情報処理学会論文誌 Vol.62 No.4 1065–1075 (Apr. 2021)

コードクローンの自動集約に基づく削減可能なソースコード行数の測定

中川 将[1,a)] 肥後 芳樹[1,b)] 松本 淳之介[1,c)] 楠本 真二[1,d)]

受付日 2020年2月3日, 採録日 2021年1月12日

概要：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一致または類似しているコード片を指す．クローンはソフトウェア保守を困難にする要因の１つとして知られている．リファクタリングを行い，クローンを１つのメソッドやクラスなどのモジュールに集約することにより保守が容易になる．しかし，リファクタリングは正常に稼働しているソースコードに手を加える作業であり，またリファクタリングによりかえってバグが混入する可能性がある．そのため，リファクタリングを行う動機となる指標が必要となる．クローンのリファクタリングによる削減可能な行数がその指標として用いられる．既存研究では，クローンの情報や行数の解析に基づいて削減可能なソースコード行数を推定する手法を提案している．しかし，削減行数の推定にとどまっているため，リファクタリングにより実際に削減される行数とは乖離が存在する場合があるのではないかと著者らは考えた．そこで本研究では，削減可能なソースコード行数をより正確に推定する手法を提案する．提案手法では，クローンの検出，集約，ソースコードのコンパイル，テストを繰り返し自動で行い，削減行数を測定する．また，提案手法を Java プロジェクトに対して適用し，既存手法との比較を行った．その結果，既存手法と比較してより正確にソースコードの削減行数を測定できた．

キーワード：ソフトウェア保守，コードクローン，リファクタリング

Measurement Reducible Lines of Code Based on Automated Merging Code Clones

Tasuku Nakagawa[1,a)] Yoshiki Higo[1,b)] Junnosuke Matsumoto[1,c)] Shinji Kusumoto[1,d)]

Received: February 3, 2020, Accepted: January 12, 2021

Abstract: A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. The presence of clones is known as one of the factors that make the software maintenance difficult. Refactoring clones, merging clones as a module such as a method or class, makes the software maintenance more manageable. However, refactoring is modifying source code of which software working regularly and, in some cases, introduces a new bug. Thus, developers need criteria for refactoring. Lines of code (in short, LoC) that can be reduced by refactoring clones is used as the criterion. The existing study proposed a technique to estimate reducible LoC based on analyzing the information and the amount of source code of clones. However, we think that reducible LoC that the existing technique estimates is different from actual value because the existing technique only estimates. Consequently, in this research, we propose a new technique to calculate reducible LoC more accurately. The proposed technique performs a loop processing of detecting clones, merging them, compiling the edited source files, and testing them. After finishing the loop, reducible LoC is calculated from the edited source files. This paper also includes comparison results of the proposed technique and the existing one. As a result, we confirmed that the proposed technique was able to calculate more accurate reducible LoC than the existing technique.

Keywords: code clone, refactoring, software maintenance

1. はじめに

一度完成したソフトウェアであっても，機能追加やバグ修正といった理由によりソースコードには変更が加え続けられる．ソースコードの変更の結果，設計の崩壊や可読性の悪化により，保守コストが肥大化する可能性がある [4], [13], [15]．ソフトウェアの保守に必要なコストはそのソフトウェアの規模や複雑さから算出されることが多い [15]．巨大で複雑なソフトウェアを保守し続けるためには莫大な金額が必要となる．そこで，利用者は現在使用しているソフトウェアの価値を見積もることで，保守コストを支払って使い続けるか，そのソフトウェアを放棄し新しいソフトウェアを構築するかを判断することがある [16]．

ソフトウェアの保守性を悪化させる原因の 1 つとして，コードクローン（以下，クローン）があげられる．クローンとは，ソースコード中に存在する互いに一致または類似しているコード片である．クローンは，ソフトウェアの開発や保守の過程で作りこまれる [6], [12]．クローンが多く存在することで，ソースコードが冗長となり，また潜在的なバグの発生箇所が多く存在することになる [10]．したがって，ソフトウェアの保守性を向上させるという観点において，クローンの集約は有効な手段の 1 つである．

リファクタリングによりクローンを集約できる．リファクタリングとはソフトウェアの外部的振舞いを保ちつつ内部の構造を改善する作業を指す [5]．リファクタリングの 1 つにメソッド抽出という手法がある．メソッド抽出とは，既存のソースコードから一部のコード片を切り出し新たなメソッドを生成し，元のコード片をそのメソッド呼び出しに置き換える作業である．メソッド抽出により，クローンの集合（以下，クローンセット）を 1 つのメソッドに集約できる．その結果，ソースコードの一貫した修正が容易になるので，将来の保守コストの削減が期待できる．しかし，リファクタリングは問題なく稼働しているソフトウェアのソースコードに手を加える作業であり，またリファクタリングによりかえってバグが混入する可能性もある．そのため，開発者にはリファクタリングを行う動機となる指標が必要となる．クローンのリファクタリングによる削減可能な行数がその指標として用いられる．

既存研究では，クローンのリファクタリングによる削減可能なソースコード行数を算出する手法を提案している [20]．既存手法では，Tsantalis らが開発したツール JDeodorant [17] を用いてクローンのリファクタリングの可否を判定し，リファクタリングが可能だと判定されたクローンの行数から削減可能行数を計算する．削減可能行数の計算にはヒューリスティックな解法である貪欲法を用いて，削減できる行数が最大となるようにリファクタリング対象のクローンセットを選択する．

しかし，この手法には 2 つの課題があるため，削減可能行数を正しく推定できない場合があると著者らは考えた．

- 1 つ目の課題は，クローンのリファクタリングの可否を正しく判定できないということである．JDeodorant では，いくつかの条件を設け，そのすべての条件に一致するクローンだけをリファクタリング可能であると判定している．しかし，JDeodorant が設けたすべての条件に一致しているクローンであっても，実際にはリファクタリングできない可能性がある．あるいは，条件に当てはまらないクローンであっても，リファクタリングが可能な場合も考えられる [20]．一方，検出された大量のクローンすべてに対する，手作業によるリファクタリングの可否の確認には多大な労力が必要となり現実的ではない．
- 2 つ目の課題は，オーバラップしているクローンに対して，削減可能行数を適切に計算できないということである．既存手法が採用している貪欲法では，オーバラップしているそれぞれのクローンの削減可能行数を別々に計算したうえで，最も削減可能行数が大きいクローンのみをリファクタリングの対象にし，それ以外は対象にしない．しかし，実際には，オーバラップしているクローンの 1 つをリファクタリングしたうえで，それ以外のクローンもさらにリファクタリングできる場合がある．そのため，実際の削減行数とは異なる行数が推定されてしまう．

