The University of Osaka
Institutional Knowledge Archive

Title	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
Author(s)	工藤, 良介; 伊達, 浩典; 石尾, 隆 他
Citation	情報処理学会研究報告. ソフトウェア工学研究会報告. 2013, 2013-SE-179(15), p. 1-8
Version Type	VoR
URL	https://hdl.handle.net/11094/50218
rights	© 2013 Information Processing Society of Japan
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

Vol.2013-SE-179 No.15
2013/3/11

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

工藤 良介[1,a)]　伊達 浩典[1,b)]　石尾 隆[1,c)]　井上 克郎[1,d)]

概要：コードクローンとは，互いに類似したコード片の組あるいはコード片の集合のことである．コードクローンの検出手法や利用方法については様々な研究がなされているが，コードクローンとなっているコード片自体がどのようなソースコードであるのかは明らかとなっていない．本研究では，コードクローン検出ツールの 1 つである CCFinder が検出するコードクローンを対象として，その中に含まれるメソッド呼び出しに着目した分析を行った．その結果，コード片ごとに多少の差異が含まれることはあるが，メソッドの中にある「重要なメソッド呼び出し」の記述は同一であることが多いことから，類似した処理を実装したコード片がコードクローンとして検出されていることを確認した．

キーワード：コードクローン, CCFinder, Java, メソッド呼び出し

An Empirical Study on Method Call Differences among Code Clones

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

Abstract: A code clone is a code fragment that has identical or similar portion in source code. While various code clone detection tools and their applications have been reported, source code characteristics of code clones in detail are not investigated. In this research, we have analyzed method calls involved in code clones which are detected by CCFinder. As a result, code clones often involve the same “important method calls” that likely implement a similar functionality in different source code locations.

Keywords: Code clones, CCFinder, Java, Method calls

1. はじめに

コードクローンとは，互いに類似したコード片の組あるいはコード片の集合のことであり，多くの開発者にとってコードクローンはソフトウェア保守に問題をもたらす可能性があると認識されている．そのため，ソースコード中からコードクローンを検出するツールの開発や，コードクローンを解消するための手法の研究が行われている．

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] r-kudou@ist.osaka-u.ac.jp
[b)] h-date@ist.osaka-u.ac.jp
[c)] ishio@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

コードクローンの検出手法はそれぞれ字句系列や構文木などの様々なソースコードの特徴に着目しており，たとえば CCFinder[1] の場合は，ソースコードをトークン列として，変数名やメソッド名の違い，空白の違いを無視して比較し，ある閾値よりも長く一致したトークン列の範囲をコードクローンとして検出する．このように，検出のための定義は明確であるが，CCFinder が検出するコードクローンが実際にどのような処理を多く含んでいるか，またそれらのコードクローンがどのような理由で作成されているのかについては，はっきりとわかっていない．コードクローンが作られる事例として，たとえば，オブジェクト指向プログラムでは，開発者がほとんど同一の処理を複数の場所に記述しなくてはならない場合があると言われている [2]．一方で，制御構造だけを再利用してメソッド呼び出しを変更

© 2013 Information Processing Society of Japan

する場合もある [3]．どのようなソースコードをコードクローンとして検出しているのか，その内容に関する性質を知ることは，クローンの分析手法を構築する上で重要な課題である．

本研究では，コードクローンが持つ特徴を明らかにするための第一歩として，メソッドの内部に含まれるコードクローンについて，そのメソッド呼び出しに着目した調査を行った．具体的には，コードクローン中に出現する「重要なメソッド呼び出し」のメソッド名を比較することで，クローンがメソッドの内部で重要な処理をコピーしたものであるのか，それとも同じ制御構造を持つが異なる処理を作成したものであるのかを調査した．調査の範囲は，Java で開発された 6 つのオープンソースソフトウェアのソースコードから，CCFinder が検出したコードクローンである．

以下，2 章では本研究の背景について解説し，3 章では調査方法について述べる．4 章では調査の結果を示しそれに対する考察を述べる．最後に 5 章では本研究の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

開発者が記述するソースコードがある程度以上の長さで偶然一致する確率は低く [4]，多くの場合，コードクローンは開発者がソースコードを意図的に複製して再利用することによって生じる [3]．複製されたソースコードはそのまま使えるとは限らないため，たとえば変数名を複製先のソースコードに合わせるなど，様々な編集が行われる [5]．

