情報処理学会論文誌 Vol.56 No.6 1471–1480 (June 2015)

メソッド呼び出しの差異に基づくコードクローンの分類手法

石尾 隆[1,a)] 伊達 浩典[1,b)] 井上 克郎[1,c)]

受付日 2014年9月19日, 採録日 2015年3月4日

概要：本論文は，Java プログラムに出現するコードクローンの分類として，コードクローンが出現するメソッドに共通の振舞いを提供するかどうかという基準を提案する．この基準によって，メソッド呼び出しに差異があるコードクローンを，多少の差異はあっても共通の振舞いを提供するコードクローンと，異なる振舞いを提供するコードクローンとに分類することを可能とする．分類を自動的に行うために，メソッドのサマリ抽出技術として提案されている既存の経験的な基準を応用して，コードクローンに出現するメソッド呼び出しがメソッドの振舞いに与える影響を調べる手法を定義した．

キーワード：コードクローン，Java，静的解析，ソースコード差分抽出

Classification of Code Clones based on Method Call Differences

Takashi Ishio[1,a)] Hironori Date[1,b)] Katsuro Inoue[1,c)]

Received: September 19, 2014, Accepted: March 4, 2015

Abstract: This paper proposes a classification technique for code clone in Java program using a new criterion: whether a clone set provides the same behavior to methods or not. The criterion distinguishes code clones that provide the same behavior from code clones that provide different behavior. To automate the classification process, this paper defines a technique to analyze the relationship between method calls in code clones and behavior of methods including the code clones. The technique employs heuristics originally used for automatic summarization of a Java method.

Keywords: code clone, Java, static analysis, source code differencing

1. はじめに

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．1つの変更を行うために複数のコードクローンを一貫して修正する必要が生じる場合があるため，コードクローンは「不吉なにおい」の1つであると認識されている [1]．コードクローンの検出を目的としたツールは多数存在しており，企業におけるソフトウェア製品の分析にも活用されている [2], [3]．

多くのコードクローン検出ツールは，ソースコードの比較において多少の差異を許容することで有用なコードクローンを検出する [4] が，許容する差異は，検出方法に基づいて定義されたものであり，開発者の意図に対応するわけではない．たとえば，CCFinder [5] は識別子などの名前の違いを無視すると同一のトークンの系列となるようなコード片をコードクローンとして検出するツールであり，コピーされたソースコードに名前の変更があってもコードクローンとして検出することができる．しかし，名前だけが異なるという字句的な特徴を持つコードクローンが，開発者にとってどのような意味があるかは明らかではなく，その中から分析する価値のあるコードクローンを選定することは困難である．開発者が分析対象のコードクローンを効果的に選択するためには，コードクローンの意味的な特徴に基づく分類が必要である．

本研究では，コードクローンの意味に基づく分類方法として，コードクローンに含まれるメソッド呼び出しに着目

1 大阪大学
Osaka University, Suita, Osaka 565–0871, Japan
a) ishio@ist.osaka-u.ac.jp
b) h-date@ist.osaka-u.ac.jp
c) inoue@ist.osaka-u.ac.jp

© 2015 Information Processing Society of Japan

し，メソッド呼び出しの差異がコードクローンを含むメソッドの振舞いに影響を与えるかどうかという基準でのコードクローンの分類を提案する．これにより，多少の差異はあっても共通の振舞いを提供するコードクローンと，異なる振舞いを提供するコードクローンとを区別する．前者のクローンはアプリケーション固有の処理やビジネスロジックに関連し，リファクタリングや変更時の影響波及の分析に有益な情報であることが見込まれる．また，後者は複数の機能の実装に共通する実装上の特徴を理解するために有益であると考えられる．分類を自動的に行う方法として，Java のメソッドのサマリを自動抽出する Sridhara らの手法 [6] で提案された経験的な基準を応用する．この基準を用いると，メソッドの振舞いを説明するサマリに含めるべき重要なメソッド呼び出しの列を抽出することができるため，コードクローンの範囲に含まれるメソッド呼び出しが複数のメソッドに同一のサマリの内容を提供するなら，それらのコードクローンは共通の振舞いを提供すると判定する．

提案手法をコードクローン検出ツール CCFinder のコードクローンを対象として実装し，Qualitas Corpus 20130901r [7] に収録されたソフトウェアを対象とした分析を実施した．その結果，(1) メソッドのサマリに影響するメソッド呼び出しと，そうでない呼び出しに差異がある確率に違いはなく，偏った分類にはならないこと，(2) 提案手法で得られる分類が，同一の振舞いを異なる変数や定数などを参照して実行するためのコードクローンと，同一の変数や定数を用いて異なる振舞いを実行するためのコードクローンとを区別していること，(3) リファクタリング対象となるコードクローンを選定するために使用されたメトリクス [8] による単純な並べ替えなどで特定の分類に属するコードクローンを取り出すことは期待できないことを確認した．

本論文の構成は次のとおりである．2 章では研究の背景であるコードクローン検出技術とメソッドのサマリ抽出技術について解説する．3 章では本研究の提案であるコードクローンの自動分類手法について記述する．4 章では手法を実際のコードクローンに適用した結果を示す．5 章で妥当性への脅威について整理し，6 章でまとめを述べる．

2. 背景

2.1 コードクローン検出

開発者がソースコードをよく複製して再利用することは，Kim らによって報告されている [9]．Zhang らによるインタビュー調査 [10] では，開発者がソースコードを複製する理由として，再利用による作業時間の節約に加えて，機能ごとに利用するコードを複製しておくことで各機能を独立して変更できる状態に保ちたいという要求があげられている．

コードクローン検出は，複製されたソースコードを自動的に検出する技術である．複数の人間が同一の機能を実装したとしても同一のコードになることは珍しい [11] ことから，ソースコード断片を何らかの基準で比較して類似したものを選び出すと，多くは意図的にコピーされたものであると期待できる．一方で，ソースコードをコピーしたとしても，それが元のコードと同一であるとは限らない．たとえばコピー先で使用している変数名に合わせて，コピーしたソースコード中の変数名を変更することがある [4].

コードクローンにおけるソースコードの差異の度合いについては，以下の分類が知られている [12], [13].