そこで本研究では，クローンの検出，ソースコードの変更，コンパイルおよびテストをすべて繰り返し自動で行い，実際に削減行数を測定することで，より正確なソースコードの削減可能行数を算出する手法を提案する．提案手法を用いて Java 言語で記述されたプロジェクトに対して実験を行い，削減可能行数を算出した．また，算出した値に対して，既存手法で算出した値との比較を行った．その結果，提案手法は既存手法に比べてより正確に削減可能行数を算出できることを示した．

2. 準備

2.1 クローン検出

クローンとは，ソースコード中に存在する互いに一致または類似しているコード片を指す [9]．互いがクローンとなっているコード片の組はクローンペア，含まれるすべての組がクローンとなっているコード片の集合はクローンセットと呼ばれる．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 [19]．コピーアンドペーストの利用によ

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0971, Japan
a) t-nakagw@ist.osaka-u.ac.jp
b) higo@ist.osaka-u.ac.jp
c) j-matumt@ist.osaka-u.ac.jp
d) kusumoto@ist.osaka-u.ac.jp

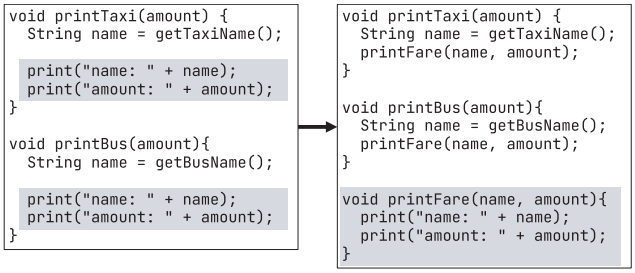


図 1 メソッド抽出の例
Fig. 1 An example of extract method.

りソフトウェアの実装速度が速くなる一方で，発生するクローンはソフトウェアの保守性が低下する原因の 1 つとなる [7]．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 [10]．したがって，クローンがソースコード中のどこに存在しているか，どの程度存在しているかの把握はソフトウェアの保守において重要である．

そのため，自動でクローンを検出する研究がさかんに行われており，多くのクローン検出ツールが開発されている [1], [8], [9]．コード片が一致または類似しているかを判断する基準はそれぞれの検出手法や検出ツールによって異なる．

2.2 クローンのリファクタリング

リファクタリングとは，外部的振舞いを保ちつつ内部の構造を改善する作業である [5]．リファクタリングを提唱した Fowler は，リファクタリングを行うべきコードとして重複したコード，すなわちクローンをあげている．これまでに様々なリファクタリング手法が考案されており，クローンの集約に有効な手法も複数存在する [21]．クローンの集約に有効なリファクタリング手法の 1 つがメソッド抽出である．メソッド抽出とは，既存のソースコードから一部のコード片を切り出し新たなメソッドにする作業を指す．メソッド抽出の例を**図 1** に示す．クローンセットに対するメソッド抽出によりクローンを集約できる．その結果，潜在的なバグの修正箇所とソースコード行数の削減が可能である．

2.3 クローンのオーバラップ

異なるクローンの一部が重なっている，あるいは片方がもう片方を内包しているとき，それらのクローンはオーバラップしているという．オーバラップしているクローンの例を**図 2** (a) に示す．図では，クローン A がクローン B を内包している．クローンのリファクタリングでは，オーバラップしているクローンセットを同時にリファクタリングできない場合がある．そのため，クローンのリファクタリングによる削減可能なソースコード行数の見積もりには，

図 2 クローンのオーバラップの例
Fig. 2 An example of clone overlapping.

オーバラップしているクローンセットに対して何らかの工夫を行う必要がある．たとえば，既存研究 [18] では図 2 (b) のように，クローン A を集約するよりもクローン B を集約した方がより多い削減可能行数を見込める場合，クローン B のみを集約対象とし削減可能行数を見積もる手法を提案している．既存手法 [20] ではこの手法を採用している．あるいは，図 2 (c) のように，オーバラップしている部分でコード片を分割し，新たにクローン A’，クローン B，クローン C とし，クローン A とクローン B の両方を集約できるようにしたうえで削減可能行数を見積もる手法も提案している．

2.4 削減可能行数の計算

クローンになっているコード片に対するメソッド抽出により，ソースコードの行数の削減が可能である．あるクローンセットが n 個のクローンから成り立っており，1 つのクローンの行数が C_{size} であるとする．メソッド抽出により，元のクローンだったコード片を 1 行のメソッド呼び出し文に置換できる．すなわち，クローンセット内のクローンすべてを集約した場合のソースコード行数の削減量 C_{all} は

$$C_{all} = n * C_{size} - n \tag{1}$$

と表される．また，C 言語での関数宣言文や Java におけるメソッド宣言文は一般的に “シグニチャ + {”，“本体”，及び “}” で構成される．すなわち，関数やメソッドの行数は（本体 +2）行となる．抽出されたメソッドの本体はクローンとなっているコード片となるので，メソッドの行数 M は

