社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

大規模ソフトウェア群に対するメソッド単位のコードクローン検出

石原　知也†　堀田　圭佑†　肥後　芳樹†　井垣　宏†　楠本　真二†

† 大阪大学大学院情報科学研究科

あらまし　ソフトウェアシステムには，他のソフトウェアにまたがるコードクローンが多数存在することが予測される．このようなコードクローンを検出することは，ライセンスに違反しているソースコードを発見できる，また複数のソフトウェア間に共通する処理をライブラリとしてまとめる等の観点から有益である．しかし既存研究では，ファイルの一部がコードクローンであるものは検出できない．本研究では大規模ソフトウェア群から，メソッド単位のコードクローンを高速に検出するツールを開発し，実験を行った．

キーワード　コードクローン，メソッドクローン，ソースコード流用，ソフトウェア保守

Method-based Code Clone Detection for a Large Number of Software Systems

Tomoya ISHIHARA†, Keisuke HOTTA†, Yoshiki HIGO†, Hiroshi IGAKI†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University

Abstract　It is predicted that there are many code clones across software systems. Detecting such code clones is useful from the viewpoint that we can discover source code licensing violations or merge common functions into libraries. However, existing research cannot detect code clones which are a part of a source file. In this research, we developed a tool that quickly detects method-based code clones and experimented on a huge set of software systems.

Key words　Code Clone, Method Clone, Source Code Plagiarism, Software Maintenance

1. は じ め に

再利用などの理由により，複数のソフトウェアにまたがるコードクローンが多数存在することが予測される [1] [2]．コードクローンとは，ソースコード中に存在する同一，あるいは類似したコード片のことである．このような複数のソフトウェアにまたがるコードクローンを検出することは，ライセンスに違反して流用されているソースコードの特定や，複数のソフトウェア間に頻出する処理のライブラリ化による開発効率の向上などの観点から有益である．しかし，既存のコードクローン検出手法 [3] [4] は 1 つのソフトウェア内のコードクローンを見つけることを目的としているため，ソースコードを文や字句などの細粒度で比較している．そのため，大規模なソフトウェア群を対象としたコードクローン検出には膨大な時間的・空間的コストを必要とする．この問題を改善し，大規模なソフトウェア群から実用的なコストでコードクローンを検出する手法として，ファイル単位のコードクローン検出手法が提案されている [5] [6]．ファイル単位のコードクローン検出手法は，大規模なソフトウェア群に対して高速にコードクローン検出を行うことができる反面，ファイルの一部がコードクローンであるものは検出できないという問題点を抱えている．例えば，ファイルのある一部分のみが流用されている場合，ファイル単位のコードクローン検出手法では流用部分をコードクローンとして検出することができない．

本研究では，ファイル単位より小さい粒度であるメソッド単位でのコードクローン検出手法を提案する．メソッド単位でのコードクローン検出手法では，大規模なソフトウェア群に対して実用的な時間で検出を行うことができると同時に，ファイル単位のコードクローン検出手法では検出できない，ファイルの一部がコードクローンであるものを検出することが可能である．実験の結果，約 3 億 6 千万行のソースコードから約 4.91 時間でコードクローンを検出することができた．また，検出されたコードクローンを分析した結果，ソースコードの流用を特定することや複数のソフトウェア間に共通する処理をライブラリ化することに有用であるコードクローンの存在を確認することができた．

```
78  public class TreeBidiMap implements
OrderedBidiMap, Serializable {

（中略）

219 public void putAll(Map t){
220  Iterator it = t.entrySet().iterator();
221  while(it.hasNext()){
222   Map.Entry entry = (Map.Entry)it.next();
223   put(entry.getKey(),entry.getValue());
224  }
225 }
```

(a) apache.commons.collections TreeBidiMapクラス

```
40  final class PortletRequestScopeMap implements
Map {

（中略）

132 public void putAll(Map t){
133  Iterator entries = t.entrySet().iterator();
134  while(entries.hasNext()){
135   Map.Entry entry = (Map.Entry)entries.next();
136   put(entry.getKey(),entry.getValue());
137  }
138 }
```

(b) apache.commons.chain ServletApplicationScopeMapクラス

図 1 Motivating Example

2. 準 備

2.1 ソフトウェア間にまたがるコードクローン

本研究では特にソフトウェア間にまたがるコードクローンに着目する．ソフトウェア間にまたがるコードクローンを検出する利点として，以下の点が挙げられる [5] [6].