タイプ 1 空白やタブの有無などのコーディングスタイルを除き，内容が完全に一致するコードクローン．

タイプ 2 変数名やメソッド名などのユーザ定義名，また型名などの一部の予約語のみが異なるコードクローン．

タイプ 3 タイプ 2 の変更に加え，任意の文の挿入や削除が行われたコードクローン．

タイプ 4 類似した処理を実行するが，構文上の実装が異なるコードクローン．

検出できるコードクローンのタイプはツールによって異なり，Roy ら [13] はツールが複製されたコード片への変更をどれだけ許容するか，という観点からツールの機能を特徴付けている．

本研究で使用する CCFinder [5] は，識別子などの名前を無視すると同一となるトークン列を探索し，一定以上の長さのものをタイプ 2 のコードクローンとして検出するツールである．CCFinder は多くの研究や企業で使用されており [14]，後継ソフトウェアに当たる AIST CCFinderX [15] も広く利用可能であることから，CCFinder が検出するコードクローンを調査対象とした．

CCFinder が検出するコードクローンを効果的に分析するために，コードクローンの長さ（LEN）や同一と見なされるコードクローンの数（POP），より小さなコード片の繰り返しでないことを示す指標（RNR）などのメトリクスが提案されており，単純な代入文の繰り返しなどによって偶然発生するコードクローンは，RNR の値を用いたフィルタリングによってその多くを結果から取り除くことができる [16]．取り出されたコードクローンからリファクタリング対象の候補を選定するためにこれらのメトリクスを組み合わせた事例も報告されている [8] が，リファクタリングの判断自体は開発者が個別に確認しており，メトリクスに基づく自動的な分類がなされているわけではない．

2.2 コードクローン内部のコードの比較

検出されたコードクローンのコード内容を比較することで，そのコードクローンの特徴を理解しようという研究はいくつか行われている．Hayase ら [17] は，コードクローンに出現する識別子の対応関係を分析し，1 つのコード片

© 2015 Information Processing Society of Japan

でのある変数 x の出現が，そのコードとクローンであるような他のコード片で x と y の 2 つに対応している場合，変数の名前の変更し忘れの可能性があるとして警告を出すツールを提案している．Xing ら [18] は，開発者がコードクローンの意味的な差異を分析するツールとして，一対のコードクローンの制御フローやデータフローの違いを可視化するビューアを提案した．開発者は可視化された結果から差分を理解する必要があり，本研究のような機械的な分類とは目的が異なる．Lin ら [19] は，3 つ以上のコード片が互いにコードクローンであるような場合に，それらの共通部分と差分とを効果的に閲覧できるツールを提案している．このツールも，差分から機能の差異を理解する作業そのものは開発者に委ねている．

Choi ら [20] は，検出されたコードクローンの字句的な類似度の違いと，コードクローンに対して適用されるリファクタリングに関係があることを報告している．字句的な類似度だけでは使用されるリファクタリングを特定することはできないとも報告しており，字句的な特徴がコードクローンの分類には不十分であることを示している．

工藤ら [21] は，コードクローンにおける変数名やメソッド名の差異の有無を調査し，1 つのコード片が他の場所にコピーされて変更されたとしても，識別子に 1 対 1 の対応関係があることを報告している．そのため，コードクローンとして抽出されたコード片に出現するメソッド名などに差異があっても，何らかの関係がある可能性が高いが，具体的にどのような差異があるかまでは調査に含まれていない．また，工藤ら [22] は，コードクローンに含まれるメソッド呼び出しの比較において，Sridhara らの手法 [6] を元にして選択されるメソッド呼び出しのほうがコードクローン間で差異が少ないと報告している．本研究では，工藤ら [22] のサマリ抽出のアイディアをコードクローンの分類手法へと拡張し，分類結果の妥当性についての議論も新規に行っている．

2.3 メソッドのサマリの抽出

Sridhara ら [6] は，「メソッドの振舞いを説明するコメントの内容としてふさわしい」プログラム文を抽出し，それをコメントに変換する技術を提案した．プログラム文には様々なものがあるが，Sridhara らはサマリに対する重要性として，以下の経験的な選択基準をあげている．

Ending：多くのメソッドでは，何らかの準備を行ってから実際の処理を実行するため，メソッドの末尾に出現するプログラム文は重要である．

Void-Return：戻り値が使われていないか，戻り値の型が void であるメソッド呼び出し文は，システムに対する副作用を持つため重要である．

Same-Action：メソッドと同じ意味の操作をするプログラム文，たとえば compile というメソッド中で compileRegex というような同一の動詞を持つメソッド呼び出しは，実装の詳細を表現しているため重要である．

```
1: void returnPressed() {
2:   Shell s = getShell();
3:   String input = s.getEnteredText();
4:   history.addElement(input);           // Summary
5:   String result = evaluate(input);    // Summary
6:   s.append(result);                   // Summary
7: }
```

図 1 サマリ抽出のコード例 [6]

Fig. 1 An example code for summary extraction [6].

Controlling：上記の基準で選ばれたプログラム文の実行を直接制御するプログラム文は重要である．

Data-Facilitating：上記の基準で選ばれたプログラム文に直接データを供給するプログラム文は重要である．

Filtering：`log` や `debug` などの特定の名前で始まるメソッド呼び出しは，その動作がメソッドの目的に合致しない限り，重要ではないと考える．また，例外ハンドラに出現する文や，`x==null` の形の条件文で実行される例外処理と思われる文も，重要ではないと考える．

それぞれの経験的な基準の先頭に表記した名称は，それぞれ実装の詳細について述べる際のルールの識別に用いる．

本来の Sridhara らの手法は，これらの条件で選ばれたプログラム文に出現する単語を収集し，自然言語の文章へと変換する．プログラム文には一般的な代入文や四則演算も含まれるが，それらはコメントに反映されることがなく，メソッド呼び出しを主な情報源として用いていることから，本研究ではメソッド呼び出しの分類手法として適用する．

本研究では，この手法で抽出されるメソッド呼び出しの列を，単に「メソッドのサマリ」と呼ぶ．2 つのメソッドのサマリが同一であればそれらの振舞いが同一であると期待できるため，コードクローンに由来するサマリが同一であれば，コードクローンがそれらのメソッドに同一の振舞いを提供するためのものであると判断できると考えた．

