特別研究報告

題目

大規模データセットを対象とするメソッド単位でのコードクローン検出

指導教員
楠本 真二 教授

報告者
石原　知也

平成 24 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 23 年度 特別研究報告

大規模データセットを対象とするメソッド単位でのコードクローン検出

石原　知也

内容梗概

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と予測されている．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流用されているソースコードの発見，また複数のソフトウェア間に存在する共通処理のライブラリ化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では細粒度で検出を行っているため，大規模なソフトウェア群を対象として検出を行う場合は非常に大きなコストを必要とする．この問題点を解決するために，ファイル単位でコードクローンを検出するという手法が提案されているが，この手法にはファイルの一部がコードクローンになっているものは検出できないという欠点が存在する．

本研究では既存研究の問題点を解決するため，メソッド単位でコードクローンを検出する手法を提案する．メソッド単位でのコードクローン検出は，ファイル単位のコードクローン検出手法と同様に，大規模なソフトウェア群を対象とした場合でも実用的なコストで検出を行うことが可能であり，かつファイル単位の検出手法では検出することができないコードクローンを検出できる．

提案手法を実装し，約 1 万 3 千のソフトウェアから構成される大規模ソフトウェア群に対して適用実験を行った．その結果，約 3 億 6 千万行のソースコードに対して 4.91 時間で検出処理が完了した．また，検出されたコードクローンを分析したところ，検出されたコードクローンの約 40%が，ファイル単位の検出手法ではコードクローンとして検出できないものであるという結果が得られた．さらに検出されたコードクローンを含むファイルの約 27%が，ファイル単位のコードクローン検出手法ではコードクローンを含んでいると認識できないファイルであることが確認された．

主な用語

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

目 次

1　まえがき

ソフトウェア群には，ファイルの流用などによって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．

以降 2 章では，コードクローンの定義と既存の検出手法について述べる．3 章では研究動機について述べる．4 章では提案手法について説明し，5 章で本研究での実装方法について述べる．6 章では大規模なソフトウェア群を対象に実装したツールを適用した結果について述べ，7 章でその考察や妥当性の検証を行う．8 章では関連研究について述べ，最後に 9 章で本研究のまとめと今後の課題について述べる．

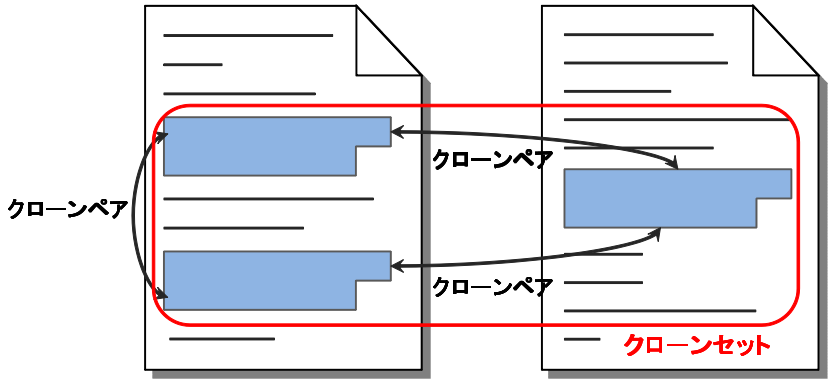


図 1: クローンペアとクローンセット

2 準備

2.1 コードクローン

2.1.1 定義

コードクローンとはソースコード中に存在する同一，あるいは類似するコード片のことである．図 1 に示すように，ソースコード中に存在する 2 つのコード片 α，β が類似しているとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるコード片も類似していないとき，α，β を極大クローンと呼ぶ．また，互いにクローンであるコード片を同値としたときの同値類をクローンセットと呼ぶ [7].

ただし，どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる．

また，コードクローン間の類似の度合いに基づきコードクローンを次の 3 つのタイプに分類することができる [8][9].

Type-1

空白やタブの有無，括弧の位置などのコーディングスタイルに依存する箇所を除いて，

完全に一致するコードクローン.

Type-2

変数名や関数名などのユーザ定義名，また変数の型など一部の予約語のみが異なるコードクローン.

Type-3

Type-2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン.

2.1.2 発生の原因

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものが挙げられる [3][10][11].

既存コードのコピーアンドペーストによる再利用

近年のソフトウェア設計手法を利用することにより構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーアンドペーストによる場当たり的な既存コードの再利用が多く行われるようになった．コピーアンドペーストによって生成されたコード片は，コピー元のコード片とコードクローン関係になる.