CCFinder[1] は，ソースコードの変数名やメソッド名の変更といった単純な修正を考慮した，Bellon らの分類でいうタイプ 2 クローンを検出するツールである [6]．DECKARD[7] のような構文木を用いる検出ツールは，文の入れ替えなど，より多くのコード修正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が，本研究では次の 2 つの理由から，分析対象を CCFinder が検出するコードクローンとした．

- CCFinder はトークンの列を比較するため，あるコード片のどのトークンが，他のコード片のどのトークンに対応づけられたかを知ることができる．すなわち，開発者がソースコードをコピーした後に変更したと思われる範囲をコードクローンの内容から判断することができる．
- トークン列の比較を用いたコードクローン検出の実装として CCFinderX[*1]が公開されており，検出されるコードクローンに関する知識が，企業などで行われている様々な分析活動に役立つ可能性がある．

[*1] AIST CCFinderX, http://www.ccfinder.net/ccfinderx-j.html

2.2 重要なメソッド呼び出し

Sridhara ら [8] は，「メソッドの概要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 らは以下の 3 つの選択基準を挙げている．

- 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．
- 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．
- 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．

Sridhara らの手法は，これらの条件で選ばれたプログラム文を実行するために必要なデータを準備するプログラム文や実行制御文も同時に抽出し，コメントへと変換する．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．

本研究では，この手法で抽出される「メソッドの概要を説明するコメントの内容としてふさわしい」メソッド呼び出しを，「重要なメソッド呼び出し」と呼ぶ．コードクローンが「重要なメソッド呼び出し」を多くコピーし，そのまま再利用しているのかどうかを調査することで，クローンがメソッド内の重要な処理をコピーしたものであるのか，それとも同じ制御構造を持つが異なる処理を作成したものであるのかを明らかにする．

重要なメソッド呼び出しの例を**図 1** に示す．この図の 4 行目の addElement(input) は戻り値が利用されていないため，このメソッドで実行されるべき何らかの副作用を担っている「重要なメソッド呼び出し」となる．また，コードの 6 行目の append(result) は returnPressed メソッドの末尾にあたるため，このメソッドが最終的に実行しなければならない処理を表現する「重要なメソッド呼び出し」となる．そして，そのメソッド呼び出しの引数に使われている変数 result に関するデータ依存関係を辿ると，5 行目の evaluate(input) 呼び出しの戻り値が result に代入されていることから，これも「重要なメソッド呼び出し」となる．

3. 調査方法

本研究では，CCFinder が検出したコードクローンについて，以下の 2 つの調査質問に関する分析を行った．

RQ1 コードクローンは，その出現場所によって処理の内容が異なることはあるのか．

RQ2 コードクローンは，メソッドの主要な処理を行なっ

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

図 1 ending，void-return，data-facilitating のコード例 [8]
Fig. 1 An example of ending, void-return and data-facilitating method calls [8]

ている部分をコピーしたものであるのか．

3.1 調査対象

調査対象としてプログラミング言語 Java で記述された 6 つのオープンソースソフトウェアを選択した．コードクローンの検出に使用した CCFinder のバージョンは 7.2.4.0 である．検出のパラメータはすべて CCFinder のデフォルト値であり，30 トークン以上の長さのコードクローンを調査対象とした．ただし，CCFinder が検出したコードクローンのうち，コードクローンメトリクス RNR[9] の値が 0.5 以下となるクローンセットを調査対象から除外している．RNR とはコードの非繰り返し率のことであり，コードに同じ並びの繰り返しが多いと，この値が低くなる．RNR の値が低いコードは変数の並びや if 文などの制御文の繰り返しになっていることが多く，開発者にとっては興味がないものが多いことが経験的にわかっているため，分析対象から除外した．調査対象の 6 つのソフトウェアの一覧とソフトウェアの規模，検出されたクローンセットの数，RNR の値によるフィルタリングを行い最終的に分析を行ったクローンセットの数を**表 1** に示す．