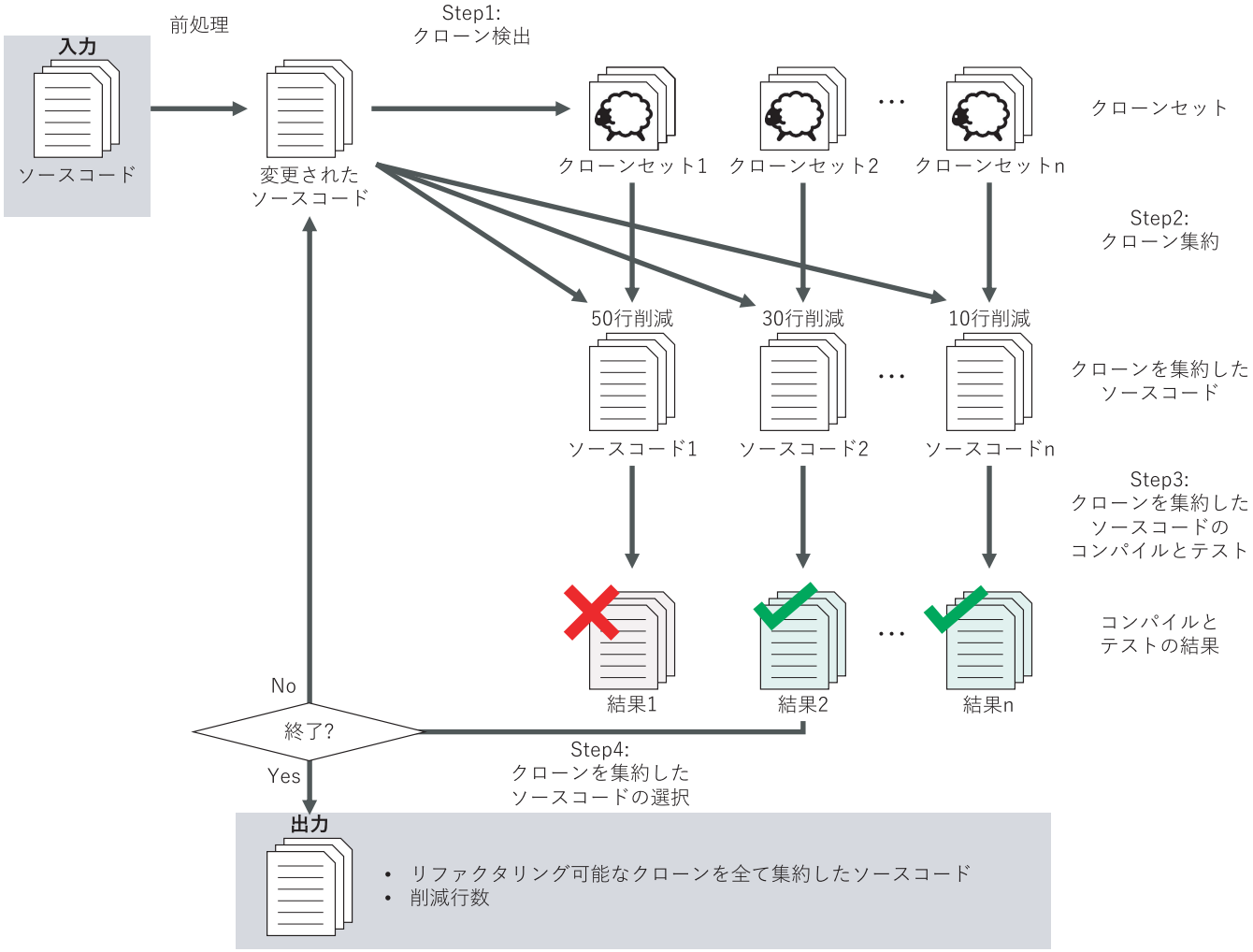


図 3 提案手法の概要
Fig. 3 An overview of the proposed technique.

$$M = C_{size} + 2 \tag{2}$$

で表される．式 (1)，式 (2) から，クローンのリファクタリング前後でのソースコード全体の削減行数 S は

$$S = C_{all} - M = (n-1) * C_{size} - n - 2 \tag{3}$$

と表される．

既存研究 [20] では，1 つのクローンセットをリファクタリングしたときの削減可能行数の見積もりに上記の式 (3) を利用している．

3. 提案手法

本研究では，クローンの検出，ソースコードの変更，コンパイル，テストを繰り返し自動で行いソースコードの削減可能行数を算出する手法を提案する．提案手法を用いることで，実際に集約するとコンパイルとテストを通過するクローンセットに対して，オーバラップしているクローンセットも考慮して集約した際の削減可能行数を算出できる．提案手法がオーバラップしているクローンセットを考慮できる理由は，提案手法はクローンの検出とソースコードの変更を繰り返し行うからである．例として，図 2 (a) のように，クローン A がクローン B を内包している場合を考える．この場合，提案手法では，クローン A の集約を行った後に再度クローンの検出を行うため，クローン A が内包していたクローン B を改めて集約対象として扱える．提案手法の概要を**図 3** に示す．本研究における提案手法の入力はソースコードである．出力はリファクタリング可能なクローンをすべて集約したソースコードと削減されたソースコード行数である．

本研究の提案手法は以下の 2 つの工程で構成される．

- 前処理
- 繰り返し処理

以降，本章では各工程について詳細に述べる．

3.1 前処理

前処理として，対象のプロジェクトに新しいクラスを用意する．メソッド抽出で切り出されたメソッドは，このクラス内に static メソッドとして宣言される．それに加え

て，ローカル変数の未初期化によるコンパイルエラー，およびコーディングスタイルによる削減行数の変化を防ぐために，対象のプロジェクトのソースコードに変更を加える．

3.2 繰り返し処理

繰り返し処理は次の 4 ステップから構成される．これらはすべて自動で行われる．

Step 1：クローン検出

Step 2：クローン集約

Step 3：クローンを集約したソースコードのコンパイルとテスト

Step 4：クローンを集約したソースコードの選択

Step 1 では，対象とするソースコード中に存在するクローンセットを検出する．Step 2 では，Step 1 で検出したクローンセットの 1 つを集約する．抽出されたメソッドは，3.1 節の前処理で用意したクラス内に宣言する．Step 3 では，変更されたソースコードに対してコンパイルとテストを行い，外部的振舞いが変化していないかどうかを判定する．以降，Step 2 で変更されたソースコードの中で，行数が削減され，かつコンパイルとテストを通過するようなソースコードを**選択可能なソースコード**と定義する．Step 1 で検出したすべてのクローンセットに対して Step 2 と Step 3 を行い，Step 4 で選択可能なソースコードの中で削減行数が最も多いソースコードを選択する．そして選択したソースコードに対して繰り返し Step 1 以降を実行する．終了条件に到達した時点で繰り返し処理を終了し，削減行数を測定し出力する．クローン集約の終了条件は以下のとおりである．