他のソフトウェアから流用したソースコードの特定

ソフトウェア間にまたがるコードクローンの生成要因の 1 つとして，他のソフトウェアからソースコードを流用する場合が考えられる．流用されたソースコードの中には，ライセンスに違反して流用されたソースコードが含まれている可能性がある．ソフトウェア間にまたがるコードクローンを検出することで，ライセンスに違反して流用されているソースコードの特定を支援することができる．

複数のソフトウェアに存在する共通処理のライブラリ化

複数のソフトウェアにまたがってコードクローンとして検出されたコード片が行う処理は，複数のソフトウェアで共有されている共通の処理である．複数のソフトウェアに頻出する共通の処理は，今後のソフトウェア開発でも使用される可能性が高いと考えられる．このような複数のソフトウェアに頻出する共通の処理をライブラリ化することで，開発者が新たに処理を記述する必要がなくなり，開発効率の向上が期待できる．

2.2 ファイル単位のコードクローン

行や字句単位などの既存のコードクローン検出手法は，主に単一のソフトウェアを検出対象としている．そのため，このような既存の手法は高精度でコードクローンを検出すべくソースコードを細粒度で比較している．しかし大規模なソフトウェア群を対象に検出を行う場合，細粒度のコードクローン検出手法では比較回数が多くなり，その検出処理に膨大な時間的・空間的コストを必要とする．このため，大規模なソフトウェア群から実用的な時間でコードクローンを検出することは困難である．

この問題を解決するために，ファイル単位のコードクローン検出手法が提案されている [5] [6]．ファイル単位のコードクローン検出手法では，ファイルを単位として比較を行い，一致する場合のみコードクローンとみなす．この手法は，行単位や字句単位のような細粒度の検出手法と比べ，ソースコードの比較回数が小さくなるため，高速な検出が可能である．しかし，ファイルの一部がコードクローンであるものはコードクローンとして検出することができない．

以降，ファイル単位のコードクローンを**ファイルクローン**，メソッド単位のコードクローンのことを**メソッドクローン**と呼ぶ．

3. 研 究 動 機

3.1 既 存 研 究

佐々木らは，ファイルクローン検出ツール FCFinder を開発し，ファイルクローンがどのような性質を持っているかを調査した [5]．FCFinder はファイルをハッシュ値に変換することでファイルクローン検出を行っている．また，FCFinder はコメント文の削除や字句解析を行っているため，コメント文やインデントの違いを吸収することが可能である．佐々木らは，大規模なソフトウェア群である FreeBSD Ports Collection に対し FCFinder を適用した結果，FreeBSD Ports Collection の約 68%がファイルクローンであったことを報告している．また，検出されたファイルクローンのうち 27%はコメントやインデントの違いであり，ファイルサイズの分布はファイルクローンとそうでないファイルで違いが見られなかったとも報告している．

Ossher らは，ファイルクローン検出手法を提案し，ファイルクローンが発生する状況を調査している [6]．Ossher らの手法は，exact，FQN，fingerprint の 3 つの要素を組み合わせてファイルクローン検出を行う．exact では，各ソースファイルを 1 つの文字列とみなし，その文字列をハッシュ値に変換して一致するかどうかを調査する．FQN では，クラスの完全限定名が等しいかを調査する．fingerprint では，ソースコード中のフィールド名とメソッド名がどの程度等しいかを調査する．Ossher らは，約 1 万 3 千の Java ソフトウェアに対し Ossher らの手法を適用し，上記 3 つの手法の結果を結合させたところ，全ファイルの約 10%超がコードクローンとして検出されたと報告している．また，ファイルクローンが発生する状況として，同じライブラリを使用していることや新たなソフトウェア開発を始めるために以前のソフトウェアを再利用することなどが挙げられるとも報告している．