3.2 識別子の変更

CCFinder によって検出されるコードクローンはコードごとのトークン数が等しく，またトークンの種類の並びも一致することが保証されているため，互いにクローンとなっているコード間で「同じ位置に存在する識別子」を決定できる．この「同じ位置に存在する識別子」の名前を比べたときに，クローンセット内のすべてのコードで名前が一致する場合，この識別子は「名前が変更されていない」，それ以外の場合は「名前が変更されている」と定義する．各識別子がメソッド呼び出しであるかどうかは各ソースファイルの構文木によって判断することができるので，「同じ位置に存在するメソッド呼び出し」の名前を比べることで「名前が変更されていないメソッド呼び出し」，「名前が変更されているメソッド呼び出し」を区別する．例として，**図 2** 中のコード 1 とコード 2 がクローンとして検出された場合に，これら 2 つのコードに含まれる識別子を順番に抽出して比較した結果を**表 2** に示す．

```
x = getX();
z = 3;
n = getN(x, z);
return n;
```

コード 1

```
y = getY();
z = 3;
n = getN(y, z);
return n;
```

コード 2

図 2 コードクローンの例
Fig. 2 An example of code clone

3.3 重要なメソッド呼び出しの認識

「重要なメソッド呼び出し」は，解析対象のソフトウェアの各メソッド定義を解析することで得られる．Sridhara ら [8] の定義した重要なプログラム文の判定には，Java ソースコードの構文だけでなく制御フロー，データフローの解析が必要であることから，本研究では Java バイトコード解析を用いて，Sridhara らの定義に従った抽出の手順を以下のように実装した．

まず，解析対象のメソッドに対して，制御フローグラフとプログラム依存グラフ（PDG）[10] を構築する．ただし，Java のバイトコードは演算にオペランドスタックとローカル変数の 2 種類を用いるため，データ依存関係は，オペランドスタックを経由したものと，ローカル変数を経由したものを区別して抽出しておく．得られたグラフを用いて，ending，void-return，same-action という 3 つのメソッド呼び出し命令の集合を以下のように抽出する．

ending ソースコードにおいてメソッド末尾の文か，return 命令の戻り値の式に登場するメソッド呼び出し命令の集合．バイトコードでは，戻り値を返さない RETURN 命令の直前にある命令，あるいは値を返す RETURN 命令自体を基点として，Java 仮想マシンのスタックを経由したデータ依存関係を後ろ向きに辿って到達可能な範囲にあるメソッド呼び出し命令の集合となる．

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命令の集合．バイトコードでは，戻り値が使用されない場合，メソッド呼び出し命令の直後に戻り値を破棄する命令が挿入されるこ

表 2 図 2 のコードクローンに含まれる識別子
Table 2 Identifiers involved in the code clones of Fig.2

コード 1	コード 2	種類	変更の有無
x	y	変数	あり
getX	getY	メソッド呼び出し	あり
z	z	変数	なし
3	3	リテラル	なし
n	n	変数	なし
getN	getN	メソッド呼び出し	なし
x	y	変数	あり
z	z	変数	なし
n	n	変数	なし

© 2013 Information Processing Society of Japan

表 1　調査対象のソフトウェアと検出されたクローンセット数
Table 1　Analyzed software and their code clones

Software	Version	#File	LOC	#Method	#Clone Set	#Analyzed Clone Set
Derby[*1]	10.9.1.0	1,445	549,911	21,653	2,302	1,384
h2[*2]	1.3.168	500	135,114	7,184	1,315	569
jTunes[*3]	(2009.12.12)	519	134,243	7,296	1,324	675
Tomcat[*4]	7.0.27	1,242	348,856	16,916	3,518	1,962
XXL[*5]	1.0	633	178,230	7,914	832	509
zk[*6]	6.5.0	406	77,772	4,851	338	229

とを利用して，該当するメソッド呼び出し命令の集合を抽出することができる．

same-action　呼び出しているメソッド名を CamelCase のルールに従って単語単位に分割した場合に，その先頭単語が解析対象のメソッドと一致するメソッド呼び出し命令の集合．

ただし，以下の条件のいずれかに当てはまるメソッド呼び出し命令は重要でないと判断し除外する．

- void-return と same-action メソッド呼び出し命令のうち，コンストラクタ，名前が get あるいは set から始まるメソッド，文字列の加算に使用される StringBuilder クラスおよび StringBuffer クラスの append メソッドに対する呼び出し命令．
- 例外ハンドラの中に記述されている呼び出し命令．
- ある変数が null であること，または null でないことが実行条件となるようなメソッド呼び出し命令．
- ロギングや例外処理に関連すると思われるメソッド呼び出し．すなわち，メソッド名に log, trace, error, debug, exception, close が含まれるメソッド呼び出し命令．

