修士学位論文

題目

異常処理除去を追加したメソッドクローン検出手法の提案と評価

指導教員
楠本 真二 教授

報告者
長瀬 義大

平成 25 年 2 月 5 日

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

平成 24 年度 修士学位論文

異常処理除去を追加したメソッドクローン検出手法の提案と評価

長瀬 義大

内容梗概

ソフトウェア開発において，多くのソフトウェアで頻繁に利用される機能をまとめたライブラリの利用は，ソフトウェアの信頼性や開発効率の向上に有用である．そのようなライブラリに含めるべき機能を特定するためには，大規模なプロジェクト群に存在するコードクローンを検出することが有益であると考えられる．そのため，このようなコードクローンを高速に細粒度で検出することを目的としたメソッド単位の検出手法が提案されている．しかし既存手法では，プログラムの正常処理が同じメソッドであっても異常処理が異なるためにコードクローンとして検出されない場合がある．異常処理とはプログラムの耐故障性の向上に利用される処理であり，正常処理には影響を与えない．そのため，ライブラリ化候補を検出する際には，異常処理が異なっていても 1 つの候補として検出する必要がある．そこで，異常処理に関するコードを除去したうえでメソッド単位のコードクローン検出を行う手法を提案する．この手法により，メソッドの正常処理が同じであれば，たとえその中に存在している異常処理が異なっていても 1 つのライブラリ化候補として検出することが可能になる．

提案手法を実装し，約 13,000 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，データセットに存在する約 360,000,000 行のソースコードに対して 2 時間以内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，82,639 のメソッドを新たにコードクローンとして検出できているという結果が得られた．

主な用語

コードクローン
ライブラリ作成
大規模データセット
ソフトウェア保守

目 次

1　はじめに

ライブラリとは再利用可能なソフトウェア部品の集合 [1] であり，ソフトウェア開発においてライブラリの利用は信頼性や開発効率の向上に有用である．これは，開発したいプログラムの機能がすでにライブラリに含まれている場合，開発者がその機能を実装する必要がないためである．

開発者がこのようなライブラリの恩恵を受けるためには，必要とする機能がライブラリに含まれている必要がある．そのためには，頻繁に利用される機能をライブラリ化しておく必要がある．ライブラリ化すべき機能を特定するためにコードクローン検出手法は有益であると考えられる．

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．ソースコードの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示されている [2]．現在，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高速に検出することを目的とした手法として，ファイル単位のコードクローン検出手法 [3][4]，並びにスケーラブルな細粒度コードクローン検出手法 [2][5][6][7][8][9] が提案されている．

ファイル単位のコードクローン（以降ファイルクローンと呼ぶ）検出手法 [3][4] では，ソースコードをファイル単位で比較することによりファイルクローンの検出を行う．しかしこれらの手法では，ファイル単位でしか比較を行わないため，ファイルの一部のみがコードクローンである場合には検出することができない．スケーラブルな細粒度コードクローン検出手法 [2][5][6][7][8][9] では，大規模なデータセットから細粒度でコードクローン検出を行う．これらの手法はファイルクローン検出手法では検出できない，ファイルの一部のみのコードクローンを検出することが可能であるが，速度面でファイルベースの手法に劣っている．また，細粒度な検出手法では検出単位が細かすぎるため，機能の一部のみを含むコードクローンを多く検出してしまう．そのため，ファイルベースの手法，細粒度な手法ともにライブラリの作成に利用するには不十分である．

これに対し石原らは，複数のソフトウェアからライブラリ化の候補を高速に検出することを目的とした，メソッド単位の検出手法を提案している [10]．この手法では，メソッドを単位として検出を行うため，コードクローン情報から簡単にライブラリ化を行うことができる．また，メソッドからハッシュ値を算出し，そのハッシュ値を用いて検出を行うため，巨大なデータセットに対しても十分高速に実行することが可能である．

しかしこのメソッド単位の手法では，プログラムの目的とする機能が同じであっても異常処理が異なるために，検出されないコードクローンが存在する．プログラムには，目的とする機能を実現するための実装（正常処理）と機能を実行する過程で発生する意図しない状況への対応を目的とした実装（異常処理）の 2 種類が含まれている．そのため，メソッド単位のコードクローン検出手法において異常処理の影響を排除することができれば，正常処理のみに着目したライブラリ化候補を検出することが可能となると考えられる．そこで本研究では，メソッド単位のコードクローン検出に異常処理の除去を追加し，コードクローン検出を行う．この手法により，メソッドの正常処理が同じであれば，たとえその中に存在している異常処理が異なっていても 1 つのライブラリ化候補として検出する

ことが可能になる.