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである.

定型処理

定義上簡単で頻繁に用いられる処理．例えば，所得税の計算や，キューの挿入処理，データ構造アクセス処理などである.

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合，同じようなアルゴリズムを用いた処理を繰り返し書かなくてはならないことがある.

パフォーマンス改善

リアルタイムシステムなど時間制約のあるソフトウェアにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し記述することによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは，あらかじめ決められたコードをベースにして自動的に生成される．このため，類似した処理を目的としたコードを生成した場合，識別子名などの違いを除き，類似したコードが生成される．

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは，各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.1.3 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 コードクローン検出手法

2.2.1 コードクローン検出手法の分類

コードクローンを検出する手法はこれまでに多数提案されている．またそれらを実装した，コードクローンを自動的に検出するツールも多数開発されている．これらの検出技術はその検出単位によって，大まかに以下の 5 つに分類することができる [12][13].

行単位の検出

行単位の検出は，ソースコードを行単位で比較してコードクローンを検出する手法であり，閾値以上連続して一致する行をコードクローンとして検出する．他の手法と異なり，ソースコードに対する事前処理を必要としない．このため，他の手法と比べて高速にコードクローンを検出可能である．しかし，同じ処理を行っているコードであっても，例えば長い行を複数行に分割した場合と分割しなかった場合など，コーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

字句単位の検出

字句単位の検出は，ソースコードを字句単位に分割し，閾値以上連続して一致する字句の部分列をコードクローンとして検出する手法である．行単位の検出と異なり，コーディングスタイルのみ違う場合などもコードクローンとして検出することが可能である．ソースコードを検出用の中間表現に変換する必要がないため，高速にコードクローン検出を行うことができるという利点もある．また，字句に事前処理を行うことで変数名などのユーザ定義名のみ異なるコードクローンなども検出可能となる．

抽象構文木を用いた検出

抽象構文木 (図 2) を用いた検出は，ソースコードに対して構文解析を行い，抽象構文木を構築した後，その抽象構文木を用いてコードクローンを検出する手法であり，抽象構文木上の同形の部分木がコードクローンとして検出される．抽象構文木を構築するという事前処理を要するため，行単位の検出や字句単位の検出と比べ，時間的，空間的コストが高くなるという欠点がある．ある関数定義の終わりから次の関数定義の先頭までを含むようなプログラムの構造を無視したコードクローンを検出しないという特徴を持つ．

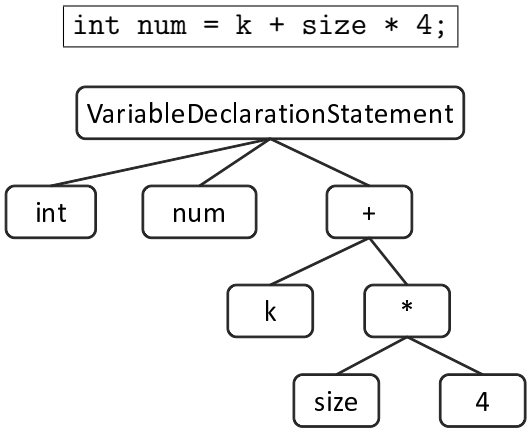


図 2: 抽象構文木の例

プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3)[14] を用いた検出は，ソースコードに対して意味解析を行い，ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後，そのプログラム依存グラフを用いてコードクローン検出を行う手法である．プログラム依存グラフ上の同形部分グラフがコードクローンとして検出される．抽象構文木を用いた検出と同様に事前処理を必要とするため，時間的，空間的コストが高くなるという欠点を持つ．ソースコードの順番が入れ替わっていても意味的に同一であるコードクローン (順序入れ替わりコードクローン) などは意味的な処理を考慮しなければ検出できないが，この手法はこれらのコードクローンを検出することができるという点が特徴として挙げられる．

順序入れ替わりコードクローンの例を図 4 に示す．この例の場合，%で表されているコード片と，#で表されているコード片が順序入れ替わりコードクローンとなる．

その他の技術を用いた検出

その他の技術を用いた検出手法として，プログラムのモジュール (ファイル，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法や，プログラムの盗用の検出やプログラムの作者を特定することを目的とした，フィンガープリントやバースマークを用いた検出手法などがある．

```
1: void sample() {
2:   for ( int i = 0 ; i < 10 ; i++ ) {
3:     System.out.println(i);
4:   }
5: }
```