以上の手順で求めた ending，void-return，same-action のメソッド呼び出しに対してデータを提供する処理，実行を制御する処理を data-facilitating と controlling として以下のように定義する．

data-facilitating　各メソッド呼び出しに対してデータを供給するメソッド呼び出し命令の集合．ending，void-return，same-action に含まれるメソッド呼び出し命令の引数それぞれを基点として，オペランドスタックによるデータ依存関係を辿り，ローカル変数に関するデータ依存関係をただ 1 度だけ辿り（プログラム文間のデータ依存関係に相当する），さらにオペランドスタックによるデータ依存関係を辿って到達できる範囲にあるメソッド呼び出し命令を抽出する．ただし，void-return および same-action に対して行ったのと同様に，コンストラクタ，名前が get あるいは set から始まるメソッド，文字列の結合に関するメソッド呼び出し命令は，この候補から除外する．

controlling　ending，void-return，same-action，data-facilitating の各命令が制御依存する条件分岐について，その条件式に使われている（条件分岐命令からオペランドスタックに関するデータ依存関係で到達可能な範囲にある）メソッド呼び出しの集合を取り出す．具体的には，制御依存関係解析を用いて，各メソッドの実行条件となるすべての（推移的な制御依存関係を持つ）条件分岐命令を取得し，それらの条件分岐の値に使われたメソッド呼び出し命令を取り出す．ここでも，get/set 等のメソッド呼び出しは候補から除外する．

この一連の処理を経て求めた, ending, void-return, same-action，data-facilitating，controlling の 5 つのメソッド呼び出し集合の和集合が，最終的に求める「重要なメソッド呼び出し」となる．コードクローンはソースコードに対して計算されるので，コードクローンに含まれるメソッド呼び出しとバイトコードに出現する呼び出しとは，行番号と名前によって対応付ける．

3.4　調査するメトリクス

各クローンセットに対して，以下の情報を収集した．

- クローンセット内のコードに含まれるすべてのメソッド呼び出しを，「重要なメソッド呼び出し」と「重要でないメソッド呼び出し」に分ける．「重要なメソッド呼び出し」のうち，名前が変更されているメソッド呼び出しの割合を「重要なメソッド呼び出しの変更度合 (A)」とする．また，「重要でないメソッド呼び出し」のうち，同様に名前が変更されているメソッド呼び出しの割合を「重要でないメソッド呼び出しの変更度合 (B)」とする．
- クローンセット内のすべてのメソッド呼び出しのうち，「重要なメソッド呼び出し」の割合を，「クローンとなっているコードの重要なメソッド呼び出しの割合

*1 Apache Derby, http://db.apache.org/derby/
*2 h2, http://www.h2database.com/html/main.html
*3 jTunes, http://sourceforge.jp/projects/sfnet_jtunes-online/
*4 Apache Tomcat, http://tomcat.apache.org/
*5 XXL, http://dbs.mathematik.uni-marburg.de/home/research/projects/xxl/
*6 zk, http://www.zkoss.org/

© 2013 Information Processing Society of Japan

(C)」とする．

- クローンセットの各コード片が存在しているメソッドの集合を取り出し，それらに含まれるメソッド呼び出しを列挙する（コードクローンを 1 行でも含んでいれば，そのメソッドはクローンが存在しているメソッドとする）．その中で，「重要なメソッド呼び出し」の割合を，「クローンが存在しているメソッド全体での重要なメソッド呼び出しの割合 (D)」とする．

なお，「重要なメソッド呼び出し」は，クローンセット内のあるコード片では重要であっても，別のコード片の対応する位置にあるメソッド呼び出しでは重要でないということがありうる．たとえば，開発者がメソッド全体をコピーして，末尾に 1 つメソッド呼び出しを書き加えると，元々末尾にあったメソッド呼び出しは，ending の条件から外れてしまう．このような場合であっても，(A)，(B)，(C)，(D) の 4 つの値は計算可能である．

