情報処理学会論文誌 Vol.60 No.4 1051–1062 (Apr. 2019)

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

石津 卓也[1,a)] 吉田 則裕[2] 崔 恩瀞[3] 井上 克郎[1]

受付日 2018年7月31日, 採録日 2019年1月15日

概要：コードクローンはソフトウェア保守を困難にさせる要因の 1 つといわれている．また，コードクローンのリファクタリングとは，外部的な振舞いを保ちながら，コードクローンを削減するプロセスであり，ソフトウェア保守性を上げる可能性がある．企業が請負うソフトウェアの保守性向上をサービスとして，ソースコードのリファクタリングが検討されることがある．しかし，コードクローンのリファクタリングによる削減できる開発のコストを見積もる際，複数の課題が発生する可能性があるため，その見積もりが難しくなることがある．そこで，本研究では，リファクタリングの見積もりに関する課題を解決して，コードクローンをリファクタリングしたと仮定したときの削減可能ソースコード量を推定する手法を提案する．また，7 つのオープンソースソフトウェア（OSS）に対して削減可能ソースコード量を推定し，コードクローンの行数と比較して平均 6.9%であることが分かった．

キーワード：コードクローン，リファクタリング可能性，削減可能ソースコード量，オープンソースソフトウェア

Analyzing the Amount of Reducible Source Code Based on the Refactorability of Software Clones

Takuya Ishizu[1,a)] Norihiro Yoshida[2] Eunjong Choi[3] Katsuro Inoue[1]

Received: July 31, 2018, Accepted: January 15, 2019

Abstract: Code clone makes software maintenance more difficult. Refactoring of code clones is one process of reducing code clones while maintaining external behavior. It is able to increase the software maintainability. When a company which is commissioned to maintain software has to estimate the cost and benefit of refactoring source code owned by a customer. They have trouble preparing their estimates because of refactoring code clones that have several problems is difficult. We proposed a solution to each problem and applied a method to investigate “the amount of reducible source code” for seven open source software products. As a result of the investigation, we found that approximately 6.9% of the detected code clones could be reduced.

Keywords: code clone, refactorable, the amount of reducible source code, open source software

1. まえがき

ソフトウェア保守を困難にさせる問題の 1 つとして，コードクローンの存在があげられる [6]．コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことである [9]．既存のコード片をコピーアンドペーストによって再利用することがコードクローンの主要な発生原因として知られている．コードクローンはソフトウェア保守を困難にさせる要因の 1 つといわれている [9]．たとえば，開発者が一部のコードクローンに変更を行った場合，変更されなかったコードクローンはそれ以降では望まれない動作を行う可能性をプログラム内に残す

[1] 大阪大学
Osaka University, Suita, Osaka 565–0871, Japan
[2] 名古屋大学
Nagoya University, Nagoya, Aichi 464–8601, Japan
[3] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] t-ishizu@ist.osaka-u.ac.jp

ことになる．そのため，潜在的なバグを生み出しにくいプログラム構造を実現するために，コードクローンを削減させる必要がある [5]．また，コードクローンを削減するための研究が多く提案されてきた [1], [2], [3], [15], [16], [18]．コードクローンの削減する代表的な方法としてコードクローンのリファクタリングがある [8]．

コードクローンのリファクタリングとは，外部的な振舞いを保ったまま，コードクローンを 1 つのメソッドにまとめるプロセスを指す [15]．コードクローンに対するリファクタリングのプロセスは 2 つの作業によって実現できる．1 つは，コードクローンをまとめるための共通のメソッドを新規に作成する作業である．もう 1 つは，コードクローン自体は共通メソッドの呼び出し文に置換する作業である．

近年，保守性が低下した商業用ソフトウェアの保守性を向上させたいという需要に応えるために，他の企業のソフトウェアの保守性向上をサービスとして提供する企業が存在する．また，これらの企業は効率的な保守活動を実現するためにソフトウェアのリファクタリングを提供する[*1]．このリファクタリングには，潜在的なバグの除外やデッドコードの特定，冗長なソースコードの削減などがあげられる．冗長なソースコードを削除する一環として，コードクローンのリファクタリングがある．依頼元企業と請負先企業との間では，リファクタリングにおける削減行数や金銭的および時間的コストなど見積もってから契約が行われる．

しかし，一般的に，巨大化したソースコードはその構造の理解が難しいため，コードクローンのリファクタリングを見積もる際に次の 2 つの問題が生じうる．(1) リファクタリングすべきコードクローンを特定することは困難であることと (2) 検出したコードクローンの位置情報が部分的にオーバラップすることである．これらの問題はリファクタリング可能な行数の見積もりに大きく誤差が生じさせ，ソフトウェアの保守性向上のためのサービスに支障をきたすことがある．問題 (1) を回避するために容易にリファクタリング可能性を判断できるコードクローンにのみ焦点を当てる方法が考えられる．しかし，容易にリファクタリング可能性を判断できるコードクローンは，ソースコードがそれほど複雑ではないなどの理由があり，本質的な解決には至らない．そのため，巨大化したソースコードを対象とした，コードクローンのリファクタリング可能性に注目した推定手法が必要になる．また，問題 (2) に対して，再帰的なリファクタリングを適用することでコードクローンを削減できる可能性がある．しかし，再帰的なリファクタリングは必ずしもすべてのオーバラップしているコードクローンに対応できるわけではない．また，再帰的にリファクタリングを適用した場合，メソッドを呼び出す階層が深くなる可能性もある．

[*1] http://info.hitachi-ics.co.jp/product/rf/index.html

そこで，本研究では，コードクローンのリファクタリングによる削減可能ソースコード量を見積もるための手法を提案する．本手法は次のとおり問題を (1) と (2) を解決した．

まず，問題 (1) を解決するために，コードクローンのリファクタリング可能性を自動で判断するために JDeodorant [18] を利用した．JDeororant はその機能の 1 つにクローンペア（互いに一致または類似しているコード片の対）からリファクタリング可能性で評価する．

問題 (2) を解決するために，削減可能ソースコード量の大きさに着目して，オーバラップしているコードクローンの中から削減可能なソースコード量が大きなコードクローンを優先的にリファクタリングする方針を採用した．この方針に従う場合，見積もりに必要な時間計算量はコードクローンの個数にともなって増加する．そのため，我々はオーバラップしているコードクローンに対して，メタヒューリスティックによる近似アルゴリズム [4], [17] を適用し，リファクタリングするべきコードクローンの組合せを選択的に決定する．