3.2 Motivating Example

図 1 は，ソフトウェア間にまたがるコードクローンの例である．2 つのクラスはそれぞれ別のソフトウェアに存在するが，いずれもインタフェース Map を継承している．そのため，putAll メソッドを実装した親クラスを作成することでライブラリ化す

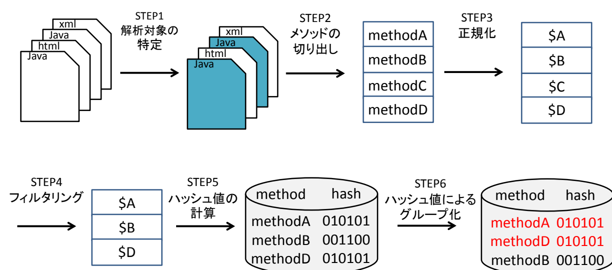

図 2 提案手法の概要

ることができる．この例では，ファイルの一部分の処理のみが，多数のソフトウェア間で共有されているライブラリ化可能な処理に該当する．しかしながら，ファイルクローン検出手法ではこのようなコード片をコードクローンとして検出できない．本研究では，この課題点を解決するためにより細かい粒度であるメソッド単位のコードクローン検出手法を提案する．

3.3 Research Question

本研究ではメソッドクローンに関する以下の問題を調査する．

RQ1：ファイル単位で検出できなかったコードクローンがメソッド単位で検出を行うことで新たにどの程度検出できるか

RQ2：メソッド単位で検出を行うことで新たに検出されたコードクローンの発生原因にはどのようなものがあるか

RQ3：メソッド単位で検出を行うことで新たに検出されたコードクローンは流用の特定などに有用であるか

4. 調査方法

この章では，大規模なソフトウェア群からメソッドクローンを検出する調査方法について述べる．今回の調査では，ファイルクローンとの比較を行うため，メソッドクローンとファイルクローンの両者を検出する．

4.1 概要

対象とするデータセットからその中に存在するメソッドクローンとファイルクローンを検出する．図 2 に検出処理の概要を示す．以降，図 2 に示す各処理について詳細に述べる．

STEP1：解析対象の特定

入力として与えられたデータセットから，解析対象となるソースファイルを特定する．今回の調査では，Java を用いて記述されたソフトウェアのみを解析対象としているため，入力として与えられたデータセットから拡張子が.java であるファイルを特定する．

STEP2：メソッドの切り出し

STEP1 で得られたソースファイルから，メソッドを切り出す．今回の調査では，ソースファイルから抽象構文木を作成してメソッドの切り出しを行う．

STEP3：正規化

このステップでは，コメント文の有無や空白・改行回数などの違いを取り除くために正規化を行う．今回の調査では，以下に示す正規化処理を行う．

- 変数名，文字列リテラル，メソッド・クラス・インターフェース宣言部のメソッド名は 1 つの特殊な文字列に置換する
- 空白，改行，修飾子，アノテーション，コメント文，インポート文，パッケージ文は削除する

STEP4：フィルタリング

ゲッタメソッドやセッタメソッドのように，処理が単純でありかつ短いメソッドは多くのソースファイルに存在すると考えられる．そのため，このようなメソッドは大量にコードクローンとして検出されるおそれがある．このようなコードクローンは，流用の特定や複数のソフトウェアに存在する共通処理のライブラリ化に対して有用ではない．したがって，処理の単純かつ短いメソッドは検出が不要である．このステップでは，このようなメソッドを検出の対象から除外する．今回の調査では，メソッド内の複文の数が 0 であるメソッドについては検出の対象から除外する．ここで複文とは複数の文から成り立つブロックを指し，do，for，if，switch，try，while 文を複文と定義する．