4. 調査結果

4.1 重要なメソッド呼び出しと重要でないメソッド呼び出しの比較

RQ1 に関して，まず，各ソフトウェアの「重要なメソッド呼び出し」が変更されていないクローンセットの数とその割合を**表 3** に示す．「重要なメソッド呼び出し」が変更されていないコードクローンは 87.0%となっており，すべての呼び出しが変更されていないコードクローンの 60.9%に比べても高い値となっている．「重要なメソッド呼び出し」の名前が変更されることが重要でないメソッド呼び出しに比べて少ないことを確認するため，各クローンセットの「重要なメソッド呼び出しの変更度合 (A)」と「重要でないメソッド呼び出しの変更度合 (B)」の分布について，Wilcoxon の順位和検定を適用した．各ソフトウェアの (A)，(B) の値の度数分布表を**表 4** に示す．検定結果を**表 5** に示すが，すべてのソフトウェアにおいて有意水準 1%で帰無仮説を棄却し，2 群に差があることを確認した．このことから，RQ1 に対する答えは，「重要なメソッド呼び出しの名前は変更されることは少なく，コードがコピーされてもその主要な処理が変わることは少ない」というものになる．

表 3 メソッド呼び出しが変更されていないクローンセットの数
Table 3 Clone sets whose “important method calls” are not modified

	重要な呼び出しのみ変更なし		全呼び出しが変更なし	
Software	個数	割合	個数	割合
Derby	1,308	94.5%	833	60.2%
h2	570	79.3%	292	51.2%
jTunes	599	88.7%	501	74.2%
Tomcat	1,661	84.7%	1,183	60.3%
XXL	423	82.9%	324	63.5%
zk	196	85.6%	114	49.8%
合計	4,639	87.0%	3,247	60.9%

表 5 (A) と (B)，(C) と (D) に対する Wilcoxon の順位和検定
Table 5 Wilcoxon’s rank sum test for (A) and (B), (C) and (D)

	(A) と (B)		(C) と (D)	
Software	p 値	帰無仮説	p 値	帰無仮説
Derby	< 0.0001	棄却	0.0013	棄却
h2	< 0.0001	棄却	0.0538	棄却されない
jTunes	0.0057	棄却	0.0730	棄却されない
Tomcat	< 0.0001	棄却	< 0.0001	棄却
XXL	0.0032	棄却	0.4451	棄却されない
zk	< 0.0001	棄却	0.4197	棄却されない

コードクローンの特徴をさらに詳しく調べるため，「重要なメソッド呼び出し」が変更されていないクローンセットと「重要なメソッド呼び出し」が多く変更されているクローンセットをそれぞれ各ソフトウェアから 5 例ずつ計 30 例を無作為に取り出し，実際にコードを閲覧して調査した．

「重要なメソッド呼び出し」が変更されていないクローンセットに含まれるコードを調べた結果，30 例のうち 22 例で，「別パッケージの同名クラス」や「別クラスの同名メソッド」など，完全に同一ではないが，存在するパッケージ名，クラス名，メソッド名のいずれかが同じであるコードがクローンセットとなっていた．たとえば，Tomcat の，AjpNioProcessor クラスの process メソッドと，AjpAprProcessor クラスの process メソッドから「重要なメソッド呼び出し」が変更されていないクローンセットが検出されている．

一方，「重要なメソッド呼び出し」の変更度合が高いクローンセットを調べたところ，制御構文を繰り返し記述しているコードが多く確認された．これは，開発者にとって興味がないと思われる種類のクローンセットであるが，繰り返し回数が少ないため RNR によってフィルタリングされなかったと考えられる．**図 3** はコード 1，コード 2 ともに zk の ConfigParser クラスから検出されたコードであり，この 2 つのコードがクローンセットとなっている．この例では，制御構文が繰り返し現れて，それぞれの制御構文の中で戻り値のないメソッドを呼び出している．このメソッド呼び出しは void-return の「重要なメソッド呼び出し」である．このメソッド呼び出しで呼び出しているメソッドがクローン間で異なるため，「重要なメソッド呼び出し」の変更度合が高くなっている．このような，制御構文の繰り返しで記述されたクローンセットが多数確認され，調査した 30 例中 18 例がこれに該当した．また，残った 12 例中 7 例は，戻り値のないメソッドを繰り返し呼び出しているコードであった．これらのコードクローンは開発者が検出したいコードクローンである可能性は低い．このことから，「重要なメソッド呼び出し」の変更度合が高いクローンセットをフィルタリングすることで，開発者の興味のないコードクローンを除去し，目的のコードクローンを検出する手助

表 4 (A) および (B) の度数分布表
Table 4 Frequency distribution table of (A) and (B)