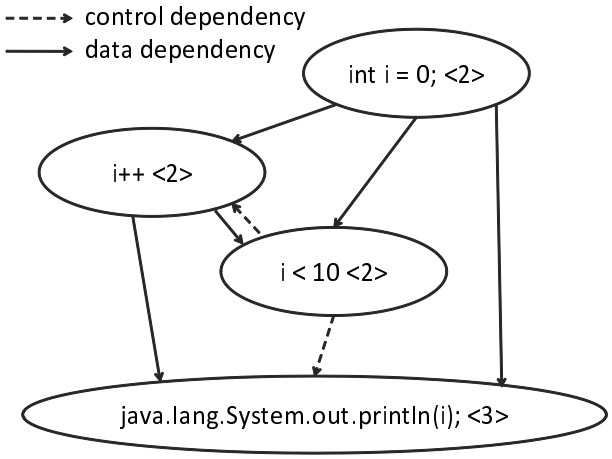


図 3: プログラム依存グラフの例

```
  fp = lookaheadset + tokensetsize;
  for (l = lookaheas(state) ; l < k ; i++) {
%     fp1 = LA + i * tokensetsize;
%     fp2 = lookaheadset;
%     while (fp2 < fp3)
%           *fp2++ |= fp1++;
  }
```

(a) コード片 1

```
  fp3 = base + tokensetsize;
  ...
  if (rp) {
      while ((j = *rp++) >= 0) {
          ...
#             fp1 = lookaheadset;
#             fp2 = LA + j * tokensetsize;
#             while (fp1 < fp3)
#                   *fp1++ |= *fp2++;
      }
  }
```

(b) コード片 2

図 4: 順序入れ替わりコードクローン

2.2.2 ファイル単位のコードクローン

2.2.1 で述べた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，2.2.1 で述べたコードクローン検出手法は高精度で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コード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 5 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．

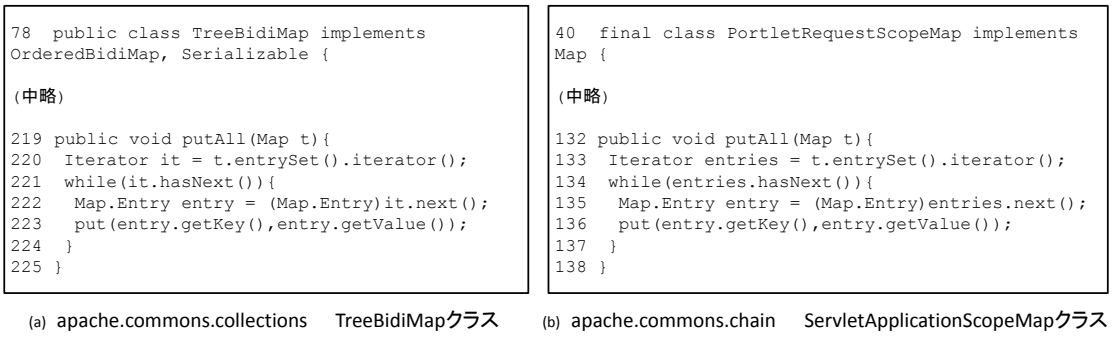


図 5: Motivating Example

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

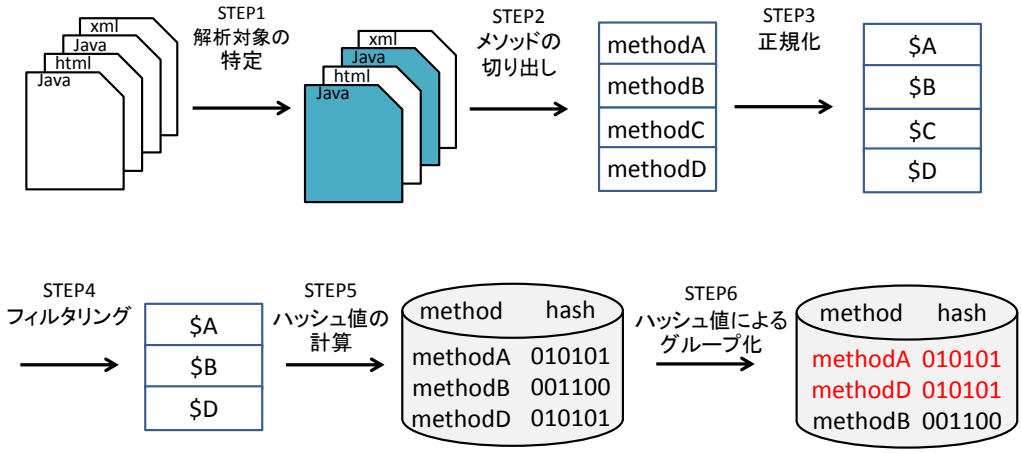


図 6: 提案手法の概要

4 提案手法

この章では，提案するメソッドクローン検出手法について説明する．

4.1 手法の概要

提案手法は，対象とするデータセットを入力として受け取り，それらの中に存在するメソッドクローンを検出する．図 6 に提案手法の概要を示す．以降，図 6 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．