STEP5：ハッシュ値の計算

解析対象となるメソッドそれぞれに対して，ハッシュ値を算出する．今回の調査では，*MD5* [7] によりハッシュ値を計算する．等しい記述を持つメソッドはハッシュ値が等しくなるため，ハッシュ値が等しいメソッドはコードクローンと考えられる．算出されたハッシュ値は，メソッドごとデータベースに格納する．

STEP6：ハッシュ値によるグループ化

ハッシュ値の等しいメソッドをグループ化する．それらのメソッドグループのうち，要素数が 2 以上のグループがメソッドクローンとして検出される．

5. 実験

5.1 実験対象

本研究では，実験対象として”*UCI source code data sets*” [6] [8](以降，*UCIdatasets* と呼ぶ) を用いた．しかし，*UCIdatasets* には trunk,tags,branches を同時に含むソフトウェアやバージョンが違う同一ソフトウェアが複数含まれている．ソフトウェアは処理の追加，修正，削除を行い，新しいバージョンに進化していく．しかし，処理の修正などが行われないソースコードは，バージョンが新しくなってもその内容が変更されないため，コードクローンとして検出される．したがって，このようなソフトウェアに対してコードクローン検出を行うと，検出されるコードクローンが不必要に多くなるおそれがある．そこで本研究では，trunk,tags,branches を同時に含むソフトウェアは trunk 以下のファイルのみを，バージョンの違う同一ソフトウェアは最新バージョンのみを検出対象とした．また，いくつかのソフトウェアにはソースコード自動生成ツールによって作成されたソースコードが存在した．このようなソースコードはコードクローンとして検出されるが，検出されたコードクローンは流用の特定や複数のソフトウェアに存在する共通処理のライブラリ化に対して有用ではない．そのため本研究では，ソースコード自動生成ツールによって作成されたソースコードは検出の対象から除外した．さらに，*UCIdatasets* から今回解析対象とする拡張子が.java であるファイル以外を削除している．こ

のような処理を行うことで，解析対象以外のファイルに対して解析対象であるかを判定する必要がなくなり，検出速度が高速化する．

表 1 は，*UCIdatasets* の構成を表したものである．上記処理の結果，検出対象ファイル数が全ファイル数の約半数に減少している．

5.2 定量的な結果

UCIdatasets に対して，実装したツールを適用してコードクローンを検出した．メソッドクローン数は 2,937,047，メソッドクローンの種類数は 814,391 となった．814,391 種類のうち，約 60%にあたる 490,206 が複数のソフトウェアにまたがっていた．

表 2，3 は，それぞれメソッドクローンとコードクローンを含むファイルの分析結果である．また表中の百分率は，それぞれ検出対象メソッド数，検出対象ファイル数に対する割合になっている．表 2 より，ファイルクローンでないファイルに存在するメソッドクローンは，検出された全メソッドクローン 2,937,047 の 40%を占めることがわかる．また表 3 より，ファイルクローンではないがメソッドクローンを含むファイルは，コードクローンを含む全ファイル 1,079,789 の 27%を占めることがわかる．

5.3 発 生 原 因

検出したメソッドクローンにどのようなものが存在するかを分析するために，またがっているソフトウェア数上位 100 種類のメソッドクローンを調査した．調査の結果を表 4 に示す．表 4 より，上位 100 種類のメソッドクローンのうち 40%が GUI 関連の処理を行うメソッドであることがわかる．また，GUI 関連の処理をするメソッドのうち約半数は，SwingWorker などの抽象クラス内もしくは AbstractTableModel などの抽象クラスを継承したクラス内で宣言されているメソッドであることがわかった．AbstractTableModel は Table 関連の処理を行うクラスであり，SwingWorker は GUI における時間のかかる処理を別のスレッドで実行させるためのクラスである．

また，他のソフトウェアとコードクローンを共有しているファイルにどのようなものがあるかを分析するため，コードクローンを共有しているソフトウェア数上位 500 ファイルを調査した．調査の結果，約 330 ファイルが SwingWorker クラス，約 150 ファイルが ResourceBundle を使用するクラス，残り 20 ファイルほどが GUI 関連のメソッドクローンを内包しているファイルであった．ResourceBundle はプロパティファイルに記述されているデータを読み込むために使用される．