- クローンが検出されなくなる．
- 選択可能なソースコードが存在しなくなる．

4. 実装

本研究では，提案手法をツールとして実装した．使用した言語は Java であり，Java で書かれたソースコードを対象に実行できる．

4.1 前処理

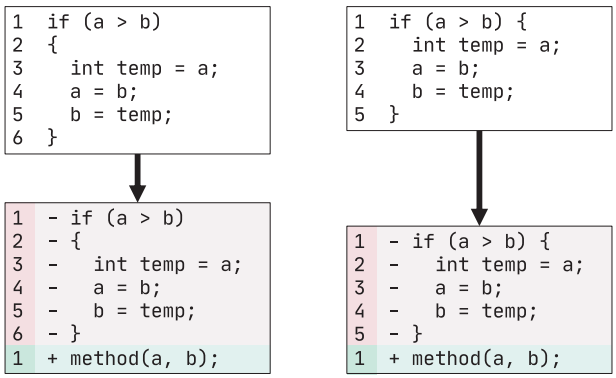
対象とするソースコードに 3.1 節で述べた 3 つの変更を加える．

抽出されたメソッドの宣言を行うクラスの用意

メソッド抽出で切り出されたメソッドを宣言するクラスを新しく用意する．このクラスは新しく用意したパッケージ内で宣言される．このクラス内で宣言されたメソッドが元のコード片があった場所から呼び出される際には完全修飾名で呼び出される．

ローカル変数の初期化

Java の言語規約上，初期化されていないローカル変数の参照は禁止されている．そのため，抽出されたメソッドを呼び出すときに，初期化されていないローカル変数を引数として与えるとコンパイルエラーが発生する．開発者がリファクタリングを行う場合，何らかの値で初期化してから引数として与えることが考えられる．したがって本研究では，`final` 修飾子がついておらず，宣言時に初期化されていないすべてのローカル変数を事前に初期化する．ローカル変数が `boolean` 型以外のプリミティブ型の場合は `0`，`boolean` 型の場合は `false`，参照型の場合は `null` で初期化する．



(a) 5 行削減　　(b) 4 行削減

図 4 コーディングスタイルによって削減行数が異なる例

Fig. 4 An example of different reducible LoC because of different coding styles.

フォーマッタの適用

コーディングスタイルの規約はプロジェクトごとに定められている．たとえば，改行を多く使うような規約もあればその逆も存在する．プロジェクト内でコーディング規約に従わないコードが存在した場合，正確な削減行数を測定できない可能性がある．コードによって削減行数が異なる例を**図 4** に示す．図 4 (a) ではコード片をメソッドに切り出すと 5 行削減されるのに対して，図 4 (b) では 4 行の削減になる．したがって，事前にフォーマッタを利用しコーディングスタイルを統一する必要がある．本来ならプロジェクトごとのコーディング規約を再現したフォーマッタを用意するべきだが，それは困難なため，本研究では適用するフォーマッタは Java の統合開発環境の 1 つである Eclipse の基本設定のフォーマッタで統一した [2]．

4.2 繰り返し処理

Step 1：クローン検出

クローン検出の例を**図 5** に示す．クローンはブロック単位で検出する．本研究では，Eclipse JDT で Statement クラスを継承しているクラスのうち，以下のクラスをブロックとして検出する [3]．

- Block
- DoStatement
- EnhancedForStatement

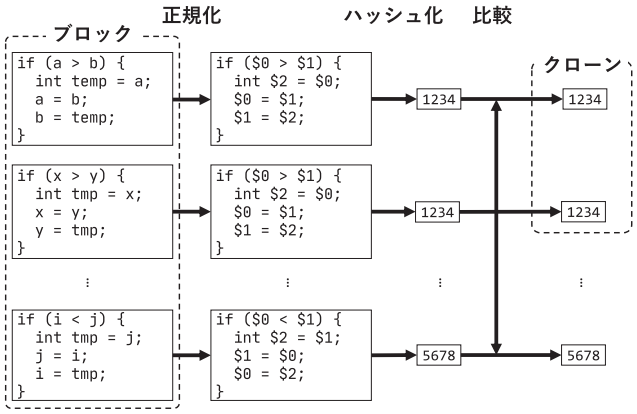


図 5 クローン検出の例
Fig. 5 An example of clone detection.

- ForStatement
- IfStatement
- SwitchStatement
- SynchronizedStatement
- TryStatement
- WhileStatement

ブロック単位で検出されたクローンは字句単位で検出されたクローンと比べると粗粒度であり検出されるクローンの数は少なくなるが，文の集合として検出されるため，メソッドとしての抽出が容易であるという特徴がある．本研究では Eclipse JDT を用いて構文解析を行いソースコード中のブロックを検出する．このとき，return 文を含むコード片に対するメソッド抽出は難しいとされているため，そのようなブロックは検出しない [11]．

識別子の正規化により多くのクローンを検出できるため，検出したブロックに対して以下のルールで正規化を行う．

- 変数名は “$” + “数字” で正規化する．
- 同一の変数名は同一の名前で正規化する．
- リテラルはすべて “$” で正規化する．
- 修飾された変数名は 1 つの変数として正規化する．
- クラス名は正規化しない．
- メソッド名は正規化しない．

同一の変数名に対する同一の名前での正規化により誤検出を防ぐ．クラス名とメソッド名を正規化しない理由は，型名や呼び出しているメソッドが異なるクローンの集約は困難なので，クローンとして検出されるのを防ぐためである．

正規化を行った後にブロックのハッシュ値を計算する．この計算には SHA256 [14] を利用する．SHA256 は 256 ビットのハッシュ値を出力するため，ハッシュ値の衝突の可能性は十分に低いと考えられる．

最後にそれぞれのブロックのハッシュ値を比較し，同じ値のブロックをクローンとして検出する．

Step 2：クローン集約

メソッド抽出を用いて，Step 1 で検出されたクローンセットの 1 つを集約する．クローン集約の例を**図 6** に示す．提案手法では，クローンセットの中から 1 つのコード片を選択し，そのコード片からメソッドを生成する．クローンに対するメソッド抽出により，それぞれのクローンをメソッド呼び出しに変更でき，ソースコード行数を削減できる．抽出されたメソッドは 3.1 節で作成したクラスに宣言する．このメソッドは完全修飾名で呼び出される．