メソッドのサマリの抽出例を，**図 1** に示すメソッドを用いて説明する．まず，このメソッドの 6 行目で呼び出される `append` メソッドが，メソッドの末尾で実行されているため，メソッドのサマリに含まれる．この呼び出しの引数である `result` は 5 行目で計算されているため，そこで呼び出される `evaluate` もメソッドのサマリに含まれる．4 行目の `addElement` は戻り値が利用されておらず，副作用に意味があると考えられるため，メソッドのサマリに含まれる．3 行目の `getEnteredText` はそれにデータを供給する役割を持っているが，`get` から始まる名前のメソッドなので，メソッドのサマリには含まれない．2 行目の `getShell` は，サマリに含めない 3 行目へのデータ供給であり，同様にメソッドのサマリには含まれない．結果として，このメソッドのサマリに含まれるメソッド呼び出しは 4 行目

© 2015 Information Processing Society of Japan

の addElement，5 行目の evaluate，6 行目の append となる．Sridhara らのサマリ抽出手法で最終的に出力されるサマリは，これらのメソッド呼び出しとその引数を文章の雛形に当てはめてつなげたもの，すなわち「Add input to history. Evaluate input, and get result. Append result to Shell.」という文となる．開発者がこのメソッド全体をコピーして新しいメソッドを定義したとき，これら 3 つのメソッド呼び出しを変更せず同一のメソッド呼び出しがサマリに含まれる場合，コードクローンが共通の振舞いを提供していると見なす．

3. コードクローンの自動分類

本研究では，コードクローンが複数のメソッドに共通の振舞いを提供しているかどうかを判定する．対象となるコードクローンは，CCFinder が検出する，互いに類似または一致するコード片の集合（クローンセット）のリストである．1 つのクローンセット $C = \{c_1, \cdots, c_n\}$ は $\forall c_i, c_j \in C : c_i \simeq c_j$ を満たすような n 個のコード片の集合である．ここで $c_i \simeq c_j$ は，コード片に出現するコメントと空白を取り除き，すべての名前（型名の予約語を含む）やリテラルを単一の記号（たとえば $p という文字列）で正規化すると完全一致することを表す．論文中では，各コード片 c_i のことをコードクローンと呼ぶ．たとえば，図 1 に示したコードの 2 行目から 7 行目は，**図 2** に示したコードの 3 行目から 8 行目までと，呼び出しているメソッドの名前を除くと一致するため，これら 2 つのコード片は同一のクローンセットに属するコードクローンである．クローンセットのうち，各コードクローンの出現位置がそれぞれ異なる単一メソッドに収まっているものだけを本研究の対象とし，たとえばファイル全体が複製されている場合や，単一メソッド内部の 2 カ所が同一の記述となっている場合などは扱わない．

コードクローンの振舞いに関する判定の基準は，コードクローンに含まれるメソッド呼び出しについての以下の 2 つの観点である．

- P_1：メソッドのサマリに含まれるコードクローン内のメソッド呼び出しが，クローンセットに含まれる他のコードクローンすべてと同一であるか．
- P_2：メソッドのサマリに含まれないコードクローン内のメソッド呼び出しが，クローンセットに含まれる他のコードクローンすべてと同一であるか．

```
1: void returnPressed() {
2:   Logger.debug("returnPressed");
3:   Shell s = getShell();              // Clone
4:   String input = s.getInputText();  // Clone
5:   history.addElement(input);         // Clone, Summary
6:   String result = evaluate(input);  // Clone, Summary
7:   s.append(result);                  // Clone, Summary
8: }                                    // Clone
```

図 2 図 1 のメソッドとサマリが同一であるクローンの例
Fig. 2 An example clone that has the same summary as Fig. 1.

コードの差異を分析するときには，すべての出現に共通なコードとそれ以外の可変部分を分けて考えることが重要となる [23] ため，クローンセット全体で同一であることを判定基準としている．いずれの基準も，メソッド呼び出しの同一性として定義されるので，クローンセット C について，分類結果を S（Same），D（Different），N（Not Available）の 3 種類で表現する．コードクローンで互いに対応する位置に出現するメソッド呼び出しを比較したとき，複数のメソッドのサマリに含まれているものがすべて同名のメソッド呼び出しであれば（同一のサマリを提供するなら）$P_1(C) = S$，1 つでも異なる名前のメソッド呼び出しであれば（その結果サマリが異なるなら）$P_1(C) = D$，そもそもクローンセット C の中のどの位置のメソッド呼び出しもサマリに含まれていなければ $P_1(C) = N$ とする．同様に，クローンセット C に含まれるメソッド呼び出しのうち，メソッドのサマリに含まれない範囲がすべて同名のメソッド呼び出しであれば $P_2(C) = S$，1 つでも異なる名前のメソッド呼び出しであれば $P_2(C) = D$，サマリに含まれない範囲にメソッド呼び出しが存在しなければ $P_2(C) = N$ とする．この比較では，呼び出すメソッドの名前だけを使用し，所属するクラス名や引数の型は含まない．

クローンセットの分類例を，図 1 と図 2 のコードを用いて説明する．図 2 のメソッドは，図 1 のコードの 1 行目と 2 行目の間に debug メソッド呼び出しを追加し，さらに 4 行目となったメソッドの呼び出し先を変更している（図 2 中の下線部）．CCFinder は，図 1 の 2 行目から 7 行目と，図 2 の 3 行目から 8 行目をコードクローンとして検出する．この 2 つのメソッドからは 2.3 節で述べた基準により同一のサマリが得られるため，$P_1(C) = S$ である．また，サマリに含まれていないコードクローン内のメソッド呼び出しは図 1 では **getShell**，**getEnteredText** であり，図 2 では **getShell**，**getInputText** と 1 つ異なることから，$P_2(C) = D$ である．つまり，このクローンセットは，共通の振舞いを 2 つのメソッドに提供している一方，それ以外の点で処理内容に違いがあると分類される．

本研究で用いるこれらの基準で，コードクローンは 3×3 の 9 通りに分類されるが，コードクローンに含まれるメソッド呼び出しの列を比較するだけでもメソッド呼び出しが同一である（$P_1(C) = S$，$P_2(C) = S$）グループとメソッド呼び出しを含まない（$P_1(C) = N$，$P_2(C) = N$）グループは分類することが可能である．本研究の分類の新規性は，コードクローンに含まれるメソッド呼び出しが 1 つ以上異なる場合に，それが $P_1(C) = D$ なのか，それとも $P_2(C) = D$ なのかを区別することにある．

© 2015 Information Processing Society of Japan

表 1　コードクローンに含まれるメソッド呼び出しの系列の例
Table 1　Lists of method calls in code clones.