	Derby		h2		jTunes		Tomcat		XXL		zk	
	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)
[0.0 − 0.1]	1,308	886	454	368	601	567	1,662	1,430	423	390	196	136
(0.1 − 0.2]	6	40	11	71	11	21	24	90	12	16	1	14
(0.2 − 0.3]	4	23	4	24	20	16	19	76	15	13	2	4
(0.3 − 0.4]	10	59	22	42	6	28	49	99	19	19	6	18
(0.4 − 0.5]	23	47	27	19	7	9	70	85	18	24	10	9
(0.5 − 0.6]	0	6	3	3	0	0	9	22	2	1	3	1
(0.6 − 0.7]	4	13	13	10	3	2	20	15	5	5	2	2
(0.7 − 0.8]	2	3	3	1	1	2	7	12	0	2	0	2
(0.8 − 0.9]	0	0	0	1	0	0	0	8	0	0	0	0
(0.9 − 1.0]	27	307	32	30	26	30	102	125	15	39	9	43

```
if (v != null)
  config.setSessionMaxRequests(v.intValue());
v = parseInteger(el, "max-pushes-per-session", ANY_VALUE);
if (v != null)
  config.setSessionMaxPushes(v.intValue());
String s=el.getElementValue("timer-keep-alive", true);
if (s != null)
  config.setTimerKeepAlive("true".equals(s));
```

コード 1

```
if (v != null)
  config.setProcessingPromptDelay(v.intValue());
v = parseInteger(conf, "tooltip-delay", POSITIVE_ONLY);
if (v != null)
  config.setTooltipDelay(v.intValue());
String s=conf.getElementValue("keep-across-visits", true);
if (s != null)
  config.setKeepDesktopAcrossVisits(!"false".equals(s));
```

コード 2

図 3 「制御構文の繰り返し」のコードクローンの例
Fig. 3 A code clone including repeated control statements

けとなることが期待される．

4.2 クローン部分とクローンが含まれるメソッド全体でのメソッド呼び出しの比較

RQ2 について調査するために，各ソフトウェアから検出されたクローンセットについて，「クローンとなっているコードの重要なメソッド呼び出しの割合 (C)」と「クローンが存在しているメソッド全体での重要なメソッド呼び出しの割合 (D)」を取得し，分布に差があるかどうかを Wilcoxon の順位和検定により検証した．表 5 に示すように，6 つのソフトウェア中 4 つのソフトウェアでは，有意水準 1% で帰無仮説は棄却されず，(C) と (D) の分布に差があるということはできない．つまり，コードクローンが作成される際，「重要なメソッド呼び出し」を多く含む部分，すなわちメソッド内の主要な処理が意図的に選択されてコピーされているとは限らないと考えられる．

4.3 「重要なメソッド呼び出し」の特徴

本研究の調査は，Sridhara らが定義した「メソッドの概要を説明する」メソッド呼び出しに依存している．Sridhara らの調査 [8] では抽出結果に対する有用性についての評価は行われているが，どのようなメソッド呼び出しが具体的に「重要なメソッド呼び出し」と判定されるかは詳しく記述されていない．そこで，どのようなメソッドが「重要なメソッド呼び出し」によって呼び出され，どのようなメソッドが「重要でないメソッド呼び出し」によって呼び出されるのか調査を行った．

4.3.1 メソッドの分類

まず，各ソフトウェアのソースコード中で宣言されているメソッドを，常に重要なメソッド呼び出しによって呼び出されるもの（すべての呼び出し元が「重要なメソッド呼び出し」であるもの），常に重要でないメソッド呼び出しによって呼び出される（すべての呼び出し元が「重要でないメソッド呼び出し」であるもの），どちらからも呼び出されるもの，一度も呼び出されないものの 4 種類に分類した．ここでの解析対象は，ソースファイル内のメソッド定義およびメソッド呼び出しのみである．そのため，解析対象以外のパッケージなどから呼び出されるように設計されているメソッドは「一度も呼び出されない」に分類される．ソフトウェアごとの結果を**表 6** に示す．数字は，それぞれ分類されたメソッドの数を表す．表 6 からわかる通り，「常

表 6 メソッドの分類
Table 6 Category of method

Software	常に重要	常に重要でない	どちらからも呼び出される	一度も呼び出されない
Derby	4,930	5,126	1,312	16,064
h2	4,142	6,663	1,995	7,568
jTunes	2,162	4,131	647	3,781
Tomcat	4,202	7,435	1,645	9,158
XXL	1,938	2,963	691	4,303
zk	1,411	1,845	538	2,875