また，提案手法を用いて，7 つのオープンソースソフトウェア（以下，OSS とする）の削減可能ソースコード量を調査した．また，提案手法の妥当性を確認するために OSS である JEdit，JMater，Apache Ant に含まれる 80 個のリファクタリング可能なクローンセットに対して，手動でリファクタリングした結果と提案手法による結果を比較した．

以降，2 章では研究の背景について説明し，3 章では提案手法について述べる．4 章では削減可能ソースコード量の調査の方法と結果を説明し，5 章でまとめと今後の課題について述べる．

2. 背景

2.1 コードクローン

コードクローンとは，ソースコード中に存在する互いに一致または類似した部分を持つコード片のことである [6], [9]．また，互いに一致または類似しているコード片の対をクローンペアと呼び，コードクローンの集合をクローンセットと呼ぶ．また，コードクローンは次の 3 つのタイプ分類できる．

- **タイプ 1** 空白行やコメント行などのコーディングスタイルを除いて完全に一致するコード片を持つ．
- **タイプ 2** 変数名や関数名，変数の型などの識別子のみが異なるコードクローンである．
- **タイプ 3** タイプ 2 であるうえに命令文の挿入や削除，変更が行われたコードクローンを指す．

コードクローンの主要な発生原因として既存のコード片をコピーアンドペーストにより再利用することがあげられる．コードクローンはソースコード行数を増大させ，ソフトウェア保守を困難にする [6]．たとえば，コードクロー

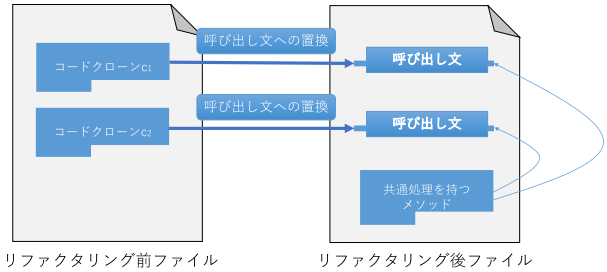

図 1 コードクローンのリファクタリング
Fig. 1 Refactoring code clone.

ンに修正が加えられる場合，そのコードクローンと同一のクローンセットに属するコードクローンに対して一貫性のある修正を検討する必要性がある．もしすべてのコードクローンに対して一貫した修正が行われてなかった場合は，バグの原因が生じる可能性がある．そのため，コードクローンの効率的管理をする必要がある．しかし，ソフトウェアの規模が大きくなるにつれて，開発者の意図しないコードクローンが発生したり，複数の開発者で開発したりするため，すべてのコードクローンの位置を把握するのは難しくなる．したがって，コードクローンを自動で検出するツールが多く提案されてきた [11], [13], [14], [20].

2.2 削減可能ソースコード量

コードクローンを削減させる方法として，リファクタリングと呼ばれるプロセスをコードクローンに適用することができる [15]．コードクローンのリファクタリングとは，コードクローンを共通のメソッドやクラスにまとめて，コードクローンは呼び出し文に置換して削減するというプロセスである．

図 1 はコードクローンのリファクタリングの例を表している．コードクローンを図 1 では，コードクローン C1 と C2 の共通の処理部分が新しいメソッドとして抽出され，コードクローン C1 と C2 は共通のメソッドの呼び出し文に置き換えられている．コードクローンのリファクタリングを実施する際，どこに共通処理のコード片をまとめるのか判断するには，各コードクローンが存在する階層構造や継承関係などを考慮する必要がある．そのため，必ずしも同一ファイル内に作成をすべきとは限らない．

コードクローンをリファクタリングするために，開発者はコードクローンの位置や階層構造に着目して，外部的な振舞いが変わらないように行う必要がある．このコードクローンのリファクタリングにおけるソースコードの削減効果を事前に推定することが求められている．この背景には，保守性が低下したソフトウェアの保守性向上をサービスとして提供する企業が存在するという事例があげられる．特に，複雑に巨大化したソースコードにおいて，その保守性は低下している場合が多く，それらから検出されたコードクローンのリファクタリングに費やすコストや得られる効果は事前に推定するのが困難であるという現実がある．したがって，開発者がコードクローンのリファクタリングにおける削減効果の推定結果は，コードクローンのリファクタリングを行う動機の判断指標として利用する場面が考えられる．本稿では，その判断指標として削減可能ソースコード量を定めた．削減可能ソースコード量とは，仮にコードクローンをリファクタリングした場合に，コードクローンの削除によって元のソースコードから減少すると予想されるソースコード行数の推定値を表す．

削減可能ソースコード量の評価基準としては，コードクローンの個数やトークン長，行数などが候補にあげられる．そのため，リファクタリング前後のソースコードの変化をどのようにすれば，効果的に表現できるのか考える必要がある．

また，ソースコード全体から検出されたコードクローンの削減可能ソースコード量を求めるには，以下の 2 つの問題を解決するための条件を満たすクローンセットを求める必要がある．2 つの問題は 2.2.1 項と 2.2.2 項でそれぞれ説明する．

- リファクタリング可能性
- オーバラップ

2.2.1 リファクタリング可能性の問題

コードクローンのリファクタリングは一部のコードクローンのみに可能であり，リファクタリングが可能である，またはリファクタリングが困難であるといった評価をリファクタリング可能性と呼ぶ [18]．リファクタリングが困難である例として，メソッドの返り値は最大で 1 つの型しか持てないが，コードクローンによっては条件分岐によって必ずしも同じ型の返り値を持つとは限らない状況や，それぞれのクラスがディレクトリ構造で見たときに非常に離れている状況があげられる．このようにリファクタリング可能性を判断するために必要な条件は複数存在する．ただし，コードクローンのリファクタリング可能性はプログラム言語の機能に大きく依存すると考えられる．

2.2.2 オーバラップの問題

コードクローン検出ツールは，互いにソースコードが重なっているコードクローンを検出することがある．このような状態をオーバラップと呼ぶ．また，クローンセットに属するコードクローン c_{a1} が別のクローンセットに属するコードクローン c_{b1} とオーバラップしているとき，それらのクローンセットはオーバラップしていると表す．オーバラップしている c_{a1} と c_{b1} を含むクローンセットを同時にリファクタリングした場合，それぞれのコードクローン c_{a1} と c_{b1} の共通処理を持つメソッド間では重複したコード片が含まれてしまい，同じ処理を 2 回実行してしまう．そのため，これらのコードクローンを同時にリファクタリングすることはできない．したがって，もしオーバラップしているクローンセットの削減可能ソースコード量を重複して

Ca1	Cb1	Cb2	Cc1		
+	+			1	if(bool1){
+	+			2	...
+	+			3	}
+		+		4	if(bool2){
+		+		5	...
+		+		6	}
+			+	7	while(loop){
+			+	8	...
+			+	9	}

図 2 オーバラップしている簡単なソースコード例
Fig. 2 Simple example of overlapping of source code.