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．

STEP3：**正規化**

コメント文の有無や空白・改行回数の違いを取り除くために正規化を行う．このステップでは，メソッド内のコメント文を削除し，空白・改行は全て 1 つの形式で統一する．図 7 は空白や改行を 1 つの形式に統一した例である．

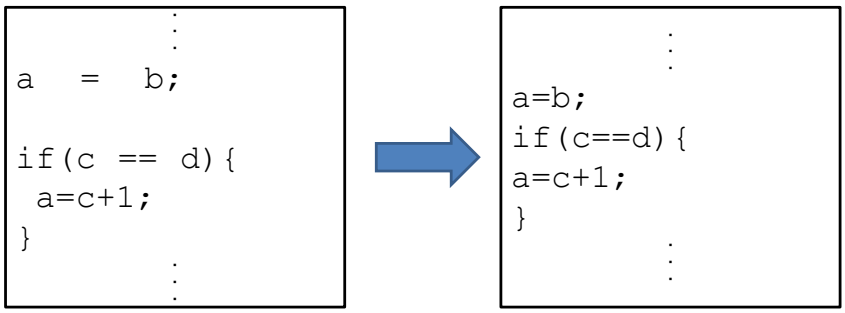


図 7: 空白や改行についての正規化例

STEP4：**フィルタリング**

getter メソッドや setter 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．

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．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．

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 以上のグループがクローンセットとして検出される．

4.2 メトリクス

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などに有用なメソッドクローンを特定しやすくするために，ファイルやソフトウェア及びクローンセットに以下のようにメトリクスを定義する．

- クローンセット

- StrideFiles：またがっているファイル数
- StrideProjects：またがっているソフトウェア数

- ファイル
  - CloneMethodRate : ファイル内で宣言されている全メソッドのうちコードクローンであるメソッドの割合
  - SharedFiles : ファイル内に存在するメソッドクローンを共有しているファイル数
  - SharedProjects$_f$：ファイル内に存在するメソッドクローンを共有しているソフトウェア数

- ソフトウェア
  - CloneFileRate：ソフトウェア内にある全てのファイルのうちコードクローンを含むファイルの割合
  - SharedProjects$_p$: ソフトウェア内に存在するメソッドクローンを共有しているソフトウェア数

流用しているもしくはされていると考えられるファイルやソフトウェアは，その大部分がコードクローンになっていると考えられる．そのため，このようなファイルやソフトウェアは CloneMethodRate,CloneFileRate メトリクスが大きくなる．また，流用が要因で生成されたコードクローンは多くのソフトウェアにはまたがらないと考えられる．そのため，このようなコードクローンは StrideFiles や StrideProjects メトリクスが小さくなる．

ライブラリ化すべきメソッドクローンは多くのソフトウェアやファイルにまたがっていると考えられる．そのため，このようなコードクローンは StrideFiles や StrideProjects メトリクスが大きくなる．また，ライブラリ化すべきメソッドクローンを含むファイルやソフトウェアは SharedFiles,SharedProjects$_f$, SharedProjects$_p$ メトリクスが大きくなる．

5　実装

4 章で述べた提案手法を実装し，メソッドクローンを検出するツールを開発した．以降，図 6 に示した各処理の実装方法について述べる．

5.1　解析対象の特定

実装したツールは，現在のところ 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．そのため，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．

5.2　メソッドの切り出し

実装したツールでは，ソースファイルから抽象構文木を作成する．抽象構文木の作成には JavaDevelopmentTools[15](以降 JDT と呼ぶ) を使用している．JDT では，メソッドの内容はメソッド宣言ノード以下に作られるため，メソッド宣言ノード以下の部分木を取り出すことによってメソッドの切り出しを実現している．

5.3　正規化

4.1 で述べた正規化処理に加えて，実装したツールでは以下に記述した項目に関する違いをソースコードから取り除く．

- 変数名
- 文字列リテラル
- メソッド宣言部のメソッド名
- 修飾子
- アノテーション

図 8 は抽象構文木上での正規化の例である．メソッド内に存在する変数名や文字列リテラル，及びメソッド宣言部のメソッド名は全て 1 つの特殊な文字列に置換される．修飾子やアノテーションは削除する．