l	$M_1[l]$	サマリ	$M_2[l]$	サマリ
1	getShell		getShell	
2	getEnteredText		getInputText	
3	addElement	Yes	addElement	Yes
4	evaluate	Yes	evaluate	Yes
5	append	Yes	append	Yes

3.1　分類方法

クローンセット $C = \{c_1, \cdots, c_n\}$ に対して，各コードクローンの内部に出現するメソッド呼び出しの系列の集合 $M_1, \cdots, M_n$ を求める．CCFinder で抽出するコードクローンはトークンの並びが名前を除いて一致するため，Java の構文解析で同じ意味を持つ．つまり，呼び出し列 $M_1, \cdots, M_n$ の長さはいずれも同じであり，それを m とすると，個々の呼び出しを $M_i[l](1 \leq i \leq n, 1 \leq l \leq m)$ と表現できる．

続いて，コードクローンの出現位置である各メソッドに対してメソッドのサマリを計算し，コードクローン内部に出現しているそれぞれのメソッド呼び出しを，サマリに属するか，属さないかの 2 値に分類する．例として，図 1 の 2 行目から 7 行目と図 2 の 3 行目から 8 行目からなるクローンセットから得られるメソッド呼び出しの系列の集合と，サマリの計算結果を**表 1** に示す．「サマリ」列に Yes と書かれたメソッド呼び出しは，そのコードクローンを含むメソッドのサマリに含まれていることを示している．

コードクローン c_i の中の l 番目に出現するメソッド呼び出しがサマリに属することを $isSummary(i, l)$ という述語で表現し，すべてのメソッドのサマリに影響するようなコードクローン中のメソッド呼び出し位置の集合 L を以下のように定義する．

$$L = \{l \in [1, m] \mid \forall i \in [1, n] : isSummary(i, l)\}$$

クローンセットに含まれるすべてのコードクローンの l 番目のメソッド呼び出しが，各出現位置のメソッドでサマリに含まれているとき，l が L の要素となる．表 1 の場合，$L = \{3, 4, 5\}$ である．

基準 P_1 として，クローンセット C がメソッドのサマリに同一のメソッド呼び出しを提供しているかどうかを，以下のように求める．

$$P_1(C) = \begin{cases} N & \text{if} \quad L = \phi \\ S & \text{if} \quad \forall i, j \in [1, n], \forall l \in L : M_i[l] = M_j[l] \\ D & \text{otherwise} \end{cases}$$

同様に基準 P_2 として，クローンセット C がメソッドのサマリ以外の部分に同一のメソッド呼び出しを提供しているかどうかを，以下のように求める．

$$P_2(C) = \begin{cases} N & \text{if} \quad \overline{L} = \phi \\ S & \text{if} \quad \forall i, j \in [1, n], \forall l \in \overline{L} : M_i[l] = M_j[l] \\ D & \text{otherwise} \end{cases}$$

ただし $\overline{L} = \{l \in [1, m] | l \notin L\}$ である．

表 1 のデータについて上記基準を求めると，$M_1[3] = M_2[3] =$addElement，$M_1[4] = M_2[4] =$evaluate，$M_1[5] = M_2[5] =$append であるから，$P_1(C) = S$ となる．また，$\overline{L} = \{1, 2\}$ であり，$M_1[2] \neq M_2[2]$ となることから $P_2(C) = D$ となる．

3.2　メソッドに対するサマリ抽出の実装

本研究では，サマリ抽出を Java バイトコードに対して実装した．Java のソースコードからバイトコードへのコンパイルの時点では大きな最適化が行われることはなく，分割コンパイルやデバッグ支援のためにソースコード上での行番号や変数名に関する多くの情報も保っていることから，本研究ではバイトコードを解析の容易な中間表現と見なした．経験的な基準については，文献 [6] には厳密なルールが記述されていなかったため，可能な限り同一であることを目指しているが，実装としては独自のものとなっている．

サマリ抽出は，1 つの Java のメソッドを入力とする手続き単位の解析である．メソッド内部に出現するメソッド呼び出し命令の系列 $M = \langle m_1, \cdots, m_{|M|} \rangle$ から，部分系列であるようなサマリ $S(M)$ を抽出する．各メソッド呼び出し命令は，Java バイトコードにおける INVOKE 命令群である．本節の言及に登場する「文」は，抽象構文木上で定義される文ではなく，仮想マシンのオペランドスタックに値を積み上げてそれを使用する一連の命令であり，オペランドスタックが空になる時点を文の境界と見なす．これは，デバッガにおける 1 ステップの実行にほぼ対応しており，たとえば for 文における初期化，条件評価，インクリメントの各節はそれぞれ個別の文と見なす．

本研究では，文献 [6] と同様に，まず 3 つの経験的な基準 Ending（*END*），Void-Return（*VR*），Same-Action（*SA*）によってメソッド呼び出し集合 M から部分集合を抽出する．これらの基準で選択されたメソッド呼び出し集合を H とすると，H に含まれるメソッド呼び出しの実行を制御する，あるいは呼び出しに引数を提供するメソッド呼び出し集合を H に加えたものをサマリとして求める．具体的な式は以下のとおりである．

$$H(M) = filter(END(M) \cup VR(M) \cup SA(M))$$
$$S(M) = H(M) \cup CTRL(H(M)) \cup DF(H(M))$$

この式に登場する経験的な基準の定義を以下に示す．

- $END(M)$ はメソッド内部で RETURN 命令の直前に実行される文か，あるいは ARETURN，LRETURN，IRETURN，FRETURN，DRETURN のいずれかの命令を含む文の中に