© 2013 Information Processing Society of Japan

に重要」,「常に重要でない」メソッドに比べ「どちらからも呼び出される」メソッドの数は少なくなっている．このことから,「重要なメソッド呼び出し」と「重要でないメソッド呼び出し」がそれぞれ呼び出すメソッドはある程度傾向が分かれていると推測される．

4.3.2 メソッド名に使用される単語の調査

次に,「常に重要なメソッド呼び出しによって呼び出される」メソッドと「常に重要でないメソッド呼び出しによって呼び出される」メソッド，それぞれのメソッド名でよく使用されている単語について調査を行った．方法としては，メソッド名を CamelCase に従い単語に分割し，先頭で使用されている単語を集計して頻度順に並べ替えることで調査した．なお，調査するのは頻度の上位 10 単語とした．まずは「常に重要なメソッド呼び出しによって呼び出される」メソッド,「常に重要でないメソッド呼び出しによって呼び出される」メソッドそれぞれの結果を**表 7** に示す．

表 7 を見ると get や is などの単語がどちらにも含まれている．一方でのみよく使用される単語を示すため,「常に重要なメソッド呼び出しによって呼び出される」メソッドと「常に重要でないメソッド呼び出しによって呼び出される」メソッドで共通して存在する単語には「—」を引いた．なお，ここでの「共通して存在する単語」とは，ソフトウェアが同じかどうかを考慮しない．たとえば，重要なメソッド呼び出しの Tomcat の 10 番目と重要でないメソッド呼び出しの zk の 9 番目に do という単語が存在する．このように異なるソフトウェア間で共通する単語も除去の対象となる．共通する単語を除いたあとの表を見ると，重要なメソッド呼び出しからは write や remove，重要でないメソッド呼び出しからは close，to などが複数のソフトウェアに出現している．write や remove は,「書き出す」「除去する」という動作で戻り値を利用することは少なく，void-return に該当することが多いため重要なメソッド呼び出しに表れたと推測される．to は，toString などのように与えられたデータを別のかたちに変換するメソッドで使用されやすく，変換後のデータが戻り値として利用されていたため「重要なメソッド呼び出し」とならず，重要でないメソッド呼び出しに表れたと推測される．close は，3 章で定義したようにこの単語が使われたメソッドの呼び出しは ending 以外では「重要なメソッド呼び出し」の候補から除去されるため，その多くが「重要でないメソッド呼び出し」と分類され，重要でないメソッド呼び出しに表れたと考えられる．

4.4 調査結果とその妥当性

本研究における調査の妥当性について注意すべき点を以下にまとめる．

- 本研究は Java で記述されたソースコードについてのみ調査を行った．文法規則の異なる他の言語では違った傾向が見られるかもしれない．調査結果をより一般的なものとするために，対象の言語を増やす必要がある．
- メソッド呼び出しの「変更された」という判断は，その名前のみで確認しており，メソッド名が同一であっても実際の束縛先が異なる可能性については考慮していない．Java の動的束縛を考慮すると，2 つのメソッド呼び出し命令が同一のメソッドを常に呼び出すことを正確には判定できないため，本研究では名前のみを用いた．
- 本研究で対象にしたコードクローンは，CCFinder が抽出対象としている正規化しトークン列が一致するソースコード断片である．他のコードクローン検出ツールが取り出すコードクローンについては，本研究の結果があてはまるとは限らない．
- 本研究で対象にしたコードクローンは，RNR を用いたフィルタリング後のものである．RNR の値が小さいコードクローンの多くは開発者にとって意味がないコードクローンであると言われているが，その理由に関する調査は本研究には含まれていない．
- 本研究では，6 つのオープンソースソフトウェアを対象に実験を行った．これらのソフトウェアは多様なドメインに属していると考えているが，異なるドメインのソフトウェアや，企業で開発されているソフトウェアでは，異なる傾向が発見される可能性がある．
- 4.1 節では，各ソフトウェアのコードクローンからサンプルを取り出して調査を行った．サンプル数の増加，あるいはソフトウェアの変更によって，結果が変わる可能性はある．
- 「重要なメソッド呼び出し」の抽出方法は，文献 [8] を参考にして独自に作成したものである．文献 [8] で評価されている重要なメソッド呼び出しとは実装の差異による違いが存在する可能性がある．