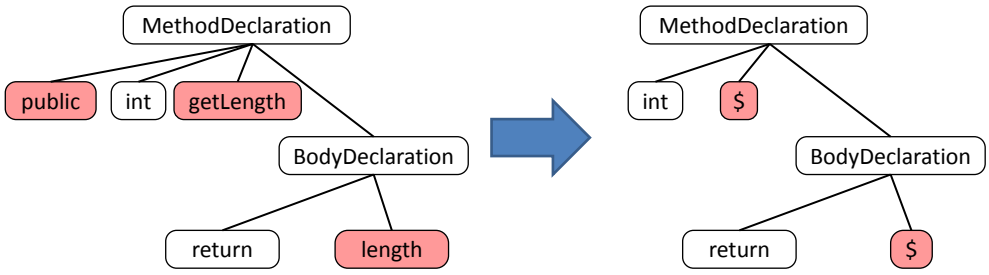


図 8: 抽象構文木上での正規化の例

5.4 フィルタリング

4.1 で述べたメソッドを対象から除外するために，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した．ここで複文とは，複数の文から成り立つブロックを指す．実装したツールでは，以下に挙げる文を複文と定義する．

- do 文
- for 文
- if 文
- switch 文
- try 文
- while 文
- synchronized 文

図 9 は検出対象メソッドの例である．このメソッドは if 文を含むため検出対象メソッドとなる．一方，図 10 は検出対象でないメソッドの例である．このメソッドは上記で定義されている複文を含まないため検出対象から除外される．

5.5 ハッシュ値の計算

実装したツールでは，*MD5*[16] によりハッシュ値を計算する．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．

```
public void step() {
    index++;
    if (index<length){
    next = (NamespaceImpl)nslist.get(index);
    } else {
        next = null;
    }
}
```

図 9: 検出対象メソッドの例

```
String getString() {
   return this.string;
}
```

図 10: 検出対象でないメソッドの例

5.6 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した後，実装したツールでは 4.2 で定義したメトリクスを算出する．

6 実験

6.1 実験対象

本研究では，実験対象として*"UCI source code data sets"*[6][17](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．

6.2 実験のためのツールの拡張

本研究では，ファイルクローン検出手法では検出できないコードクローンの有用性を評価するため，実装したツールにファイルクローンを検出する処理を追加した．具体的には，

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

解析対象となるファイルそれぞれに対して，その文字列をもとに MD5 によりハッシュ値を算出し，データベースに格納する．解析対象のファイルには，5.3 で述べた正規化に加えて，以下の正規化を行っている．

- パッケージ文の削除
- インポート文の削除
- クラス，インターフェース宣言部の名前定義部分を特殊な文字列に置換

6.3 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．メソッドクローンとして検出されたメソッド数は 2,937,047，クローンセット数は 814,391 となった．クローンセット数 814,391 のうち，約 60%にあたる 490,206 のクローンセットが複数のソフトウェアにまたがっていた．

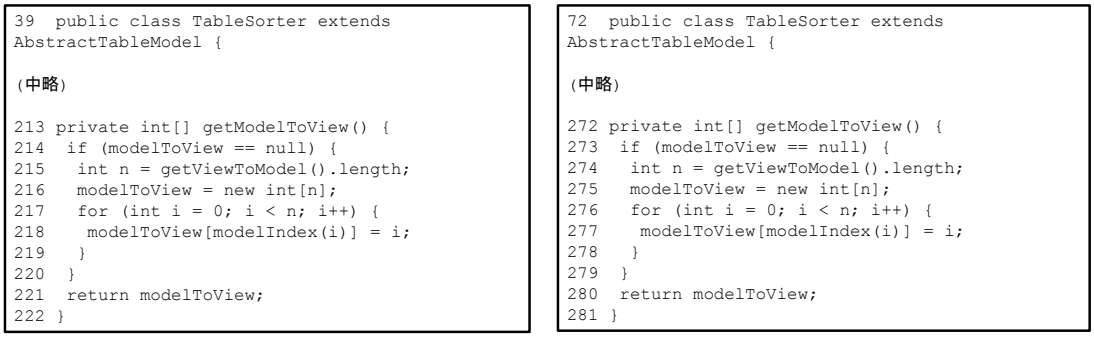
表 2，3 は，それぞれメソッドクローンとコードクローンを含むファイルの分析結果である．また表中の百分率は，それぞれ検出対象メソッド数，検出対象ファイル数に対する割合になっている．表 2 より，ファイルクローンでないファイルに存在するメソッドクローンは，検出された全メソッドクローン 2,937,047 の 40%を占めることがわかる．また表 3 より，ファイルクローンではないがメソッドクローンを含むファイルは，コードクローンを含む全ファイル 1,079,789 の 27%を占めることがわかる．