数えてしまった場合，オーバラップしているクローンセットの削減可能ソースコード量と実際の削減行数とは異なる可能性がある．この問題を解決するために，コードクローンが互いにオーバラップしている場合，リファクタリングするコード片を分割したり，一方だけをリファクタリングしたりするといった工夫が必要である．

図 2 は，オーバラップしているコードクローンの例を示す．この表は，3 つのクローンセット a，b，c にそれぞれ属する 4 つのコードクローン c_{a1}，c_{b1}，c_{b2}，c_{c1} について対応する関係を示している．クローンセット a はコードクローン c_{a1} を持つ．クローンセット b はコードクローン c_{b1}，c_{b2} を持つ．クローンセット c はコードクローン c_{c1} を持つ．

図 2 に示すソースコードは 2 つの IF 文（1 行目から 3 行目と 4 行目から 6 行目）と 1 つの WHILE 文（7 行目から 9 行目）によって構成されている．クローンツールは検出条件としてしきい値などを満たしているコードクローンであれば，図 2 のように 3 つのクローンセットに分けて検出する可能性がある．このとき，コードクローン c_{a1} は c_{b1}，c_{b2}，c_{c1} のそれぞれとオーバラップしている．また，クローンセットに基づいて表現すると，クローンセット a と b，a と c はそれぞれオーバラップしていると表現できる．

3. 提案手法

この章では，ソフトウェアから検出されたコードクローンをリファクタリングした場合の削減可能ソースコード量を算出する手法について説明する．**図 3** は提案手法の概要を表している．提案手法は図 3 に示すように．5 つのステップから構成される．

(1) コードクローンの検出
(2) コードクローンのリファクタリング可能性の計測
(3) リファクタリング可能なクローンセットの抽出
(4) クローンセット 1 つの削減可能ソースコード量の計算
(5) オーバラップの解決

以降の節でそれぞれのステップの詳細について説明する．

3.1 ステップ 1：コードクローンの検出

まず，最初にコードクローンを検出する．本研究ではコードクローンの検出をするために CCFinderX [12] を用いた．CCFinderX は様々な大規模ソフトウェアに適用されており [7]，高いスケーラビリティ性を示しているため，本研究で利用した．CCFinderX は 2.1 節で説明したタイプ 1 とタイプ 2 のコードクローンを検出することができる．

また，タイプ 3 のコードクローンについては，本研究では対象に含まない．タイプ 3 のコードクローンをリファクタリングする際には一部の行をコードクローンの範囲外へ移動させる場合が存在している．しかし，3.4 節で説明する計算方法では，移動作業を考慮をしていないので削減可能ソースコード量を見積もれないからである．よって，タイプ 3 のコードクローンを対象とした計算は今後の課題である．

3.2 ステップ 2：コードクローンのリファクタリング可能性の計測

このステップでは，検出されたコードクローンのリファクタリング可能性を計測する．本研究ではリファクタリング可能性を計測するために JDeodorant を使用した [18], [19]．JDeodorant はリファクタリングを支援する Eclipse プラグインとして開発されリファクタリング可能性を判断する機能を提供する．このツールは複数のコードクローンがリファクタリング可能になるのに必要な前提条件とネスト構造木，プログラム依存グラフを利用することによりリファクタリング可能性を非常に高い精度で評価する．

JDeodorant が計測するリファクタリング可能性について説明する．JDeodorant はクローンペアのリファクタリング可能性の前提条件を以下のように示している [18]．これらの前提条件に 1 つでも違反していた場合，そのクローンペアはリファクタリング困難と見なされる．

文献 [16] では，11 名のプログラマが Eclipse 上で様々な条件をもとにリファクタリングできるのか観察する被験者実験が行われた．その中で下記で示した条件を満たさないリファクタリングは困難であると述べられている．そのため，本研究においても文献 [16] で定められた前提条件に従う．

(1) 変数のパラメータ化の際に制御依存，データ依存，反復操作や出力の振舞いに変更がない必要がある．
(2) それぞれ異なる子クラス型を持つ変数は，共通の親クラスで宣言されているか，あるいはオーバライドされたメソッド内でのみ呼び出されている必要がある．
(3) フィールド変数のパラメータ化は，そのフィールド変数が変更可能ならば許可しない．
(4) メソッド呼び出しのパラメータ化は，void 型を返さないときのみ可能である．
(5) 抽出されたメソッドは，2 つ以上の変数を返さない必

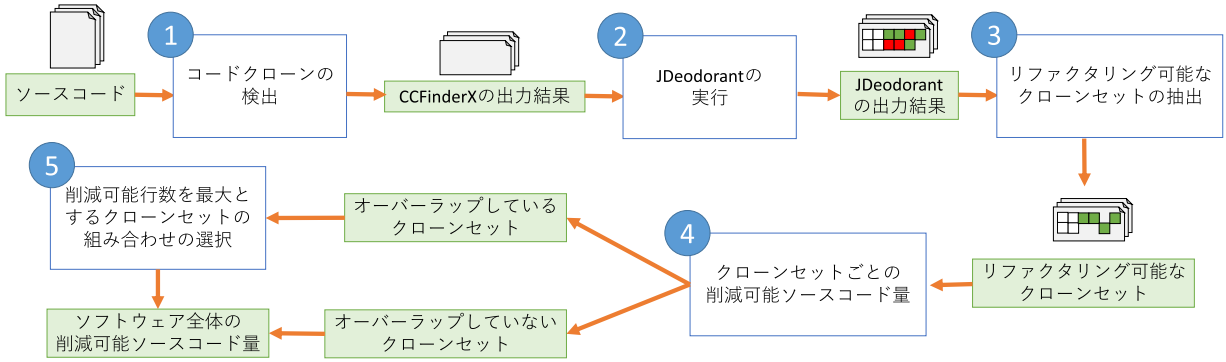

図 3 削減可能ソースコード量算出の概要図
Fig. 3 Overview of calculating the amount of reducible source code.