5. まとめ

本研究ではコードクローンとなっているコードが持つ特徴を調査するため，そのメトリクスを計測し，分析を行った．その結果,「重要なメソッド呼び出し」はそうでないメソッド呼び出しに比べて名前の修正が加えられにくく，CCFinder で検出されたコードクローンの多くは，細かい差異はあるが，その出現位置であるメソッドの中で担当している重要な処理の内容はあまり変わらないことが分かった．「重要なメソッド呼び出し」が変更されていないコードクローンの多くは，同一パッケージ内部，あるいは類似した名前のクラスなどに出現しており，特定の処理をクラス間，メソッド間で共有する方法がないために作られているのだと考えられる．一方,「重要なメソッド呼び出しの変更度合」の高いコードクローンは制御構造の再利用によるものであると考えられるが,「重要なメソッド呼び出し」が変更されていないコードクローンは全体の 87%を占めてい

表 7 「常に重要なメソッド呼び出しによって呼び出されるメソッド」,「常に重要でないメソッド呼び出しによって呼び出されるメソッド」の名前に使われる単語

Table 7 Words in method names which appear in "important" calls and "not important" calls

	「常に重要」メソッドで頻出						「常に重要でない」メソッドで頻出					
	Derby	h2	jTunes	Tomcat	XXL	zk	Derby	h2	jTunes	Tomcat	XXL	zk
1	~~get~~	~~get~~	~~add~~	~~get~~	~~get~~	~~get~~	~~get~~	~~get~~	~~get~~	~~get~~	~~get~~	~~get~~
2	update	~~add~~	decode	~~add~~	write	~~add~~	~~set~~	~~set~~	~~set~~	~~set~~	~~set~~	~~set~~
3	~~is~~	~~is~~	~~get~~	~~is~~	~~read~~	~~is~~	close	close	~~create~~	~~is~~	close	~~is~~
4	~~read~~	~~read~~	~~create~~	remove	update	~~parse~~	~~is~~	~~is~~	~~is~~	close	~~is~~	to
5	init	write	paint	~~create~~	open	remove	~~new~~	~~add~~	name	~~create~~	to	~~add~~
6	~~add~~	check	remove	write	remove	on	setup	~~create~~	contains	find	~~has~~	~~new~~
7	write	remove	~~is~~	jj	~~has~~	render	~~has~~	debug	close	log	print	resolve
8	check	~~parse~~	draw	~~parse~~	~~is~~	~~new~~	to	~~read~~	~~add~~	to	~~read~~	append
9	~~set~~	init	install	run	~~add~~	~~set~~	make	log	to	~~read~~	index	~~do~~
10	bind	~~create~~	init	~~do~~	next	~~has~~	find	to	~~read~~	~~add~~	size	~~parse~~

ることから，コードクローンの多くがメソッド呼び出しによって実現されている処理の再利用であると考えられる．これらの事実は，CCFinder のコードクローン検出の手法からは直接明らかになっていなかった性質であり，その確認を行ったことが，本研究の貢献である．

今後の課題としては，対象ソフトウェアの数や種類を増やしたり，対象言語を増やすことで今回の分析結果の一般性を確認することが挙げられる．また，「重要なメソッド呼び出し」についての調査を行うことで，ソフトウェア全体における「重要な処理」とクローンとの関わりを調べていくことも考えられる．

謝辞 本研究は，日本学術振興会科学研究費補助金若手研究 (A)（課題番号:23680001）の助成を得た．

参考文献

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

[2] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M.: N degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st ACM/IEEE International Conference on Software Engineering*, pp. 107–119 (1999).

[3] Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proceedings of the 3rd International Symposium on Empirical Software Engineering*, pp. 83–92 (2004).

[4] Juergens, E., Deissenboeck, F. and Hummel, B.: Code Similarities Beyond Copy and Paste, *Proceedings of the 14th IEEE European Conference on Software Maintenance and Reengineering*, pp. 78–87 (2010).

[5] Roy, C. K. and Cordy, J. R.: Scenario-Based Comparison of Clone Detection Techniques, *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pp. 153–162 (2008).

[6] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).

[7] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105 (2007).

[8] Sridhara, G., Hill, E., Muppaneni, D. and Pollick, L.: Towards Automatically Generating Summary Comments for Java Methods, *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pp. 43–52 (2010).

[9] 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).

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

© 2013 Information Processing Society of Japan