6.4 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，クローンセットの StrideProjects メトリクス上位 100 のクローンセットを調査した．調査の結果を表 4 に示す．表 4 より，上位 100 クローンセットのうち 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

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%)



```
39  public class TableSorter extends
AbstractTableModel {

（中略）

213 private int[] getModelToView() {
214  if (modelToView == null) {
215   int n = getViewToModel().length;
216   modelToView = new int[n];
217   for (int i = 0; i < n; i++) {
218    modelToView[modelIndex(i)] = i;
219   }
220  }
221  return modelToView;
222 }
```

(a) Jug-avis TableSorterクラス

```
72  public class TableSorter extends
AbstractTableModel {

（中略）

272 private int[] getModelToView() {
273  if (modelToView == null) {
274   int n = getViewToModel().length;
275   modelToView = new int[n];
276   for (int i = 0; i < n; i++) {
277    modelToView[modelIndex(i)] = i;
278   }
279  }
280  return modelToView;
281 }
```

(b) jPodder TableSorterクラス

図 11: AbstractTableModel を継承したクラス内のメソッドクローン

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．図 11，12 はそれぞれ AbstractTableModel を継承したクラス内と SwingWorker クラス内のメソッドクローンの例である．

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，ファイルの $\mathrm{SharedProjects}_f$ メトリクス上位 500 の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 関連のクローンセット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．図 13 は ResourceBundle を使用するクラス内のメソッドクローンの例である．

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．

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: StrideProjects メトリクス上位 100 クローンセットの分類

メソッドの種類		クローンセット数
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

```
20 public abstract class SwingWorker
21 {

（中略）

91 public void interrupt() {
92  Thread t = threadVar.get();
93  if (t != null) {
94   t.interrupt();
95  }
96  threadVar.clear();
97 }
```

(a) Xoetrope　SwingWorkerクラス

```
26 public abstract class SwingWorker {

（中略）

83 public void interrupt() {
84  Thread t = threadVar.get();
85  if (t != null) {
86   t.interrupt();
87  }
88  threadVar.clear();
89 }
```

(b) Wonderly　SwingWorkerクラス

図 12: SwingWorker クラス内のメソッドクローン

```
78 public static String getString(String key) {
79  try {
80   return RESOURCE_BUNDLE.getString(key);
81  }
82  catch (MissingResourceException e) {
83   return '!' + key + '!';
84  }
85 }
```

(a) Classpath Helper　PreferenceConstantsクラス

```
22 public static String getString(String key) {
23  try {
24   return RESOURCE_BUNDLE.getString(key);
25  } catch (MissingResourceException e) {
26   return '!' + key + '!';
27  }
28 }
```

(b) Hudson　Messagesクラス

図 13: ResourceBundle を使用するクラス内のメソッドクローン

```
33 public class ChordSheetFileFilter extends FileFilter{

（中略）

77 protected static String getExtension(File f) {
78  String ext = null;
79  String s = f.getName();
80  int i = s.lastIndexOf('.');
81  if (i > 0 && i < s.length() - 1) {
82   ext = s.substring(i + 1).toLowerCase();
83  }
84  return ext;
85 }
```

(a) ChordCast　ChordSheetFileFilterクラス

```
17 public class XodeFileFilter extends FileFilter {

（中略）

43 private String getExtension(File f) {
44  String ext = null;
45  String s = f.getName();
46  int i = s.lastIndexOf('.');
47
48  if (i > 0 &&  i < s.length() - 1) {
49   ext = s.substring(i+1).toLowerCase();
50  }
51  return ext;
52 }
```

(b) NetBeans　XodeFileFilterクラス

図 14: FileFilter を継承したクラス内のメソッドクローン

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．今回の実験では，AbstractTableModel や FileFilter などの抽象クラスを継承したクラス内のメソッドがコードクローンとして多く検出された．図 14 は FileFilter を継承したクラス内のメソッドクローンの例である．

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．今回の実験結果では，64bit-encorder,decorder が該当する．