実験の結果から，メソッドクローンの発生原因として以下の要因が挙げられる．

抽象クラスの継承，インターフェースの実装

抽象クラスを継承するクラスやインターフェースを実装するクラスでは，全ての抽象メソッドをオーバーライドしなければならない．またこのようなクラスでは，新たに処理を記述する際に別々のソフトウェアであっても処理が類似することがある．その結果，多くのソフトウェアにまたがるコードクローンが発生すると考えられる．

ソースコードの流用

ソフトウェア開発では，web 上などで公開されているソースコードを開発中のソフトウェアに使用することがある．このとき，自分の作成したソースコードに対応させるといった理由で，使用するソースコードに変更を加えることが考えられる．その結果，ソースコードの一部のメソッドがコードクローンとして検出される．

汎用的な処理を行うメソッド

size や close といったメソッドは多くのクラスで定義されている．このようなメソッドは，特定の変数が null や 0 であるといった条件判定と併用されやすい．そのため，4. で述べたフィルタリング処理を通過しコードクローンとして検出される．

5.4 解 析 速 度

FCFinder [5] では，1CPU,1core(2.50GHz)，メモリ 4GByte の環境で，11.2GByte のソースコードに対して 17.16 時間でコードクローンの検出を終えている．一方今回の実験では，1CPU,4core(2.00GHz)，メモリ 8GByte の環境で実装したツー

表 1 実験対象の構成

全ファイル数	3,963,896
検出対象ファイル数	2,072,490
ソフトウェア数	13,193
検出対象メソッド数	5,953,165
検出対象ファイルの総行数	361,663,992
全容量	30.6GByte

表 2 検出したメソッドクローンの数

	ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
ファイルクローンの一部	1,407,338(24%)	365,150(6%)	1,772,448(30%)
ファイルクローンでない	658,500(11%)	506,059(9%)	1,164,559(20%)
合計	2,065,838(35%)	871,209(15%)	2,937,047(49%)

表 3 検出したコードクローンを含むファイルの数

		ソフトウェアにまたがる	ソフトウェアにまたがらない	合計
ファイルクローン	検出対象メソッドを含む	288,185(14%)	84,213(4%)	372,398(18%)
	検出対象メソッドを含まない	304,779(15%)	114,412(6%)	419,191(20%)
	合計	592,964(29%)	198,625(10%)	791,589(38%)
ファイルクローンではないがメソッドクローンを含む		147,532(7%)	140,668(7%)	288,200(14%)
合計		740,496(36%)	339,293(16%)	1,079,789(52%)

表 4 またがっているソフトウェア数上位 100 種類のメソッドクローンの分類

メソッドの種類		検出数
GUI 関連	AbstractTableModel	15
	その他 Table 関連	10
	SwingWorker	7
	その他	8
64bit-encorder,decorder		13
FileFilter		7
ResourceBundle		4
その他		36

```
39 private static Converter stringConverter =
new Converter() {
40   public Short convert(Object o) {
41    return parseShort(((String) o));
42   }
43 };

（中略）

49 public Object convertFrom(Object in) {
50  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
51     + in.getClass().getName() + " to: " +
Short.class.getName());
52  return CNV.get(in.getClass()).convert(in);
53 }

（中略）

62   CNV.put(String.class,
63    stringConverter
64   );
```

(a) mvel　ShortCHクラス

```
17 public Object convertFrom(Object in) {
18  if (!CNV.containsKey(in.getClass())) throw
new ConversionException("cannot convert type: "
19     + in.getClass().getName() + " to: " +
Short.class.getName());
20  return CNV.get(in.getClass()).convert(in);
21 }

（中略）

30   CNV.put(String.class,
31    new Converter() {
32     public Short convert(Object o) {
33      return Short.parseShort(((String) o));
34     }
35    }
36   );
```