要がある．

(6) 条件付き return 文がコードクローンのコード片に含まない必要がある．

(7) 分岐処理を意味する命令文（BREAK，CONTINUE）があれば，それに対応する反復命令文がコード片を含まない必要がある．

3.3 ステップ 3：リファクタリング可能なクローンセットの抽出

次に，リファクタリング困難なコードクローンを除外する．ただし，JDeodorant はクローンペア単位でリファクタリング可能性を評価している．そのため，同じクローンセットに属するコードクローンでも，特定の組合せの場合はリファクタリング困難になっている場合がある．そこで，同じクローンセット内でリファクタリング可能であると計測されたクローンペアを 1 つも含んでないコードクローンをリファクタリング困難なコードクローンとして除外する．そして，1 回以上リファクタリング可能と計測されたクローンペアについては，そのどちらのコードクローンもリファクタリング可能と見なす．

3.4 ステップ 4：クローンセットごとの削減可能ソースコード量の計算

このステップでは，リファクタリング可能なコードクローンのみが含まれるクローンセットごとの削減可能ソースコード量を求める．本研究では，削減可能ソースコード量の単位はソースコード行数とする．まず，削減可能ソースコード量をクローンセットごとに計算する目的について説明する．ここで求めるクローンセットごとの削減可能ソースコード量は次のステップ 5-2 において，オーバラップしているクローンセットに対してメタヒューリスティックを適用するために用いる．また，オーバラップしていないクローンセットの削減可能ソースコード量についてはこのステップ 4 で求めた削減可能ソースコード量を用いる．次に，クローンセットごとの削減可能ソースコード量の算出式について説明する．ある 1 つのクローンセット S に属する n 個のコードクローン c_i を入力として得られる削減可能ソースコード量 $T(S)$ は次の式 (1)〜(3) によって計算される．

$$size(c) = l_{end}(c) - l_{begin}(c) + 1 \tag{1}$$

$$c^*_{size} = \frac{1}{n}\sum_{c_i \in S}^{n} size(c_i) \tag{2}$$

$$T(S) = n * c^*_{size} - (c^*_{size} + 2 + n) \tag{3}$$

式 (1) において，$size(c)$ はコードクローン c が存在する行数を表し，$l_{end}(c)$ と $l_{begin}(c)$ はそれぞれコードクローン c の終了行番号と開始行番号を示している．これらの情報はクローン検出ツールによって得られる．式 (2) は，n 個のコードクローンの平均行数 c^*_{size} を示している．これは同一クローンセットに属するコードクローンであっても，開発者ごとに改行などの好みのコーディングスタイルが存在するためにこれらのコードクローンが異なる行数を持っている可能性がある．そのために，本研究では平均的なコードクローン c^* を用いた．

式 (3) で求められる削減可能ソースコード量 $T(S)$ は減算演算子を境目にその前後で別々のソースコードの状態を計算する．式 (3) の第 1 項では，コードクローンのリファクタリングが行われる前のソースコードに含まれるクローンセット S に属する n 個のコードクローン c_i 全体のソースコード量を表している．減算演算子以降では，2 つの観点で計算されている．1 つ目は $c^*_{size} + 2$ の部分であり，ここではコードクローンの共通処理が抽出されたメソッドの行数を表している．この式で 2 が加えられているのは，メソッドの導入文および終了文を追加されるためである．なお，ここでの計算は主に Java 言語を対象としているが，ほかの言語では必ずしも 2 行の追加でサブルーチンの記述が完了するわけではないことに気を付けたい．

式 (3) の計算結果は負の数，すなわちコードクローンをリファクタリングすることでソースコード全体の行数が増

加する場合が考えられる．本研究では，ソースコードをどれだけ縮小させられるのかが調査することが目的であるため，リファクタリングすることで全体のコードが長くなるコードクローンを選択しない仕組みを採用している．

2 つ目の式は n であり，これはすべてのコードクローンのコード片を呼び出し文に置換した場合に必要な行数と想定している．この本研究では，Java 言語においてメソッドの呼び出し文は一行で完結するものと想定しているが，他の言語では必ずしもそうではない．これら 2 つの計算部分がコードクローン削除によるソフトウェアの振舞いの変更を防ぐための記述に必要な最低限の行数である．よって，元のコードクローンの行数とこれらの差が削除可能ソースコード量である．

3.5 ステップ 5：削減可能行数を最大とするクローンセットの組合せの選択

最後にソースコード全体の削減可能ソースコード量の計算するために，前のステップ 4 で計算したクローンセットごとの削減可能ソースコード量を用いて，オーバラップしているクローンセットとオーバラップしていないクローンセットに分けてそれぞれの削減可能ソースコード量を計算して，最後に足し合わせる．オーバラップしていないクローンセットはクローンセットごとの削減可能ソースコード量をすべて足し合わせればよいが，オーバラップしているクローンセットは同時にリファクタリングして削減可能行数を貪欲法を用いてクローンセットの組合せを選択する．よって，オーバラップを解決する手順は次のとおりである．

- ステップ 5-1 オーバラップの検出
- ステップ 5-2 メタヒューリスティックの適用

ステップ 5-1:オーバラップしているコードクローンの検出

2 つのクローンセットにそれぞれ属するコードクローン c_1，c_2 のオーバラップの検出は次のような条件を調べることで可能になる．なお，あるクローン c の開始行を $t_b(c)$，終了行を $t_e(c)$ とする．(1) コードクローン c_1，c_2 がオーバラップしない場合：

$$t_e(c_1) < t_b(c_2) \vee t_e(c_2) < t_b(c_1)$$

(2) コードクローン c_1，c_2 がオーバラップする，かつ，包含関係にない場合：

$$(t_b(c_1) < t_b(c_2) \wedge t_e(c_1) > t_b(c_2) \wedge t_e(c_1) < t_e(c_2)) \vee (t_b(c_2) < t_b(c_1) \wedge t_e(c_2) > t_b(c_1) \wedge t_e(c_2) < t_e(c_1))$$

(3) コードクローン c_1，c_2 がオーバラップする，かつ，包含関係にある場合：

$$(t_b(c_1) < t_b(c_2) \wedge t_e(c_1) > t_e(c_2) \vee (t_b(c_2) < t_b(c_1) \wedge t_e(c_2) > t_e(c_1))$$

(1) はオーバラップしないための論理式である．(2) と (3) はどちらもオーバラップをする関係を示した論理式であるが，包含関係にあるか否かで場合分けをしている．