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，5.4 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．図 15 は size メソッドがメソッドクローンとして検出された例である．

6.5　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，

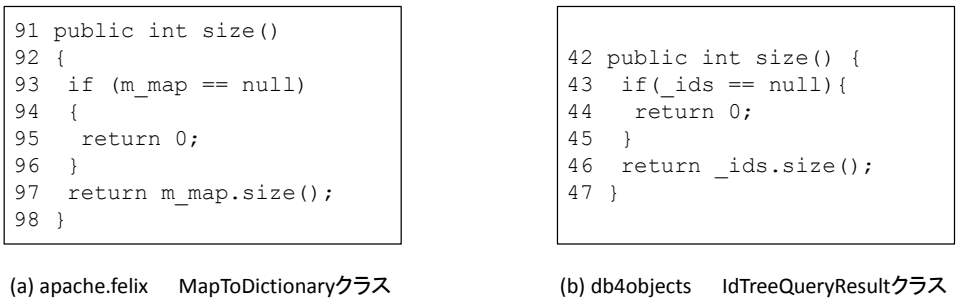


図 15: size メソッドのメソッドクローン

1CPU,4core(2.00GHz)，メモリ 8GByte の環境で実装したツールを *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．

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	2.98h
正規化	
フィルタリング	
ハッシュ値の計算	
ハッシュ値のグループ化	1.47h
合計	4.91h

7　議論

7.1　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，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 StrideProjects メトリクスの下位を，処理のライブラリ化は StrideProjects メトリクスの上位を調査した．図 16，17 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．どちらのコードクローンもファイルの一部分がコードクローンとなっているため，ファイルクローン検出手法では検出できない．

まず図 16 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 16(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 16(b) のソースコードは記述が存在しなかった．また，図 16(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 16(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 16 はソースコード流用の一例であると考えられる．

次に図 17 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 17 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，このようなコードクローンを今回の調査で検出することができた．