(b) mvflex　ShortCHクラス

図 3　流用特定の例

```
244 public void n2sort() {
245  for(int i = 0; i < getRowCount(); i++) {
246   for(int j = i+1; j < getRowCount(); j++) {
247    if (compare(indexes[i], indexes[j]) == -1) {
248     swap(i, j);
249    }
250   }
251  }
252 }
```

(a) sun　TableSorterクラス

```
219 public void n2sort() {
220  for (int i = 0; i < getRowCount(); i++) {
221   for (int j = i+1; j < getRowCount(); j++) {
222    if (compare(indexes[i], indexes[j]) == -1) {
223     swap(i, j);
224    }
225   }
226  }
227 }
```

(b) Perham　TableSorterクラス

図 4　ライブラリ化の例

ルを *UCIdatasets* に適用した結果，4.91 時間でコードクローンの検出が完了した．表 5 に各処理に要した時間を示す．このような検出時間でコードクローン検出が完了した理由として，データベースを SSD 上においているためデータベースアクセス速度が高速であることや，*UCIdatasets* から今回解析対象とする拡張子が.java であるファイル以外を削除していることが挙げられる．実行環境やソースコードのサイズが異なるため厳密な比較は困難であるが，十分に実用的な時間で検出が完了したものと考えられる．

6. 議　論

6.1 検出したメソッドクローンの有用性

検出したメソッドクローンが有用であるかを評価するために，

表 5　解析時間

処理内容	時間
対象ファイルの特定	0.46h
メソッドの切り出し 正規化 フィルタリング ハッシュ値の計算	2.98h
ハッシュ値のグループ化	1.47h
合計	4.91h

ファイルクローンとしては検出されず，かつ流用の特定や処理のライブラリ化が実現できそうなメソッドクローンが存在するかを調査した．流用の特定はまたがっているソフトウェア数の下位，処理のライブラリ化はまたがっているソフトウェア数の上位のメソッドクローンを調査した．図 3，4 は実装したツールが検出したメソッドクローンの例である．

まず図 3 について，ハイライトされている部分が検出されたメソッドクローンである．図 3(a) のソースコードはファイルの先頭にライセンスについての記述があったが，図 3(b) のソースコードは記述が存在しなかった．また，図 3(a) のソースコードでは stringConverter という変数を宣言し，put メソッドで stringConverter を呼び出している．一方，図 3(b) のソースコードでは put メソッドの内部で処理を記述している．いずれのソースコードも処理の内容自体は変わらないため，違いは記述方法のみである．以上二点から，図 3 はソースコード流用の一例であると考えられる．

次に図 4 について，2 つのソースコードは Table に関する処理を行うクラスで宣言されているソートを行うメソッドである．swing の JTable におけるソート機能は Java1.6 から実装された機能であるため，それ以前の開発ではソート機能は開発者が自ら実装する必要があった．したがって，図 4 の 2 つのソースコードは実際にライブラリ化された例であり，このようなメ

ソッドクローンを今回の調査で検出することができた.

6.2 Research Question に対する回答

3.3 で述べた RQ に対して得られた考察を述べる.

RQ1 は，ファイル単位の検出手法で検出できなかったコードクローンがメソッド単位で検出を行うことで新たにどの程度検出できるかというものである．今回対象とした *UCIdatasets* ではファイルクローン検出手法に対し，新たに 1,164,559 のメソッドクローンが検出され，それらは全メソッドクローンの約 40%を占めることが確認された．また，ファイルクローンではないがコードクローンを含むファイルが 288,200 存在し，それらがコードクローンを含むファイルの約 27%を占めることが確認された.

RQ2 は，メソッド単位で検出を行うことで新たに検出されたコードクローンの発生原因にはどのようなものがあるかというものである．発生原因として以下のようなものが存在した.