ステップ 5-2：メタヒューリスティックの適用

本手法では，オーバラップしているコードクローンの中から同時にリファクタリングして削減可能行数を最大とするクローンセットの組合せを選択するために，メタヒューリスティックな手法を採用した [4], [17]．我々の先行研究 [10] では，Java 言語の OSS に対して 4 つのメタヒューリスティックなアルゴリズムを用いて，削減可能ソースコード量を推定した．その結果，これらのアルゴリズムによる推定値の差はほぼないことが判明した．そこで本研究では，実行時間が最も短い貪欲法を採用した．

貪欲法によるクローンセットの組合せを決定するアルゴリズムについて要点を説明する．詳細は文献 [10] である．また，この詳細は文献 [4], [17] に従い提案したものである．提案手法では，メタヒューリスティックと呼ばれる組合せ最適化問題における，近似解を求める解放を持つアルゴリズムの基本的な枠組みを用いている．このメタヒューリスティックでは，対象問題の表現方法，状態の更新条件，評価値の決定をユーザが決める．貪欲法では，オーバラップしているクローンセットに対して，評価値を式 (3) で求められる削減可能ソースコード量 $T(S)$ として組合せの候補として選択する．

オーバラップしていないクローンセットと貪欲法により解の候補に含まれるすべてのコードクローンに対するリファクタリングによる削減可能ソースコード量の総和が対象ソフトウェアの削減可能ソースコード量として求められる．

自身とのオーバラップ

オーバラップにおける考慮をしなければならない状況がある．クローンセットのオーバラップには，同一のクローンセット内で生じる場合もある．この現象は SWITCH 文など同じ命令文が連続しやすいコード片で生じやすい．同じ命令文が続くコード片であるため，コード片を分割すればリファクタリングできそうではあるが，本研究ではこのような現象が起きるクローンセットを調査対象から除外する．これは削減可能ソースコード量の算出手法に適用できないリファクタリングである可能が高いためである．

4. 調査

我々は提案手法を 7 つの OSS の削減可能ソースコード量を本手法を用いて推定している．また，提案手法の妥当性を確認するために JEdit [*2]，JMeter [*3]，Apache Ant [*4] に含まれる 80 個のリファクタリング可能なクローンセット

[*2] http://www.jedit.org/
[*3] https://jmeter.apache.org/
[*4] https://ant.apache.org/

表 1 調査対象の OSS（*単位は *kLoC*）
Table 1 Statistics of the analyzed OSS systems (*unit is kLoC).

プロジェクト名	バージョン	行数*	CLoC*（%）
Ant	1.10.1	268	60（22.3）
Columba	1.4	54	4.6（8.5）
JMeter	3.2	91	5.6（6.1）
JEdit	5.4.0	180	1.8（1.0）
JFreeChart	1.0.19	310	175（56.4）
JRuby	1.7.27	325	61（18.8）
Xerces	2.10.0	238	83（34.9）

に対して，手動でリファクタリングした結果と提案手法による結果を比較した．

4.1 各オープンソースソフトウェアに含まれる削減可能ソースコード量の分析

動機

提案手法が実際の OSS から検出されたコードクローンの行数に対してどのくらい削減できるのかを分析する必要がある．リファクタリング困難なコードクローンやオーバラップするコードクローンがどの程度の頻度で出現するのかはプロジェクトごとに異なる．たとえば，すべてのコードクローンのリファクタリング可能性やオーバラップを考慮せずに，リファクタリングできるコードクローンの行数を見積もった場合，その見積もりはコードクローンの数に比例すると予測できる．しかし，リファクタリング可能性やオーバラップの影響を考慮した場合，比例するかどうかは調査すべき事項であると考える．

調査対象

本調査で対象とした OSS について説明する．すべてのプロジェクトが主に Java 言語で記述されている．

表 1 は，対象とした 7 つの OSS プロジェクトとその行数，コードクローンが占める行数（CLoC）を表している．本研究では，既存研究 [10], [18] で対象となったプロジェクトの中から，Java 言語で記述されていて，コンパイルが可能だったプロジェクトを調査対象として選択した．

表 1 で分かるように最もコードクローンが検出されたプロジェクトは JFreeChart である．ソースコード全体の 56.4%がコードクローンであることが分かる．JFreeChart [*5]はグラフを作成するための Java のライブラリである．作成するグラフごとに機能を実装しているので，非常に多くのコードクローンが見られるものと考えられる．次にコードクローンを多く含むプロジェクトは Apache Xerces [*6]で 34.9%である．このプロジェクトは XML 文書のパースを操作するためのパッケージである．その反面，最もコードクローン行数が少ないプロジェクトは JEdit である．

[*5] http://www.jfree.org/jfreechart/
[*6] http://xerces.apache.org/

表 2 各 OSS から検出されたコードクローンとクローンセット
Table 2 The numbers of code clones and clone sets in each OSS.

プロジェクト名	コードクローン数	クローンセット数	POP
Ant	2,981	515	5.79
Columba	335	135	2.48
JMeter	437	201	2.17
JEdit	124	54	2.29
JFreeChart	9,976	2,309	4.32
JRuby	4,383	1,398	3.13
Xerces	4,889	1,311	3.73

表 3 検出されたクローンペア数とリファクタリング可能なクローンペア数
Table 3 The numbers of detected clone pairs and refactorable clone pairs.

プロジェクト名	クローンペア数	リファクタリング可能なクローンペア数（%）
Ant	8,785	2,068（23.5）
Columba	597	187（31.3）
JMeter	294	99（33.7）
JEdit	100	17（17.0）
JFreeChart	84,551	8,765（10.4）
JRuby	15,546	830（5.3）
Xerces	44,764	1,612（3.6）

表 2 は CCFinderX で検出したコードクローンとクローンセットの個数，クローンセット 1 つあたりに属するコードクローンの個数を示す．コードクローンメトリクス *POP* の値の平均を示す．コードクローンメトリクス *POP* はコードクローンの総和からクローンセットの個数の除算で求められる．コードクローンメトリクス *POP* の値が大きいほど削減可能ソースコード量は大きくなる．したがって，この *POP* によって，リファクタリング時に削除されるコードクローンの数がより多くなることが予想できる．

表 2 で分かるようにコードクローン数が最も多いのは，JFreeChart で，コードクローン行数に反映されている．全体のおおよその傾向として，コードクローンの個数とコードクローン行数は比例することが分かる．また，*POP* の値は，最大が Apaache Ant の 5.15 で，最小が JMeter の 2.17 である．

表 3 は対象 OSS から検出されたクローンペアの個数と JDeodorant で評価したリファクタリング可能なクローンペアの個数を示す．全体のクローンペアのうち，最大で 30%程度のクローンペアがリファクタリング可能なクローンペアである．これはクローン検出ツールが多様なコードクローンを見つけようとして，複数のメソッドにまたがるコード片を持つようなコードクローンや類似した命令文の繰り返しをするようなコードクローンが検出結果に多く含

表 4 JDeodorant に基づいたリファクタリング可能な行数とクローンセット数

Table 4 The numbers of refactorable lines and clone sets based on JDeodorant.