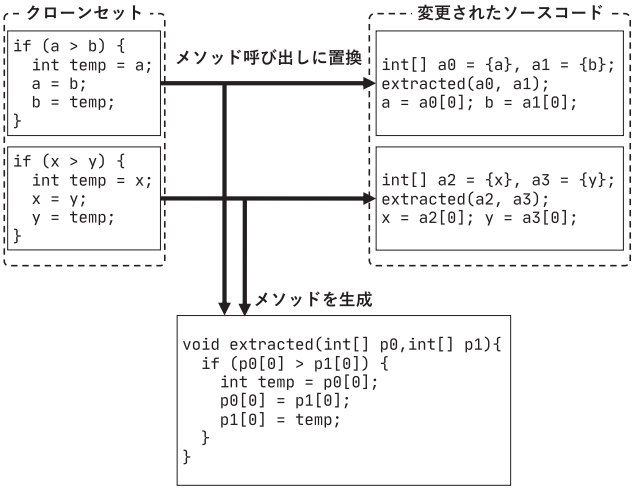


図 6 クローン集約の例
Fig. 6 An example of merging clones.

また，実際のメソッド抽出では，引数として与えた変数がメソッド内で変更されるかに注意する必要がある．たとえば，与えられた複数の引数のうち，そのうちの 1 つだけが変更される場合は，その値を return して，呼び出し側で変数に代入すればよい．逆に，Java の言語仕様上 2 つ以上の値を同時に return できないため，2 つ以上の値が変更される場合はリファクタリングするのは難しいといえる．しかし，本ツールは自動で集約を行うので，その判断を行うのは難しいと考えた．そこで，抽出されたメソッドに参照渡しで引数を与えることで外部的振舞いを保つようにする．引数を参照渡しにするために配列を利用する．メソッド呼び出し前にそのメソッド内で利用される変数の型の配列を定義する．これらの配列を各変数で初期化した後にそれらの配列をメソッドに引数として与える．抽出されたメソッド内では，引数として与えられた配列の添字 0 の要素を参照する．メソッド呼び出し後に元の変数に配列の添字 0 の要素を代入する．これはあくまで実装上の工夫であり，実際にこのような手順でリファクタリングが行われることは少ない．

Step 3：クローンを集約したソースコードのコンパイルとテスト

Step 2 で変更した結果，行数が削減されたソースコードに対してコンパイルとテストを行う．初めにコンパイルを行い，コンパイルに成功すると次にテストを行う．コンパイルとテストのどちらにも成功した場合，変更されたソー

表 2 実験と比較の結果
Table 2 The results of the experiment and the comparison.

プロジェクト名	#検出 CS	提案手法			既存手法	
		#集約対象 CS	削減行数	実行時間	#集約対象 CS	削減可能行数
Ant	583	15	456	1 h 15 m	155	1,246
jEdit	421	58	548	34 m	91	494
JFreeChart	810	176	1,491	2 h 54 m	250	1,963
JMeter.Core	103	22	65	12 m	34	115
JRuby.Core	747	54	570	1 h 40 m	210	1,142

スコードを選択可能なソースコードとして記録する．その後，ソースコードを変更を加える前の状態に戻し，次のクローンセットに対して Step 2 から繰り返す．

コンパイルあるいはテストのどちらかに失敗した場合，そのソースコードは記録せず，成功した場合と同様にソースコードを変更を加える前の状態に戻し，次のクローンセットに対して Step 2 から繰り返す．

Step 4：クローンを集約したソースコードの選択

Step 1 で検出されたすべてのクローンセットに対して Step 2，Step 3 の処理を行った後に Step 4 に移る．Step 4 では，選択可能なソースコードの中から最も削減行数が大きいソースコードを選択する．選択されたソースコードに対して，Step 1 から繰り返す．

5. 実験

5.1 実験概要

提案手法を評価するために，オープンソースプロジェクトに対して提案手法を適用し，得られた結果を既存手法の結果と比較した．また，提案手法と既存手法が集約対象にしたクローンセットの違いを調査した．本研究では，既存研究 [20] が対象にしているプロジェクトのうち，我々の環境[*1]でコンパイルできなかった Columba と Xerces 以外を実験対象として選択した．対象としたプロジェクトの名前，バージョンおよびソースコードの総行数を**表 1** に示す．プロジェクトのテストやチュートリアルなどのソースコードは実験対象に含まない．ソースコードの行数は 4 章で説明したフォーマッタを適用した後の行数である．

表 1 実験対象
Table 1 Experimental targets.

プロジェクト名	バージョン	総行数
Ant	1.10.7	231,634
jEdit	5.5.0	161,329
JFreeChart	1.5.0	210,823
JMeter.Core	5.2	82,360
JRuby.Core	9.2.9.0	360,724

5.2 実験結果

5.2.1 集約対象のクローンセット数と削減行数の比較

対象プロジェクトに対して提案手法を適用し，提案手法が集約対象にしたクローンセット数と計測した削減行数と実行時間，および対象プロジェクトに既存手法を適用し，既存手法が集約対象にしたクローンセット数と推定した削減可能行数を**表 2** に示す．比較の結果から，すべてのプロジェクトにおいて，提案手法が実際に集約したクローンセットの数は，既存手法が集約対象にしたクローンセットの数より小さいことが分かる．また，jEdit 以外のプロジェクトでは，提案手法が測定した削減行数は既存手法が見積もった削減可能行数と比較して少ないことが分かる．これは，集約した際にコンパイルまたはテストを通過しないクローンセットの多くを JDeodorant がリファクタリング可能と判断し，その結果，既存手法がそれらと実際に集約可能なクローンセットの両方を対象に削減可能行数を見積もっているからである．**表 3** に，集約した際にコンパイルとテストを通過するクローンセット（表中 CS1）数，JDeodorant がリファクタリング可能と判定したクローンセット（表中 CS2）数，およびそれらの共通集合の個数を示す．ここで，集約した際にコンパイルとテストを通過するクローンセットは，提案手法が削減行数を測定する対象である．また，JDeodorant がリファクタリング可能と判定したクローンセットは，既存手法が削減可能行数を見積もる対象である．表から分かるように，それぞれのプロジェクトにおいて，集約した際にコンパイルとテストを通

表 3 集約した際にコンパイルとテストを通過するクローンセット（CS1）と JDeodorant がリファクタリング可能と判定したクローンセット（CS2）の個数
Table 3 The number of clone sets that the project passed compile and test when merged (CS1) and clone sets that JDeodorant determined to be refactorable (CS2).