© 2015 Information Processing Society of Japan

出現するメソッド呼び出しを取り出す．

- $VR(M)$ は戻り値を持たない（戻り値の型が void である）か，戻り値を持つが使用されずに破棄されるメソッド呼び出しを取り出す．ただし，set メソッドについては，オブジェクトの初期化など中間的な計算でもよく使用されるため除外する．
- $SA(M)$ は解析対象メソッドの名前と呼び出し先のメソッドの名前をそれぞれ大文字およびアンダースコアで単語に分解したとき，先頭の単語が一致するものだけを取り出す．ただし，get と set から始まるメソッドと，<init>（コンストラクタ）に関しては，使われる頻度が高く偶然の一致が多く見られるため，この計算からは除外する．
- *filter* は，経験的な基準 Filtering に対応し，上記 3 つの基準で選択されたメソッド呼び出しの中から，以下の条件のいずれかに合致する呼び出しを除外する．
  (1) メソッド名に log，trace，error，debug，exception，close のいずれかが含まれる．
  (2) メソッド呼び出しが catch ブロック内部にある．
  (3) メソッド呼び出しが，何らかのオブジェクト参照が null であると判定されたときにのみ（ifnull 命令か ifnonnull 命令による条件分岐によってのみ）到達する制御パスに出現している．
- $CTRL(H(M))$ は Controlling に対応し，$H(M)$ に含まれるメソッド呼び出しに対して直接の制御依存辺を持つ条件分岐の判定文に出現するメソッド呼び出しの集合である．制御依存辺は，メソッドの実行開始と終了をそれぞれ単一の入口と出口とする制御フローグラフ上で，通常のプログラム依存グラフ [24] と同じ方法で計算する．ただし，メソッド呼び出しによって生じる例外は考慮しない．また，throw 文はメソッドの実行終了と見なす．
- $DF(H(M))$ は Data-Facilitating に対応し，$H(M)$ に含まれるメソッド呼び出しの引数に使われる値を計算するメソッド呼び出しと，引数として参照されるローカル変数の値を定義する文の計算式に含まれるメソッド呼び出しの集合である．文間で直接の依存関係があるものだけを扱い，推移的な依存関係は取り扱わない．また，get，set から始まるメソッドとコンストラクタは結果に含めない．

結果として得られたメソッドのサマリとコードクローンの範囲とを対応付けることで，コードクローン c_i に含まれるメソッドのサマリの集合を得る．

3.3 メソッドのサマリとコードクローンの対応付けの実装

サマリの計算は Java バイトコード上で実行されるため，バイトコードに格納されたデバッグ情報，すなわち各バイトコードの元になったソースファイル名と行番号を用いて，CCFinder が検出したコードクローンとの対応付けを行う．具体的には，コードクローンの範囲に含まれるメソッド名に対して，対応する行番号のバイトコードに同名のメソッドの呼び出し命令があれば，そのメソッド呼び出し命令がソースコードに対応すると考える．この方式により，ソースコードに出現しないがバイトコード上には出現するようなメソッド呼び出し，たとえば拡張 for 文に対して生成される Iterator に対する操作などをコードクローン内のサマリからは除外している．

4. コードクローンの分類結果の調査

提案手法で得られるコードクローンの分類について，以下の観点から妥当性の評価を行う．

- 共通の振舞いを提供するクローンとそうでないクローンの割合に差がないか．差がない（一方が大多数を占めない）ほうが，調査対象などの絞り込みの用途では有用性が高い．
- 2 つの分類から取り出したクローンが実際に振舞いの共通性に関係しているか．
- 既存のコードクローン分析用のメトリクスでは同様の分類を得ることが難しいか．

評価対象として，Qualitas Corpus 20130901r [7] に収録された各プログラムについて提案手法を適用する．同コーパスは，複数のプログラムのコンパイル元のソースコードとコンパイル後のバイトコードとを組で収録しており，以下の手順で手法の適用を行う．

(1) 対象プログラムのソースコードから拡張子が.java であるファイルをすべて列挙し，それらに対して CCFinder を実行する．CCFinder の検出オプションは，デフォルト設定の最小トークン数 30 を用いる．

(2) CCFinder が検出したすべてのコードクローンから，本研究の対象とするコードクローンのみを抽出する．
  (a) RNR 値が 0.5 より大きいコードクローンのみを抽出し，偶然の一致によって生じた可能性の高いコードクローンを取り除く [16]．
  (b) 複数のメソッド定義をまたぐコードクローンを取り除く．
  (c) 同一クローンセットのコードクローンが 1 つのメソッドに 2 つ以上含まれる場合は，ソースコード上で最初に開始するものだけを使用し，1 つのクローンセットに含まれるコードクローンがそれぞれ異なるメソッドに出現する状態とする．ソースコード上で早く出現するコードクローンのほうが，メソッドの末尾の文を含まず，サマリに含まれないコードとしての出現である可能性が高い．コードクローンの分類において，コードクローンによって対応付けられた位置にあるメソッド呼び出しがすべてサマリに含まれる場合だけを「サマ

© 2015 Information Processing Society of Japan

表 2 コードクローンの分類結果（86 プログラムの合計）
Table 2 Classification result of code clones (Total of 86 projects).

	$P_2(C)=D$	$P_2(C)=S$	$P_2(C)=N$
$P_1(C)=D$	1,097	2,840	3,087
$P_1(C)=S$	3,759	15,636	4,881
$P_1(C)=N$	6,818	15,214	2,382

リに影響する」と判定しているため，サマリに含まれない可能性が高いクローンのほうを解析対象として採用している．

(3) 対象ソフトウェアのバイナリファイルのうち，デバッグ情報からコードクローンのソースファイルに対応付けることができたものを使って各コードクローンに対応するメソッドサマリを計算する．デバッグ情報を持ったバイナリが収録されておらず，サマリが計算できなかったコードクローンは，解析から除外した．

(4) メソッドサマリの情報を用いて，各クローンセット C について，分類基準 $P_1(C)$，$P_2(C)$ の値を求めた．

解析したコードクローンは 86 個のプログラムから抽出された計 55,714 個のクローンセットである．P_1，P_2 の値によって 9 通りに分類した結果を**表 2** に示す．

4.1 分類の偏りの調査

メソッド呼び出しに何らかの差異を含んでいる（$P_1(C)=D$ または $P_2(C)=D$ である）クローンセットは，全体の 31.6%に当たる．メソッド呼び出しの差異は，サマリまたはサマリではないメソッド呼び出しのどちらか一方だけが異なる場合が多く，両方が異なるコードは全体の 2.0%に当たる 1,097 クローンセットだけに限られる．このことから，サマリに該当する部分か，サマリに該当しない部分か，いずれか一方の内容を再利用するためにコードクローンが作成されている可能性が考えられる．