プロジェクト名	JDeodorant 行数	クローンセット数
Ant	11,224	353
Columba	1,394	41
JMeter	1,117	45
JEdit	384	13
JFreeChart	30,495	629
JRuby	7,708	351
Xerces	16,611	374

表 5 提案手法に基づいた削減可能ソースコード量（行数）とクローンセット数（括弧内は，検出されたコードクローン全体に対する削減可能ソースコード量およびクローンセット数の割合，単位は%）

Table 5 The amount of reducible source code and the number of clone sets based on the proposed assessment (the value in parentheses shows the proportion of reducible source code and the number of clone sets to all detected code clones).

プロジェクト名	削減可能ソースコード量	クローンセット数
Ant	3,429（5.7）	264（26.2）
Columba	584（12.7）	35（25.9）
JMeter	385（6.8）	36（17.9）
JEdit	136（7.6）	13（24.1）
JFreeChart	9,700（5.5）	401（17.4）
JRuby	2,161（3.5）	251（18.0）
Xerces	5,533（6.6）	244（18.6）

まれているためと思われる．最もリファクタリング可能なクローンペアが得られたプロジェクトは JFreeChart である．ただし，割合では，Columba[*7]や JMeter が 30%を超えているのが確認できる．最小は Apache Xerces で 3.6%であった．

調査結果

表 4 は各 OSS の JDeodorant が示したリファクタリング可能なクローンセットの行数とそのクローンセット数である．また，**表 5** は JDeodorant の結果を基に提案手法を用いた削減可能ソースコード量とそのクローンセット数を示している．削減可能ソースコード量の括弧内の数値は，元のソースコードから検出された全コードクローン行数に対する比率を表している．

表 6 は表 4 で示した JDeodorant に基づいたリファクタリング可能な行数と，表 5 は JDeodorant の結果を基に提案手法を用いた削減可能ソースコード量を比較した結果を示している．7 つの OSS における平均割合は約 32%である．これは JDeodorant がリファクタリング可能と示したコードクローンは，そのリファクタリングする過程において全行がリファクタリングされるわけではなく，約 32%が削減できることを示している．

[*7] https://sourceforge.net/projects/columba/

表 6 JDeodorant に基づいたリファクタリング可能行数と削減可能ソースコード量（行数）の割合

Table 6 Comparison of the refactorable lines based on JDeodorant and the amount of reducible source code.

プロジェクト名	JDeodorant 行数	削減可能ソースコード量	割合
Ant	11,224	3,429	0.31
Columba	1,394	584	0.42
JMeter	1,117	385	0.34
JEdit	384	136	0.35
JFreeChart	30,495	9,700	0.32
JRuby	7,708	2,161	0.28
Xerces	16,611	5,533	0.33
Average	3,133	9,848	0.32

この理由には次の 2 つが考えられる．1 つ目の理由として，互いにオーバラップしているコードクローンから優先してリファクタリングすべきコードクローンを選択したことがあげられる．このとき，コードクローンのオーバラップの仕方次第では，削減可能な行数は減る可能性がある．2 つ目の理由は，リファクタリングにおける振舞いの変更を防ぐために書かれる共通処理を持つメソッドと呼び出し文の存在である．たとえば，コードクローンメトリクス *POP* の値が n において，式 (3) は $n*c^*_{size}-(c^*_{size}+n+2)=(n-1)*c^*_{size}-2-n$ となる．n の最小値は *POP* の最小値と等しく，2 である．削減可能ソースコード量は共通処理を持つメソッドの呼び出し文がオーバヘッドとなり，削減可能なコードクローン行数は半分以下になると考えられる．

得られた削減可能ソースコード量の考察

表 5 で示すように対象 OSS のソースコードからコードクローンのリファクタリングより削減可能なソースコード量は，平均 6.9%にとどまることが明らかになった．OSS の場合，多くの開発者が保守作業に携わっているため，この結果は予測しうる範囲内といえる．

また，検出されるコードクローン数にも着目した場合，JEdit の個数が変化していないのは，オーバラップしているクローンセットが存在していないためである．JEdit 以外の OSS では，オーバラップしているクローンセットが存在している．また，表 5 中のクローンセット数の列内のカッコ内の数値は，検出されたすべてのクローンセットの個数と比較した削減可能なクローンセットの個数の割合を示している．比率だけを見るなら，Apache Ant が最大の値 26.2%を示している．削減可能なクローンセットの個数を見ると，最大は 401 個の JFreeChart である．しかし，JFreeChart の場合，検出されたコードクローン全体に対する削減可能ソクローンセット数の割合は最小で 17.4%であ

る．これは，多くのコードクローンが検出され，その中にオーバラップしているコードクローンが多く含まれているからである．

4.2 手動のリファクタリングとの比較

動機

JDeodorant 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は，JDeodorant に依存する．しかし，この数値の信頼度は，JDeodorant の正当性が不可欠となる．そこで，この章では，JEdit，JMater，Apache Ant に含まれるリファクタリング可能と評価されたクローンセット 80 個を対象に，手動でリファクタリングを実行して，その削減量を計測した．その手作業による計測値と提案手法を用いて見積った削減可能ソースコード量を比較して，提案手法によって推定した削減可能ソースコード量の正当性を確認する．JEdit，JMater はそれぞれ 14 個と 36 個の削減可能なすべてのクローンセットを対象に実験を行った．また，Apache Ant に対しては，264 個の削減可能なクローンセットのうち，タスク定義を行うコアパッケージである taskdefs パッケージよりさらに小さないくつかのパッケージに含まれる合計 30 個すべてのクローンセットを対象とした．30 個という数字は削減可能なクローンセット数 264 個の約 11.3%に相当する．実験を行う対象やパッケージを絞る理由は，テストコードを実験環境で再現することが困難であるパッケージが含まれていたり，すべての削減可能なクローンセットを対象にするためには多くの時間が必要であったりするためである．

