同大学大学院博士前期課程修了．在学時，コードクローン分析やリポジトリマイニングに関する研究に従事．

肥後 芳樹　（正会員）

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学院情報科学研究科コンピュータサイエンス専攻助教．2015 年同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援およびソフトウェアリポジトリマイニングに関する研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．



楠本 真二　（正会員）

1988 年大阪大学基礎工学部卒業．1991 年同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 年同講師．1999 年同助教授．2002 年同大学大学院情報科学研究科助教授．2005 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．