サマリを含む（$P_1(C)=S$ または D である）クローンセットのうちサマリ部分が異なる（$P_1(C)=D$ である）クローンセットの割合と，サマリ以外のメソッド呼び出しを含む（$P_2(C)=S$ または D である）クローンセットのうちそれらの呼び出しが異なる（$P_2(C)=D$ である）クローンセットの割合は，それぞれ 22.4%，25.7%である．これらの割合を解析対象プログラムごとに求めた場合の分布を**図 3** に示す．この分布に対して Wilcoxon の順位和検定を行ったところ p=0.2633 となり，統計的に差があるとはいえなかった．このことから，サマリに含まれるメソッド呼び出しと，サマリに含まれないメソッド呼び出しで，どちらか一方に差異が発生しやすいということはない．つまり，観点 P_1 と P_2 によって，$P_1(C)=D$ と $P_2(C)=D$ の 2 つに偏りなくコードクローンを分類すると期待できる．

上記の結果は，サマリに含まれるメソッド呼び出しのほうが差異が少ないという報告 [22] と結果が異なる．これは，文献 [22] では $P_1(C)=S$ と $P_1(C)=N$ の場合が区別されておらず「差異なし」とまとめられていること，本研究で解析対象プログラム数を増やしたことの影響であると考えられる．

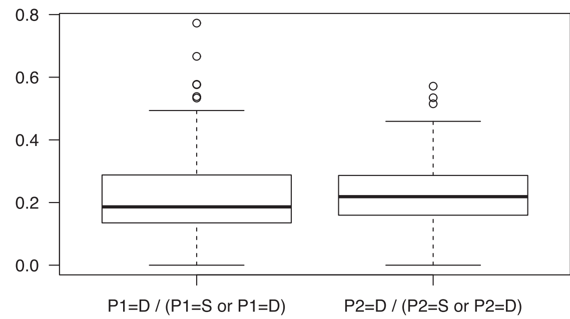


図 3 $P_1(C)=D$ および $P_2(C)=D$ であるクローンセットの割合（プログラム別）
Fig. 3 Ratio of code clones whose $P_1(C)=D$ or $P_2(C)=D$ (for each program).

4.2 分類ごとのコードクローンの内容の比較

本研究で導入した分類では，異なるメソッド呼び出しが使用されたクローンセットについて，そのメソッド呼び出しがサマリに当たるか（$P_1(C)=D$ であるか），サマリ以外の部分に当たるか（$P_2(C)=D$ であるか）を調べることができる．この 2 つの分類が，開発者にとって意味的に異なる特徴を持つクローンセットであるかどうかを調査するために，$P_1(C)=D$，$P_2(C)=S$ であるクローンセットと $P_1(C)=S$，$P_2(C)=D$ であるクローンセットから，サンプリングによる分析を行った．具体的には，データセットに収録されているプログラム群からランダムに選択した Ant，Azureus，JHotDraw，Sandmark，Weka に対して，該当する分類に含まれるクローンセットを 5 つずつ，合計で 50 個選び出し，以下の基準で分類を行った．

クローンセットで使用されているデータの同一性． クローンセットで同型・同名の変数，同一の定数，同名の get メソッドを使用している場合を「同一データ」とし，別名だが同一型であれば「同一型」，それ以外を「異なるデータ」として分類した．

クローンセットが実行する処理の同一性． 入力データの取得と思われる処理を除き，クローンセットで同名のメソッドを使っている場合を「同一処理」，異なるメソッドを使用していても動詞が共通するなど同種の機能と思われる場合（たとえば `reset` と `resetOptions` など）を「類似処理」，それ以外のメソッド名を使用している場合を「異なる処理」と分類した．

データに関する分類結果を**表 3** に，処理に関する分類結果を**表 4** に示す．表 3 から，$P_2(C)=S$ であるクローンのほうが同一データを使用しているものが多いことが分か

© 2015 Information Processing Society of Japan

表 3 コードクローンが使用するデータの比較結果
Table 3 Comparison of data used by code clones.

クローンの分類	同一データ	同一型	異なるデータ	合計
$P_1 = S, P_2 = D$	0	15	10	25
$P_1 = D, P_2 = S$	14	7	4	25

表 4 コードクローンの処理内容の比較結果
Table 4 Comparison of functionality by code clones.

クローンの分類	同一処理	類似処理	異なる処理	合計
$P_1 = S, P_2 = D$	12	13	0	25
$P_1 = D, P_2 = S$	0	14	11	25

```
public long size() {
 ZipEntry entry;
 try {
  if (this.isInArchive()) {
   entry=this.archiveEntry();
   return entry.getSize();
  } else {
   return this.getFile().
          length();
  }
 } catch(Exception ex) {
  ...
  return 0L ;
 }
} // size()
```

```
public long lastModified() {
 ZipEntry entry;
 try {
  if (this.isInArchive()) {
   entry=this.archiveEntry();
   return entry.getTime();
  } else {
   return this.getFile().
          lastModified();
  }
 } catch(Exception ex) {
  ...
  return 0L ;
 }
} // lastModified()
```

図 4 Ant に含まれる $P_1 = D$，$P_2 = S$ であるクローンの例
Fig. 4 An example $P_1 = D, P_2 = S$ clone set in Ant.

る．これは，サマリに含まれないメソッド呼び出しとして，メソッドの先頭部分でデータを読みだす get メソッドが多いことが影響している．一方で，表 4 では，$P_1(C) = S$ であるグループのほうが，共通の処理を行っているクローンセットであることが示されている．このグループのクローンの多くは，get メソッドなどで異なるデータを参照するが同一の処理を実行しており，サマリとして該当コード片の主要な機能が正しく分類に反映されたと考えられる．$P_2(C) = S$ のクローンセットでは，主にデータ構造へのアクセスに関するメソッドが共通しており，最後に実行するメソッドが異なるという事例が多かった．たとえば，Ant には，**図 4** のようなメソッドの組が存在した（スペースの都合上，ソースコードの行数が短くなるようレイアウトなどを変更している）．これらのメソッドは，同じ制御構造を持つが，最後に出力する数値の選択だけが異なっている．類似した例として，JHotDraw では，同一の制御構造で特定のオブジェクトを取得した後，`addListener` でリスナを追加する処理と，`removeListener` でリスナを削除する処理がコードクローンとなっていた．このように共通のデータを使用し，互いに関係した処理であっても，異なる処理を実行している場合はリファクタリングなどで共通化することは難しい．その一方で，新機能の拡張の際には，

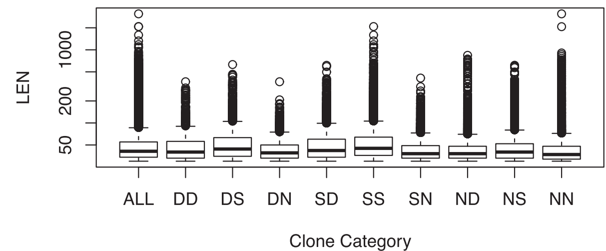


図 5 分類ごとの LEN の分布
Fig. 5 The distributions of code clone length for each category.