手動によるリファクタリング

JDeodorant でリファクタリング可能と評価された 80 個のクローンセットを手動でリファクタリングを行った．ここで，計測する削減行数は，削減可能ソースコード量に基づいて同様の計算方法を用いて測定した．すなわち，リファクタリング前のクローンセット全体の行数からリファクタリング後の共通処理を持つメソッドの行数と置換された呼び出し文の和を引いた値となる．

比較結果

表 7 は JEdit，JMater，Apache Ant で削減可能と評価されたクローンセット 80 個を対象に，手動でリファクタリングしたコードクローンの削減量と自動的に算出した削減可能ソースコード量を示している．自動的に推定された削減可能ソースコード量は全体で 1,097 行に対して，手動による削減行は 1,170 行であった．すなわち，73 行の差が存在することが分かった．

この原因が誤差が生じた原因に関する考察

73 行の誤差が発生した原因としては，次の 3 つが考えられる．

- 返り値のために return 文が必要な場合がある．
- 共通メソッドや呼び出し文の開始行や終了行を開発者のコーディングスタイルに一致させた場合に，複数行にわたる場合がある．
- instanceof 句でオブジェクトが特定のクラスと等しいか確認した後，そのオブジェクトを判定したクラスとして扱うようにキャストする処理が含まれる．

表 7 手動と自動で推定した削減可能ソースコード量の比較
Table 7 Comparison between manual and automatic estiminations of the amount of reducible source code.

プロジェクト名	クローンセット数	手動	自動
JEdit	14	131	136
JMater	36	397	385
Apache Ant	30	597	654
All	80	1,097	1,170

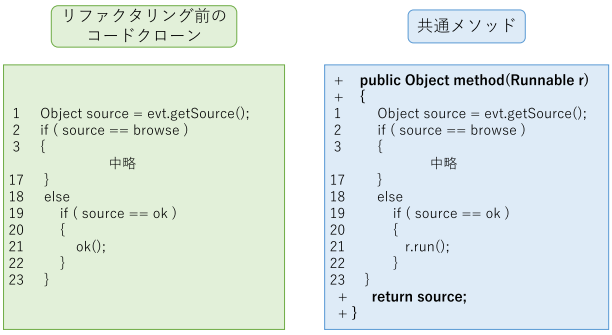

図 4 誤差が発生したコードクローンのリファクタリング例
Fig. 4 Example of refactoring code clones with errors.

1 つ目の原因は共通メソッドに返り値が必要となり，その返り値を返すための return 文を追加するために誤差が生じてしまうからである．評価対象の 80 個のクローンセットのうち，4 個のクローンセットに対して，これが原因で誤差が生じた．この誤差を解決するには，共通処理を持つメソッドとして抽出してきたコード片を分析して，必要な返り値とその型を取得する必要があると考えられる．

2 つ目の原因は開発者のコーディングスタイルを考慮した結果，共通処理を持つメソッドの開始行数が複数行にわたる場合が見られたために誤差が生まれたことである．本調査の場合は，一部の開発者がソースコード中の中括弧のみを改行して 2 行にわたるコーディングをしていたため，誤差が生じた．この場合以外にも，extends 文や implement 文，長すぎる引数などを改行するコーディングスタイルは考えられる．これらを解決するためには，開発者ごとのコーディングスタイルを考慮する必要がある．あるいは，改行そのものはプログラムの動作の本質そのものには影響がないと考えて，コーディングスタイルを考慮しない削減可能ソースコード量を測る手法も考えられる．

図 4 は誤差が発生したコードクローンの一部とそのリファクタリングするにあたって作成した共通処理を持つメソッドの例である．ソースコードの一部は冗長であるために中略している．元のコード片は，Object 型の変数 source

を getSource() メソッドで取得した後，そこに代入されている変数に応じて処理が分岐している．また，このコードクローン箇所の後の処理で，ふたたび変数 source を利用するために source の変数を返す必要がある．そのため，クローンペアがまとめられた共通処理を持つメソッドには，その変数 source を返すために return 文を追加している．また，このソースコードを作成した開発者はメソッドの開始行数を中括弧で改行するコーディングスタイルである．そのため，開発者のコーディングスタイルを考慮して，作成した共通処理を持つメソッドの開始行でも中括弧を改行させている．

3 つ目の原因は，オブジェクトクラスのインスタンスが特定のクラスであるか判別して，そのクラスへキャストする処理が含まれるコードクローンの存在である．このようなソースコードを提案手法どおりにリファクタリングすることは削減困難であると判断した．

4.3 調査結果の考察

4.1 節の調査結果から JDeodorant が計測する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は，各 OSS の平均 6.9%であることが判明した．また，4.2 節の調査結果により手動によるリファクタリング結果は 1,097 行となり，推定された削減可能ソースコード量 1,170 行と比較した．発生した誤差については，それぞれ適切な対処を検討する必要がある．そして，この傾向は調査対象の OSS だけではなく他のオープンソースソフトウェアについても同様に得られると考えている．

JDeodorant は開発者にとって外部的な振舞いが変わることがない簡単なコードクローンのリファクタリングを支援している．しかし，JDeodorant がリファクタリング可能性を評価する際に用いる前提条件に違反するリファクタリングであっても，コードクローンのリファクタリング方法を工夫することで回避できる可能性があるため，4.1 節や 4.2 節の結果で得られた傾向は変わる恐れがある．しかしながら，検出したコードクローンを開発者がほとんど加工をせずに得られる削減可能ソースコード量としては重要な知見を得られたと考える．

5. 妥当性への脅威

5.1 計算式の妥当性

3.4 節の式 (3) は，リファクタリング対象のコードクローンによっては適切な推定値を得ることができない．本研究では，式 (3) はクラスやメソッドを宣言する文を含まないことを前提に削減可能ソースコード量を推定している．これは，コードクローンがクラスやメソッドである場合が稀であるためである．しかし，クラスまたはメソッドをリファクタリングする場合，式 (3) の計算手法では正しい結果は得られない．たとえば，メソッドをリファクタリングする場合，共通の処理を持つメソッドを作成する際にメソッドの開始行と終了行を書く必要はない．

5.2 リファクタリング可能性の前提条件への妥当性

リファクタリング可能性の前提条件は，文献 [16] 中の被験者実験の結果に基づいて定められている．このことは，リファクタリング可能性に強い制約を持たせており，リファクタリングを行う開発者に高度なプログラミング技術を求めないものと思われる．そのため，提案手法におけるリファクタリング可能性の制約条件が，ツールや開発者の技術が実際のソースコードの削減に与える影響は大きく，必ずしも提案手法の削減可能なソースコード量の推定結果が正しくないといえる．そのため，JDeodorant とは異なるリファクタリング可能性の前提条件を持つリファクタリングツールなどを用いて同様の検証を行う必要があると考える．