プロジェクト名	# CS1	# CS2	# 共通集合
Ant	225	292	105
jEdit	163	211	118
JFreeChart	377	539	275
JMeter.Core	48	68	40
JRuby.Core	60	396	40

[*1] Ubuntu 18.04，OpenJDK 1.8.0_222，Ant 1.10.5，CPU：8 コア 3.6 GHz，メモリ：64 GB.

過するクローンセット数は，JDeodorant がリファクタリング可能と判定したクローンセット数よりも少ない．以上から，提案手法が測定する削減行数は既存手法が見積もった削減可能行数と比較して少なくなることがあるが，提案手法は集約した際にコンパイルとテストを通過するクローンセットのみを集約の対象としているため，既存手法と比較してより正確に削減行数を測定できると考えられる．

5.2.2 集約対象となったクローンセットの違いの調査

5.2.1 項で削減行数に差異が発生した理由として，削減行数の計算時に集約の対象にしたクローンセットが異なるということがあげられる．そこで，それぞれの手法が集約の対象としたクローンセットの違いを調査した．例として，jEdit に対して提案手法と既存手法を適用した結果から，提案手法が集約したクローンセットと既存手法が集約対象としたクローンセットの数の違いを**図 7** に示す．図から，提案手法が集約したクローンセットと，既存手法が集約対象にしたクローンセットには共通しているクローンセットもあるが，異なるクローンセットもあることが分かる．提案手法では集約の対象になったが既存手法では集約の対象外となったクローンセットと，既存手法では集約の対象となったが提案手法では集約の対象外となったクローンセットについて，例として 4 つのクローンセットを紹介する．

図 8 は `org/gjt/sp/jedit/Buffer.java` 内のコード片であり，`org/gjt/sp/jedit/View.java` 内のコード片とクローンとなっている．このソースコード中で使われている変数 `waitSocket` はどちらも `Socket` 型である．提案手法では，このクローンセットは集約した後にコンパイルとテストに成功している．つまり，実際にリファクタリング可能である．一方，既存手法ではこのクローンセットはリファクタリング不可能と判定している．既存手法が利用している JDeodorant ではリファクタリング可能なクローンを判定するために複数の条件を設けているが，その中に「フィールド変数はパラメータ化できない」という条件が存在する．図 8 で用いられている変数 `websocket` はフィールド変数であり，この条件に違反したため既存手法ではリファクタリング不可能と判定されたと考えられる．このように，JDeodorant の条件に違反するため既存手法ではリファクタリング不可能と判定されたが，実際はリファクタリング可能なため，提案手法では集約されたクローンセットが存在した．

図 9 は `org/gjt/sp/gui/ExtendedGridLayout.java` 内のコード片であり，同じクラス内のコード片とクローン（以下，クローン A）となっている．このクローンセットは 2 つのコード片から構成されていた．一方，図 9 の 775〜779 行，781〜785 行，および 787〜792 行のコード片もそれぞれクローン（以下，クローン B）となっており，このクローンセットは 6 つのコード片から構成されていた．この例では，クローン A がクローン B を包含している．2.4 節で説明した式 (3) を用いると，クローン A を集約した場合 16 行の削減が，クローン B を集約した場合 17 行の削減が見込める．そのため，貪欲法を用いている既存手法では，

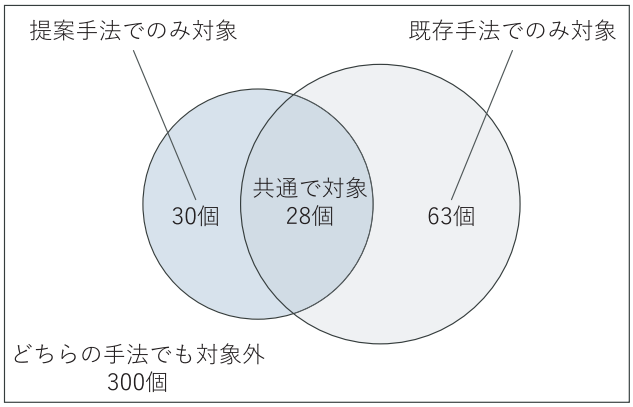


図 7 提案手法と既存手法で集約対象となったクローンセットのベン図

Fig. 7 Venn diagram representing clone sets that are targeted to merge by the proposed technique and/or the existing one.

```
1600 if (waitSocket != null) {
1601  try {
1602   waitSocket.getOutputStream().write('¥0');
1603   waitSocket.getOutputStream().flush();
1604   waitSocket.getInputStream().close();
1605   waitSocket.getOutputStream().close();
1606   waitSocket.close();
1607  } catch {IOException e) {
1608   // Log.log(Log.ERROR,this,io);
1609  }
1610 }
```

図 8 実際にはリファクタリング可能だが，クローン内の変数がフィールド変数なので既存手法では集約の対象外となったクローンの例

Fig. 8 An example of a clone that was actually refactorable, but the existing technique did not target to merge because the variables in the clones were field variables.

```
773 for (int col = fromCol; col < toCol; col++) {
774  int minimumColWidth = minimumColWidths[col];
775  if ((Integer.MAX_VALUE - minimumColWidth) < ...) {
776   currentMinimumColWidth = Integer.MAX_VALUE;
777  } else {
778   currentMinimumColWidth += minimumColWidth;
779  }
780  int preferredColWidth = preferredColWidths[col];
781  if ((Integer.MAX_VALUE - preferredColWidth) < ...) {
782   currentPreferredColWidth = Integer.MAX_VALUE;
783  } else {
784   currentPreferredColWidth += preferredColWidth;
785  }
786  int maximumColWidth = maximumColWidths[col];
787  if ((Integer.MAX_VALUE - maximumColWidth) <  ...) {
788   currentMaximumColWidth = Integer.MAX_VALUE;
789  } else {
790   currentMaximumColWidth += maximumColWidth;
791  }
792 }
```