表 5 クローンに関するメトリクスのカテゴリ別分布の統計値
Table 5 Quantile values.

	1st Qu.	Median	3rd Qu.	代表値の差
LEN	32–36	37–45	48–64	6.00003
POP	2	2–3	3–4	0.00009
RNR	0.71–0.88	0.88–0.93	0.94–0.97	0.04878

共通のデータを取り出す方法として同様のコードを記述して利用できる可能性もある．以上の結果は手作業での確認をともなう調査のため，サンプル数に限りはあるという制限はあるが，機械的な分類結果が意味的な違いを反映していることを示している．

4.3 分類ごとのメトリクス値の比較

提案手法で得られた分類がコードクローンに対するメトリクスによる分類では得られないことを確認するために，既存研究でリファクタリングすべきコードクローンの選定 [8] に使われたメトリクス LEN，RNR，POP について，コードクローン全体の分布と，各分類に属するクローンセットだけで，メトリクス値の分布の比較を行った．例として，コードクローンの長さ LEN の分布を比較したものを**図 5** に示す．図中の分類名は，P_1 と P_2 の文字を連結した表記であり，たとえば SD は $P_1(C) = S$，$P_2(C) = D$ であるクローンの LEN の分布を表す．メトリクスの最小値はどの分類でもフィルタリング基準の 30 となっており，中央値が 37 から 45 までと狭い範囲に収まっていることから，分布に大きな差がないことが読み取れる．LEN，RNR，POP それぞれについて，コードクローンの分類ごとに四分位点の範囲を求めた結果を**表 5** に示す．表における「代表値の差」は，9 つのコードクローンの分類から任意の 2 つの分類を選んだときのメトリクス値の分布の差のうちの最大値である．この値は，2 群からすべての値の組み合わせを取り出したときの差の中央値に当たり，R における `wilcox.test` を使用して信頼区間 99%で求めた．この値が分布の範囲に比べて小さいことから，異なる分類のコードクローンだからといってメトリクスの値に大きな差があるわけではなく，メトリクスによる単純な並べ替えなどで特定の分類に属するコードクローンを取り出すことは期待できないと考えられる．

© 2015 Information Processing Society of Japan

表 6 LEN の値で上位 10 件となるコードクローンの分類
Table 6 Classification of top-10 code clones filtered by LEN.

	$P_2(C) = D$	$P_2(C) = S$	$P_2(C) = N$
$P_1(C) = D$	17	92	14
$P_1(C) = S$	78	383	29
$P_1(C) = N$	47	169	31

表 7 POP の値で上位 10 件となるコードクローンの分類
Table 7 Classification of top-10 code clones filtered by POP.

	$P_2(C) = D$	$P_2(C) = S$	$P_2(C) = N$
$P_1(C) = D$	11	56	62
$P_1(C) = S$	62	167	61
$P_1(C) = N$	172	229	40

表 8 RNR の値で上位 10 件となるコードクローンの分類
Table 8 Classification of top-10 code clones filtered by RNR.

	$P_2(C) = D$	$P_2(C) = S$	$P_2(C) = N$
$P_1(C) = D$	10	48	11
$P_1(C) = S$	63	249	83
$P_1(C) = N$	83	278	35

解析対象であった 86 個のプログラムから抽出されたコードクローンに対して，LEN，POP，RNR の値で上位 10 件であるようなクローンセットをそれぞれ抽出した結果を提案手法で分類したものを**表 6**，**表 7**，**表 8** に示す．既存のコードクローンの選定方法 [8] では，これらのコードクローンが優先的に調査対象となるが，提案手法での 9 つの分類が混在した結果を提示していたと考えられる．本研究の分類によって，たとえば共通のデータを取り出す処理を探したいのであれば $P_1 = D$，$P_2 = S$ のコードクローンの中から探す，共通処理を探したい場合は $P_1 = S$，$P_2 = D$ のコードクローンの中から調査する，というように，注目したいコードクローンの性質が決まっている場合には，さらに調査対象の範囲となるコードクローンを絞り込むことができる．

5. 妥当性への脅威

本研究ではコードクローンに含まれるメソッド呼び出しが 1 つでもメソッドのサマリに含まれていれば，それを共通機能を実現するためのクローンに分類している．実際には，メソッドのサマリにはクローンに含まれないメソッド呼び出しも含まれている場合があり，完全に共通機能を実現するためだけのクローン以外もこの分類に含めている可能性がある．

CCFinder が検出するコードクローンはタイプ 2 であり，ソースコードがコピーされた後に命令の挿入や削除が行われると，該当部分がコードクローンに含まれない場合がある．そのため，実際にコピーされたコードの範囲よりも狭い範囲だけに解析が限定されている可能性がある．

本研究で使用したサマリ抽出技術と本研究での議論は，いずれもメソッド名を重要な手がかりとして使用している．メソッドの処理内容はメソッドに付与された動詞と一貫性があることが多いことが知られており [25], [26]，メソッドの概要に当たるキーワードとメソッド名が合致することも多い [27] ことから，単一プログラム内部のコードクローンの分析において名前の一致を判断に用いることには妥当性がある．ただし，メソッド名が一致してもその振舞いが同一であるという保証はなく，開発者による不適切なメソッドの命名の影響を受けている可能性がある．

本研究ではバイナリとソースコードの対応が取れなかったコードクローンを調査対象から除外しており，コーパスに収録されたプログラムのすべてのコードクローンが調査対象となっているわけではない．そのため，すべてのコードクローンを対象とした場合に結果が変わる可能性がある．

6. おわりに

本研究では，複数のメソッドに共通の振舞いを提供するコードクローンと，異なる振舞いの実現に共通する処理だけを提供するコードクローンとを自動的に分類する手法を提案した．メソッドのサマリに含まれるメソッド呼び出しと，含まれないメソッド呼び出しのそれぞれが異なる確率に大きな違いはなく，従来研究では単に「メソッド呼び出しが異なる」という一括りにされていたコードクローンを 2 つのグループに分類することができる．