6. まとめと今後の課題

本稿では，コードクローンのリファクタリングによる削減可能ソースコード量の算出手法を提案した．そして，削減可能ソースコード量を算出するうえで生じる 2 つの課題であるコードクローン間のオーバラップとコードクローンのリファクタリング可能性について説明した．調査実験では 7 つの Java 言語で記述された OSS を対象とした削減可能ソースコード量を測定して，その傾向を考察した．削減可能なソースコード量は検出されたコードクローン行数全体のおよそ 6.9%である．また，検出されたクローンセット数の 20%ほどが削減可能であった．最後に，削減可能ソースコード量の正当性を評価するために，手動のリファクタリングによる削減可能ソースコード量と比較した．その結果，削減可能ソースコード量の算出手法の強い妥当性を得た．

今後の課題として，以下があげられる．

- 削減可能ソースコード量に関して，大規模な商用プログラムへの適用を行う必要がある．
- リファクタリング可能なコードクローンをリファクタリングした場合の，実際のソフトウェア保守へ与える影響を調査する必要がある．
- タイプ 1，タイプ 2 に加えて，タイプ 3 のコードクローンの削減可能ソースコード量の算出手法の提案する．
- 削減可能ソースコード量は多様なコードスタイルに影響を受けるため，ランダムサンプリングなどの手法を用いて評価する．
- 前提条件が異なるリファクタリングツールを用いた評価手法を用いる．
- 提案手法が削減可能ソースコード量を推定する実行時間について調査する．
- 今回対象の OSS は十分にリファクタリングされてい

る可能性がある．リポジトリ内ですでにリファクタリングされている変更に対して評価する．

謝辞 本研究はJSPS科研費JP25220003，JP18H04094，JP16K16034の助成を受けた．

参考文献

[1] Bavota, G., Carluccio, B.D., De Lucia, A., Di Penta, M., Oliveto, R. and Strollo, O.: When Does a Refactoring Induce Bugs? An Empirical Study, 12th IEEE International Working Conference on Source Code Analysis and Manipulation, pp.104–113, SCAM (2012).

[2] Bouktif, S., Giuliano, A., Merlo, E. and Neteler, M.: A novel approach to optimize clone refactoring activity, *Proc. GECCO 2006*, pp.1885–1892 (2006).

[3] Edwards III, B., Wu, Y., Matsushita, M. and Inoue, K.: Estimating Code Size After a Complete Code-Clone Merge，情報処理学会研究報告，Vol.2016-EMB-41, No.3, pp.1–8 (2016).

[4] Harman, M., Phil, M., Jerffeson, de Souza, T. and Yoo, S.: Search Based Software Engineering: Techniques, Taxonomy, Tutorial, *Empirical Software Engineering and Verification*, pp.1–59 (2012).

[5] 肥後芳樹，吉田則裕，楠本真二，井上克郎：産学連携に基づいたコードクローン可視化手法の改良と実装，情報処理学会論文誌，Vol.48, No.2, pp.811–822 (2007).

[6] 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，Vol.J91-D, No.6, pp.1465–1481 (2008).

[7] 肥後芳樹，楠本真二，井上克郎：コードクローン分析ツールGeminiを用いたコードクローン分析手法，電子情報通信学会，Vol.105, No.228, pp.37–42 (2005).

[8] 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.43–56 (2011).

[9] 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.47–54 (2001).

[10] 石津卓也，吉田則裕，崔 恩瀞，井上克郎：メタヒューリスティクスを用いた集約可能コードクローン量の推定，情報処理学会研究報告，Vol.2016-SE-193, No.5, pp.1–8 (2016).

[11] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. ICSE 2007*, pp.96–105 (2007).

[12] 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).

[13] 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).

[14] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: memory comparison-based clone detector, *Proc. ICSE 2011*, pp.301–310 (2011).

[15] Fowle, M.: *Refactoring: Improving the Design of Existing Code* (1999).

[16] Murphy-Hill, E. and Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method, *Proc. 30th International Conference on Software Engineering*, pp.421–430, ACM (2008).

[17] O'Keeffe, M. and Cinneide, M.O.: Search-based refactoring: an empirical study, *Proc. SBSE*, Vol.20, No.5, pp.345–364 (2008).

[18] Tsantalis, N., Mazinanian, D. and Krishnan, G.P.: Assessing the Refactorability of Software Clones, *IEEE Trans. Softw. Eng.*, Vol.41, No.11, pp.1055–1090 (2015).

[19] Tsantalis, N., Mazinanian, D. and Rostami, S.: Clone Refactoring with Lambda Expressions, *Proc. ICSE 2017*, pp.60–70 (2017).

[20] 山中裕樹，崔 恩瀞，吉田則裕，井上克郎：情報検索技術に基づく高速な関数クローン検出，情報処理学会論文誌，Vol.55, No.10, pp.2245–2255 (2014).

石津 卓也

平成27年度大阪大学基礎工学部情報科学科卒業，平成29年度大阪大学大阪大学院情報科学研究科博士前期課程修了．現在，株式会社富士通研究所に勤務．コードクローンのリファクタリング支援に関する研究に従事．

吉田 則裕 （正会員）

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年名古屋大学大学院情報科学研究科附属組込みシステム研究センター准教授．平成29年より同大学大学院情報学研究科附属組込みシステム研究センター准教授（改組による）．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．

崔 恩瀞 （正会員）

平成27年大阪大学大学院情報科学研究科博士後期課程修了．同年同大学大学院国際公共政策研究科助教．平成28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30年より同大学先端科学技術研究科助教（改組による）．博士（情報科学）．コードクローン管理やリファクタリング支援手法に関する研究に従事．

井上 克郎 （正会員）

昭和 59 年大阪大学大学院基礎工学研究科博士後期課程修了（工学博士）．同年大阪大学基礎工学部情報工学科助手．昭和 59 年～61 年，ハワイ大学マノア校コンピュータサイエンス学科助教授．平成 3 年大阪大学基礎工学部助教授．平成 7 年同学部教授．平成 14 年より大阪大学大学院情報科学研究科教授．ソフトウェア工学，特にコードクローンやコード検索等のプログラム分析や再利用技術の研究に従事．本会フェロー．