図 9 オーバラップしているため既存手法では集約の対象外となったクローンの例

Fig. 9 An example of a clone that the exsiting technique did not target to merge because of overlapping.

```
1324 if (splash != null) {
1325  splash.dispose();
1326  splash = null;
1327 }
```

図 10 クローン間の変数の型に継承関係がないためリファクタリングできないが，既存手法では集約の対象となったクローンの例

Fig. 10 An example of a clone that was actually not refactorable because of the type of the variables between the clones has no inheritance relationship, but the existing technique targeted to merge.

```
961 String word = st.nextToken();
962 if (lineLength == leadingWhitespaceWidth) {
963  // do nothing
964 } else if (lineLength + word.length() + 1 > ...) {
965  buf.append('¥n');
966  buf.append(leadingWhitespace);
967  lineLength = leadingWhitespaceWidth;
968 } else {
969  buf.append(' ');
970  lineLength++;
971 }
972 buf.append(word);
973 lineLength += word.length();
```

図 11 実際はリファクタリング可能だが，実装が不完全なために提案手法では集約の対象外となったクローンの例

Fig. 11 An example of a clone that was actually refactorable, but the proposed technique did not target to merge because our implementations are insufficient.

クローン B のみを集約対象とし，クローン A を集約対象にできない．一方，提案手法では，クローン B を先に集約し，その後クローン B のコード片をメソッド呼び出しに置換した後のクローン A を新たに検出したうえで集約し削減行数の算出を行う．このように，貪欲法が原因で既存手法では集約の対象にならなかったが，提案手法では集約されたクローンセットが存在した．

図 10 は `org/gjt/sp/jedit/GUIUtilities.java` 内のコード片であり，`org.gjt.sp.jeit.gui.ActionBar.java` 内のコード片とクローンとなっている．どちらのクラスも `dispose` メソッドが宣言されているが，この 2 つのクラス間に継承関係はない．そのため，実際はリファクタリングできないが，既存手法はリファクタリング可能と判定し，集約の対象とした．一方，提案手法では集約後にコンパイルに失敗したため，集約の対象外とした．

図 11 は `org/gjt/sp/jedit/TextUtilities.java` 内のコード片であり，同じクラス内のコード片とクローンとなっている．このクローンセットは提案手法では集約後にコンパイルに失敗したためリファクタリング不可能と判定されたが，既存手法ではリファクタリング可能と判定されている．このコード片の 965，966，969，および 972 行で変数 `buf` が用いられている．クローンセット中の 2 つのコード片のうち，片方のコード片では変数 `buf` が `StringBuilder` 型であるのに対して，もう片方のコード片は `StringBuffer` 型となっている．これら 2 つの型はどちらも `AbstractStringBuilder` クラスを継承している．そのため，メソッドの仮引数を `AbstractStringBuffer` 型で宣言したうえでメソッド抽出を行えば，このクローンセットはリファクタリング可能である．一方，提案手法では 4.2 節で述べたように，クローンセットの中のコード片の 1 つからメソッドを生成するように実装しているため，仮引数の型を抽象化したうえでの宣言が難しい．これは実装の改善点の 1 つである．

以上から，提案手法には実装の改善が必要な点が存在するが，既存手法と比較すると，実際にリファクタリング可能なクローンセットのみを集約の対象にして削減行数を算出していることが分かる．つまり，提案手法は既存手法と比較してより正確に削減行数を算出できるといえる．

6. 制限

6.1 提案手法の実行における制限

本研究では，提案手法を用いることで，既存手法よりも正確に削減可能行数を算出できることを示した．しかし，削減可能行数を算出する際に利用できる環境やかけられる時間が制限されている場合は，つねに提案手法を利用するのではなく，既存手法を利用すべき場面もあると考えられる．そのような場面に関して，両手法の実行環境と実行時間という観点から議論を行う．

6.1.1 実行環境に関する制限

提案手法の実行には，対象ソフトウェアのコンパイルとテストが必要になる．そのため，対象ソフトウェアのコンパイルとテストを行える環境が限られており，なおかつその環境を用意できる計算機が限られている場合，提案手法の実行は難しい．そのような場合，既存手法を用いるべきだといえる．

一方，既存手法の実行には，外部ツールである CCFinderX [9] と JDeodorant [17] が必要になる．そのため，これらの外部ツールを利用する機材を用意できない場合は，提案手法を利用すべきだといえる．

6.1.2 実行時間に関する制限

本研究では，数十万行規模のソフトウェアに対して，実用的な時間で既存手法より正確に削減可能行数を測定できることを示した．したがって，数十万行規模のソフトウェアに対しては，既存手法ではなく提案手法を用いるべきだと考えられる．

一方，コンパイルやテストに長時間を要するような大規模なソフトウェアに対して削減可能行数を測定する際には，必ずしも提案手法を用いるべきだとはいえない．なぜなら，提案手法はコンパイルとテストを繰り返し行うので，実用的な時間で削減可能行数を測定できない可能性があるからである．対象のソフトウェアが増分ビルドを利用しており，2 回目以降のコンパイル，テストを高速に行える場合は提案手法を利用できるが，それ以外の場合は既存手法

の利用を検討すべきだといえる．

6.2 実装における制限

本研究では，提案手法を実装する際に様々な制限を与えている．与えた制限は以下のとおりである．

抽出されたメソッドはすべて 1 つのクラスに宣言される

本研究の実装では，リファクタリングによって抽出されたメソッドはすべて 1 つのクラスに宣言される．これは，複数クラスにまたがったクローンを集約する際に，抽出されたメソッドを宣言するクラスを自動で判断するのは困難だと考えたために与えた制限である．開発者がリファクタリングを行う場合は，開発者の判断により適切なクラスにメソッドを宣言すると考えられる．

クローンはブロック単位で検出される

本研究の実装では，リファクタリング対象であるクローンはブロック単位で検出される．これは，ブロック単位で検出したクローンはメソッドの抽出が容易であると考えたために与えた制限である．開発者がリファクタリングを行う場合は，必ずしもブロック単位ではなく，連続する文などを抽出の対象とする場合もある．

return 文を含むコード片はクローンとして検出しない

本研究の実装では，return 文を含むコード片はクローンとして検出しない．これは，return 文を含むコード片に対してメソッド抽出を自動で行うのは困難だと考えたために与えた制限である．開発者が必ずしも return 文を含むクローンのリファクタリングを行わないとは限らない．