7.2　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

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．

```
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 16: 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 17: ライブラリ化の例

RQ2

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

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．7.1 で述べたように，メソッド単位の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，クローンセットの StrideProjects メトリクス上位 100 のクローンセットを調査した結果，56 のクローンセット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．

7.3 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．

ハッシュ値の衝突

提案手法では 2 つのメソッドがコードクローンであるかどうかの判定にハッシュ値を使用している．コードクローン検出の際にハッシュ値の衝突が起これば，本来コードクローンではないメソッドがコードクローンとして検出されてしまうおそれがある．しかし，本研究では対象となるメソッド数が約 600 万であり，かつ本研究では 128bit の MD5 アルゴリズムを使用しているため衝突確率は十分に低いものと考えられる [18]．本研究では，実際にハッシュ値が衝突しているかどうかを検出した全てのコードクローンに対して調査し，ハッシュ値の衝突がなかったことを確認した．

正規化の種類

実装したツールでは，5.3 や 6.2 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．正規化処理を少なくした場合，変数名などが違うものはコードクローンとして検出されなくなるため，検出されるコードクローン数が少なくなると予測される．正規化処理を増やした場合，コードクローン検出の際，型名などの違いを考慮しなくなるためコードクローン数が増えると予測される．

メソッドのフィルタリング方法

実装したツールでは，5.4 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

6.1 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．

解析対象とするソフトウェア群の性質

本研究では，解析対象として *UCIdatasets* を用いている．*UCIdatasets* は，web 上で公開されているオープンソースソフトウェアシステムを収集したものである．しかし，全てのオープンソースソフトウェアシステムを網羅しているわけではないため，本研究で得られた結果が一般的なオープンソースソフトウェアシステムの特徴を示していない可能性がある．また，商用ソフトウェアに対しては実験を行っていないため，本研究で得られた結果と異なる可能性がある．

UCIdatasets は Java で記述されているソースコードのみを収集しているため，他の言語で記述されているソフトウェア群に対して提案手法を適用した場合，本研究で得られた結果と異なる可能性がある．

8 関連研究

田中らは，ソースコードの再利用を効率的に行うために，コードクローン検出技術を用いて複数のソフトウェアから同様の機能を持つ関数集合群を検出する手法を提案している [19]. 田中らの手法では，対象となるソースコード群を複数のタスクに分割し，複数のマシンで並列に関数群の検出を行っている．数千万行のソースコードに対して実験を行い，多くの類似している関数群を検出することができたと報告している．

Livieri らは，超大規模ソースコード集合からコードクローンを検出する D-CCFinder を提案した [1][20]．D-CCFinder は 80 台のコンピュータを用いて分散処理を行い，コードクローンを検出する．約 4 億行のソースコードから 2 日余りでクローンペアを検出することができたと報告している．

本研究では，大規模ソフトウェア群に対しメソッド単位でのコードクローンの検出を行っている．メソッドをハッシュ値に変換しているため，高速にコードクローンの検出を行うことができる．約 3 億 6 千万行のソースコードから，4.91 時間でコードクローンを検出することができた．

Al-Ekram らは，オープンソースソフトウェア間に存在するコードクローンに対し，それらがコピーアンドペーストにより生じたものか，あるいは偶然生じたものであるか調査を行った [21]．テキストエディタ，ウィンドウマネージャの二種類のドメインのオープンソースソフトウェアに対して，同一ドメインのソフトウェア間でコードクローンを検出し，調査を行った．その結果，偶然発生した多くのクローンは API やライブラリの利用に対応したために生じると報告している．

本研究では，オープンソースソフトウェアを集めた *UCIdatasets* に対してコードクローン検出を行っている．その結果，メソッドクローンの発生原因として抽象クラスの継承，ソースコードの流用，汎用的な処理を行うメソッドが挙げられる．

Liu らは，ソフトウェア間で PDG の類似したソースコードを発見することにより，ソースコードの盗用を検出する手法を提案している [22]．PDG は文の順序変更や制御文などの違いを吸収するため，盗用後のソースコードにある程度変更が加えられたとしても盗用されたコードの検出が可能である．

Brixtel らは，学生のコーディング課題における剽窃を検出することにコードクローン検出手法を適用した [23]．数百ほどのソースコードに適用した結果，盗用したと思われるコードが検出できたと報告している．

Liu らや Brixtel らは比較的小規模なソースコード群に対してソースコードの盗用検出を行っているが，本研究では大規模なソフトウェア群に対してコードクローンの検出を行い，ソースコード流用の特定を支援している．

9 あとがき

本研究では，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コードクローンを検出するため，メソッド単位でのコードクローン検出手法を提案した．また，その手法の有効性を調べるために提案手法をツールとして実装し，大規模なソフトウェア群に対して実験を行った．

実験の結果，検出したメソッドクローンのうち約 40%，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，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．ただし他言語で記述されているソースコードを対象とする場合，メソッドの切り出し処理を工夫する必要性がある．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究において，多大なるご助言を頂きました 井垣 宏 特任准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究に用いた分析ツールの基礎部分の設計をしてくださり，また本研究に関して多大なるご助言，ご助力を頂きました，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博士前期課程 2 年の 堀田 圭佑 氏に深く感謝申し上げます.

その他の楠本研究室の皆様からいただいたご助言やご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[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. In *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. In *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. In *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. In *Proc. of the 27th International Conference on Software Maintenance*, pp. 283–292, Sep. 2011.

[7] 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.

[8] S. Bellon. Detection of software clones. *Technical Report, Institute for Software Technology, University of Stuttugart*, 2003. available at `http://www.bauhaus-stuttgart.de/clones/`.

[9] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 804–818, Oct. 2007.

[10] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the 14th International Conference on Software Maintenance*, pp. 368–377, Mar. 1998.

[11] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, Apr. 2005.

[12] 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.

[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. 28–42, Aug. 2011.

[14] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線, 2009.

[15] Java development tools. `http://www.eclipse.org/jdt/`.

[16] R. Rivest. The md5 message-digest algorithm. *RFC 1321(Informational)*, Apr. 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

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

[18] B. Hummel, E. Jürgens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proc. of the 26th International Conference on Software Maintenance*, pp. 1–9, 2010.

[19] 田中健介, 肥後芳樹, 楠本真二. 大規模ソースコード集合を対象とした類似関数集合群の抽出. 電子情報通信学会技術研究報告, Vol. 109, No. 456, pp. 49–54, Mar. 2010.

[20] 肥後芳樹, リビエリシモネ, 松下誠, 井上克郎. 大規模ソースコードを対象としたコードクローンの検出と可視化. 情報処理学会論文誌, Vol. 48, No. 11, pp. 3510–3519, Nov. 2007.

[21] R.Al-Ekram, C.Kapser, R.Holt, and M.Godfre. Cloning by accident: an empirical-study of source code cloning across software systems. In *Proc. of the 4th International Symposium on Empirical Software Engineering*, 2005.

[22] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 872–881, 2006.

[23] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Proc. of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 77–86, 2010.