- 同じ抽象クラスを継承する複数のクラスを実装する際に処理の内容が類似する
- ソースコードを流用した後，それぞれのソフトウェアに応じた変更を加える
- サイズを取得する処理や終了処理などの汎用的な処理

RQ3 は，メソッド単位で検出を行うことで新たに検出されたコードクローンは流用の特定などに有用であるかというものである．6.1 で述べたように，メソッド単位のコードクローン検出で新たに検出したコードクローンには，ソースコードの流用を特定することや複数のソフトウェア間に共通する処理をライブラリ化することに有用なものが存在した．また，またがっているソフトウェア数上位 100 種類のメソッドクローンを調査した結果，56 種類のメソッドクローンが流用の特定や処理のライブラリ化に有用であることを確認した.

6.3 結果の妥当性

本研究の結果の妥当性に関して，以下で挙げる点に留意する必要がある.

正規化の種類

今回の調査では，4. で述べた正規化を行っている．しかし，型名や数値リテラルを正規化することなど，異なる方法を用いて正規化を行うことで本研究で得られた結果と異なる結果が導かれる可能性がある.

メソッドのフィルタリング方法

今回の調査では，4. で述べたフィルタリング処理を行っている．しかし，このフィルタリング処理では代入文やメソッド呼び出し文のみで構成されるメソッドは，その大小にかかわらず全て検出対象から除外される．そのため，フィルタリング処理の方法を変えることで本研究で得られた結果と異なる結果が得られる可能性がある.

バージョンの異なる同一ソフトウェアの存在

5.1 で述べたように，*UCIdatasets* にはバージョンが違う同一ソフトウェアが複数含まれている．本研究では，実装したツールを適用する前に可能な限り最新のバージョン以外のソフトウェアを検出対象から除外しているが完全ではなく，バージョンの違う同一ソフトウェアが検出対象に複数含まれているおそれがある．そのため，不必要に多くコードクローンを検出している可能性がある.

7. おわりに

本研究では，メソッド単位とファイル単位の 2 つの手法を用いて大規模なソフトウェア群からコードクローン検出を行った.

実験の結果，検出したメソッド単位のコードクローンのうち約 40%，コードクローンを含むファイルのうち約 27%がメソッド単位の検出手法で新たに検出されたコードクローンであるという結果が得られた．また，ソースコードの流用を特定することや複数のソフトウェア間に共通する処理をライブラリ化することに有用であるコードクローンが存在することが確認された.

本研究の今後の課題は以下のとおりである.

- 本研究とは別の正規化やフィルタリング処理を実装して検出されるコードクローンに違いがあるかを調査する.
- 対象となるファイルを Java 以外にも拡張して，言語によって検出されるコードクローンに違いがあるかを調査する.

謝辞 本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A）（課題番号：21240002）および萌芽研究（課題番号：23650014），文部科学省科学研究費補助金若手研究（B）（課題番号：22700031）の助成を得た.

文　献

[1] S.Livieri, Y.Higo, M.Matushita, and K.Inoue, “Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder,” Proc. of the 29th International Conference on Software Engineering, pp.106–115, 2007.

[2] A.W. Brown and G. Booch, “Reusing open source software and practices: The impact of open source on commercial vendors,” Proc. of the 7th International Conference on Software Reuse, pp.123–136, 2002.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, “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, July 2002.

[4] J. Mayrand, C. Leblanc, and E.M. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” Proc. of the 12th IEEE International Conference on Software Maintenance, pp.244–253, 1996.

[5] 佐々木裕介，山本哲男，早瀬康裕，井上克郎，“大規模ソフトウェアシステムを対象としたファイルクローンの検出,” 電子情報通信学会論文誌 D，vol.J94-D，no.8，pp.1423–1433，2011.

[6] J. Ossher, H. Sajnani, and C. Lopes, “File cloning in open source java projects: The good, the bad, and the ugly,” Proc. of the 27th International Conference on Software Maintenance, pp.283–292, Sep. 2011.

[7] R. Rivest, “The md5 message-digest algorithm,” April 1992. http://www.ietf.org/rfc/rfc1321.txt.

[8] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, “Uci source code data sets”. http://www.ics.uci.edu/~lopes/datasets/.