自動分類の結果が開発者の活動にもたらす有用性の評価は今後の課題である．コードクローンに対するリファクタリングや，1 つのコード片への修正を他のコードクローンにも適用するといった活動において，対策を取るコードクローンを絞り込むために使用することが考えられる．また，本研究で導入した分類は，メソッド呼び出しの差異にのみ基づいており，コードクローン中に出現する変数や定数，型名の違いは反映していない．このようなメソッド呼び出し以外の情報を考慮した意味的な分類の拡張も，今後の課題である．

謝辞 本研究は科研費 Nos.25220003，26280021 の助成を得た．

参考文献

[1] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison Wesley (1999).
[2] Yoshida, N., Higo, Y., Kusumoto, S. and Inoue, K.: An Experience Report on Analyzing Industrial Software Systems Using Code Clone Detection Techniques, *Proc. APSEC*, pp.310–313 (2012).
[3] Yoshida, N., Choi, E., Yamanaka, Y. and Inoue, K.: How We Know the Practical Impact of Clone Analysis, *Proc. IWSC*, pp.1–5 (2014).
[4] Roy, C.K. and Cordy, J.R.: Scenario-Based Comparison of Clone Detection Techniques, *Proc. ICPC*, pp.153–162

© 2015 Information Processing Society of Japan

(2008).

[5] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Software Engineering*, Vol.28, No.7, pp.654–670 (2002).

[6] Sridhara, G., Hill, E., Muppaneni, D. and Pollick, L.: Towards Automatically Generating Summary Comments for Java Methods, *Proc. ASE*, pp.43–52 (2010).

[7] Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and Noble, J.: Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies, *Proc. APSEC*, pp.336–345 (2010).

[8] Choi, E., Yoshida, N., Ishio, T., Inoue, K. and Sano, T.: Extracting Code Clones for Refactoring Using Combinations of Clone Metrics, *Proc. IWSC*, pp.7–13 (2011).

[9] Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proc. ISESE*, pp.83–92 (2004).

[10] Zhang, G., Peng, X., Xing, Z. and Zhao, W.: Cloning practices: Why developers clone and what can be changed, *Proc. ICSM*, pp.285–294 (2012).

[11] Juergens, E., Deissenboeck, F. and Hummel, B.: Code Similarities Beyond Copy and Paste, *Proc. CSMR*, pp.78–87 (2010).

[12] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: Memory Comparison-based Clone Detector, *Proc. ICSE*, pp.301–310 (2011).

[13] 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).

[14] CCFinder in Articles, available from ⟨http://www.ccfinder.net/ccfinderinarticle.html⟩.

[15] AIST CCFinderX, available from ⟨http://www.ccfinder.net/ccfinderx-j.html⟩.

[16] Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and Implementation for Investigating Code Clones in a Software System, *Information and Software Technology*, Vol.49, No.9-10, pp.985–998 (2007).

[17] Hayase, Y., Lee, Y.Y. and Inoue, K.: A Criterion for Filtering Code Clone Related Bugs, *Proc. DEFECTS*, pp.37–38 (2008).

[18] Xing, Z., Xue, Y. and Jarzabek, S.: CloneDifferentiator: Analyzing Clones by Differentiation, *Proc. ASE*, pp.576–579 (2011).

[19] Lin, Y., Xing, Z., Xue, Y., Liu, Y., Peng, X., Sun, J. and Zhao, W.: Detecting Differences across Multiple Instances of Code Clones, *Proc. ICSE*, pp.164–174 (2014).

[20] Choi, E., Yoshida, N. and Inoue, K.: An Investigation into the Characteristics of Merged Code Clones during Software Evolution, *IEICE Trans. Information and Systems*, Vol.E97-D, No.5, pp.1244–1253 (2014).

[21] 工藤良介，伊達浩典，石尾 隆，井上克郎：リファクタリング支援のためのコードクローン間の識別子名の対応関係分析，情報処理学会研究報告，Vol.2011-SE-173, No.8, pp.1–8 (2011).

[22] 工藤良介，伊達浩典，石尾 隆，井上克郎：コードクローンに含まれるメソッド呼び出しの変更度合の調査，情報処理学会研究報告，Vol.2013-SE-179, No.15, pp.1–8 (2013).

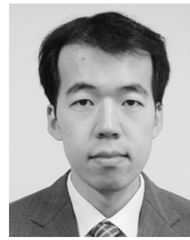
[23] Duszynski, S., Knodel, J. and Becker, M.: Analyzing the Source Code of Multiple Software Variants for Reuse Potential, *Proc. WCRE*, pp.303–307 (2011).

[24] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Programming Languages and Systems*, Vol.12, No.1, pp.26–60 (1990).

[25] Høst, E.W. and Østvold, B.M.: The Programmer's Lexicon, Volume I: The Verbs, *Proc. SCAM*, pp.193–202 (2007).

[26] Yu, S., Zhang, R. and Guan, J.: Properly and Automatically Naming Java Methods: A Machine Learning Based Approach, *Advanced Data Mining and Applications*, Lecture Notes in Computer Science, Vol.7713, Springer Berlin Heidelberg, pp.235–246 (2012).

[27] De Lucia, A., Di Penta, M., Oliveto, R., Panichella, A. and Panichella, S.: Using IR methods for labeling source code artifacts: Is it worthwhile?, *Proc. ICPC*, pp.193–202 (2012).



石尾 隆 （正会員）

2003 年大阪大学大学院基礎工学研究科博士前期課程修了．2006 年同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）．2007 年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助教．博士（情報科学）．プログラム解析，プログラム理解に関する研究に従事．

伊達 浩典 （正会員）

2007 年関西大学総合情報学部卒業．2009 年大阪大学大学院情報科学研究科博士前期課程修了．2014 年同大学大学院情報科学研究科博士後期課程退学．プログラム解析，ソフトウェアパターンに関する研究に従事．

井上 克郎 （フェロー）

1956 年生．1979 年大阪大学基礎工学部情報工学科卒業．1984 年同大学大学院博士課程修了．同年同大学基礎工学部助手．1984～1986 年ハワイ大学マノア校情報工学科助教授．1989 年大阪大学基礎工学部講師．1991 年同助教授．1995 年同教授．2002 年大阪大学大学院情報科学研究科教授．2012 年同大学大学院情報科学研究科・研究科長．工学博士．ソフトウェア工学，特に，ソフトウェア開発手法，プログラム解析，再利用技術の研究に従事．

© 2015 Information Processing Society of Japan