抽出されたメソッドに参照渡しで引数を与える

本研究の実装では，リファクタリングによって抽出されたメソッドに対して，引数を参照渡しで与えている．これは，引数として与えた変数がメソッド内で変更されているかを自動で判断するのが困難だと考えたからである．開発者がリファクタリングを行う場合は，引数として与えた変数がメソッド内で変更されているかを確認し，変更されている変数を return するなどの修正を行うと考えられる．

7. おわりに

本研究では，クローンに対して実際にリファクタリングを行い，その結果からより正確な削減可能なソースコード行数を測定する手法を提案した．提案手法では，対象のソースコードに対してクローンの検出，集約，ソースコードのコンパイル，テストを繰り返し自動で行う．また，実験の結果，提案手法は既存手法と比較して，実際にリファクタリング可能なクローンのみ集約を行い，より正確に削減可能行数を算出できることを示した．

今後は次のような課題を解決していく予定である．

複数のリファクタリング手法の採用

本研究ではクローンの集約手法としてメソッド抽出のみを使用した．しかし，クローンに対するリファクタリング手法はメソッド抽出以外にも多くの手法が提案されている．たとえば，メソッドの引き上げやテンプレートメソッドの形成などがあげられる．これらの手法の採用により，より正確な削減行数の算出が期待できる．

実装の改善

5.2.2 項で述べたように，本研究ではコンパイルエラーとなったが，実装の工夫次第ではコンパイルが可能になるようなクローンセットが存在する．このようなクローンセットをリファクタリングできるように実装を改善することで，より正確な削減行数の算出が期待できる．

大規模ソフトウェアに対する実行時間の計測

本研究では，提案手法は数十万行規模のソフトウェアに対して実用的な時間で削減可能行数を測定できることを示した．一方，コンパイル，テストに長時間を必要とするような大規模なソフトウェアに対する実行時間の計測は行っていない．そのようなソフトウェアに対しても提案手法が適用できるかどうかの確認は今後の重要な課題である．

謝辞 本研究は，日本学術振興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] Cordy, J.R. and Roy, C.K.: The NiCad Clone Detector, *2011 IEEE 19th International Conference on Program Comprehension*, pp.219–220 (2011).
[2] Eclipse, available from ⟨https://www.eclipse.org/⟩.
[3] Eclipse Java Development Tools, available from ⟨https://www.eclipse.org/jdt/⟩.
[4] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S. and Mockus, A.: Does code decay? Assessing the evidence from change management data, *IEEE Trans. Software Engineering*, Vol.27, No.1, pp.1–12 (2001).
[5] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Signature Series, Pearson Education (1999).
[6] Goon, A., Wu, Y., Matsushita, M. and Inoue, K.: Evolution of code clone ratios throughout development history of open-source C and C++ programs, *2017 IEEE 11th International Workshop on Software Clones*, pp.1–7 (2017).
[7] Higo, Y., Matsumoto, S., Kusumoto, S., Fujinami, T. and Hoshino, T.: Correlation Analysis between Code Clone Metrics and Project Data on the Same Specification Projects, *Proc. 12th International Workshop on Software Clones*, pp.37–43 (2018).
[8] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. 29th International Conference on Software Engineering*, pp.96–105, IEEE Computer Society (2007).
[9] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Software Engineering*, Vol.28, No.7, pp.654–670 (2002).
[10] Kim, M., Sazawal, V., Notkin, D. and Murphy, G.: An Empirical Study of Code Clone Genealogies, *SIGSOFT Softw. Eng. Notes*, Vol.30, No.5, pp.187–196 (2005).
[11] Komondoor, R. and Horwitz, S.: Effective, automatic

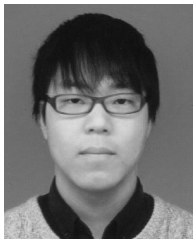
procedure extraction, *11th IEEE International Workshop on Program Comprehension*, pp.33–42 (2003).
[12] Koschke, R. and Bazrafshan, S.: Software-Clone Rates in Open-Source Programs Written in C or C++, *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol.3, pp.1–7 (2016).
[13] Lientz, B.P. and Swanson, E.B.: *Software Maintenance Management*, Addison-Wesley Longman Publishing Co., Inc. (1980).
[14] National Institute of Standards and Technology: Secure Hash Standard (2015).
[15] Pigoski, T.M.: *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, Wiley Publishing, 1st edition (1996).
[16] Sneed, H.M.: Planning the reengineering of legacy systems, *IEEE Software*, Vol.12, No.1, pp.24–34 (1995).
[17] Tsantalis, N., Mazinanian, D. and Krishnan, G.P.: Assessing the Refactorability of Software Clones, *IEEE Trans. Software Engineering*, Vol.41, No.11, pp.1055–1090 (2015).
[18] Yoshida, N., Ishizu, T., Edwards, III, B. and Inoue, K.: How Slim Will My System Be?: Estimating Refactored Code Size by Merging Clones, *Proc. 26th Conference on Program Comprehension*, pp.352–360 (2018).
[19] 井上克郎，神谷年洋，楠本真二：コードクローン検出法，コンピュータソフトウェア，Vol.18, No.5, pp.529–536 (2001).
[20] 石津卓也，吉田則裕，崔　恩瀞，井上克郎：コードクローンのリファクタリング可能性に基づいた削減可能ソースコード量の分析，情報処理学会論文誌，Vol.60, No.4, pp.1051–1062 (2019).
[21] 肥後芳樹，吉田則裕：コードクローンを対象としたリファクタリング，コンピュータソフトウェア，Vol.28, No.4, pp.4_43–4_56 (2011).

中川 将

2019 年大阪大学基礎工学部情報科学科卒業．現在，同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博士前期課程在学中．コードクローン分析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，ソフトウェアリポジトリマイニングおよび自動プログラム修正に関する研究に従事．日本ソフトウェア科学会，IEEE 各会員．



松本 淳之介

2018 年大阪大学基礎工学部情報科学科卒業．2020 年同大学大学院情報科学研究科博士前期課程終了．在学中，リポジトリマイニングに関する研究に従事．

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．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．IEEE, IFPUG 各会員．