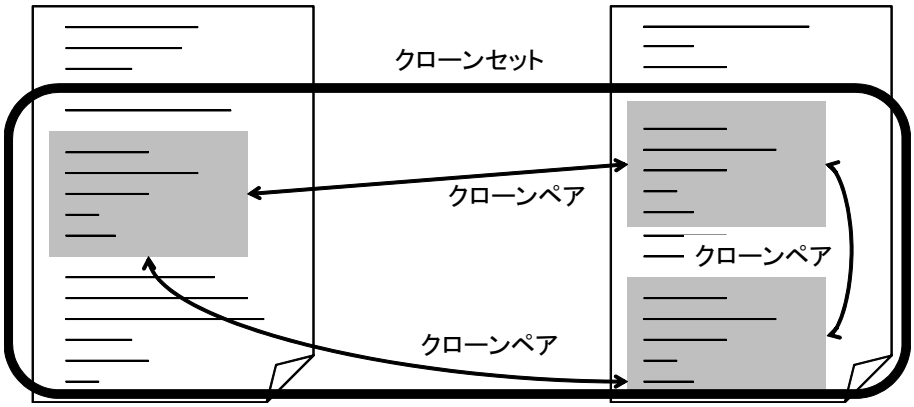


図 1: クローンペアとクローンセット

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 [11].

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 [12][13].

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 [14][15][16].

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 [3][4]．

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．

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．

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

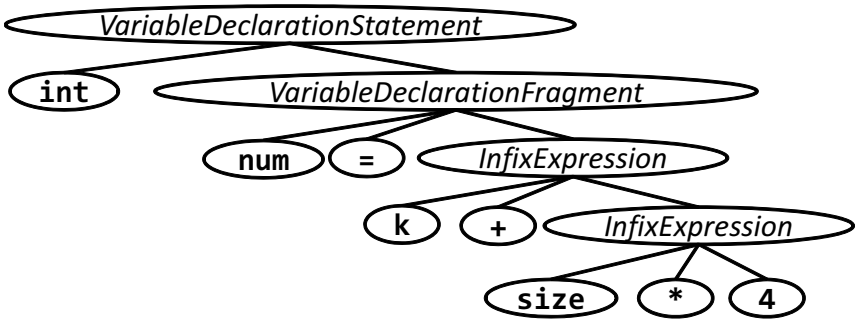
コードクローンを検出する手法はこれまでに多数提案されている．またそれらを実装した，コードクローンを自動的に検出するツールも多数開発されている．これらの検出技術はその検出単位によって，大まかに以下の 5 つに分類することができる [17][18]．

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

```
int num = k + size + 4;
```

(a) 変数宣言



(b) 対応する抽象構文木

図 2: 抽象構文木の例

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．

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

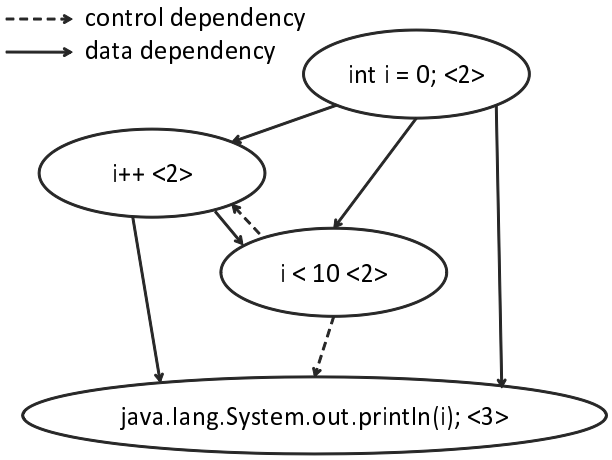


図 3: プログラム依存グラフの例

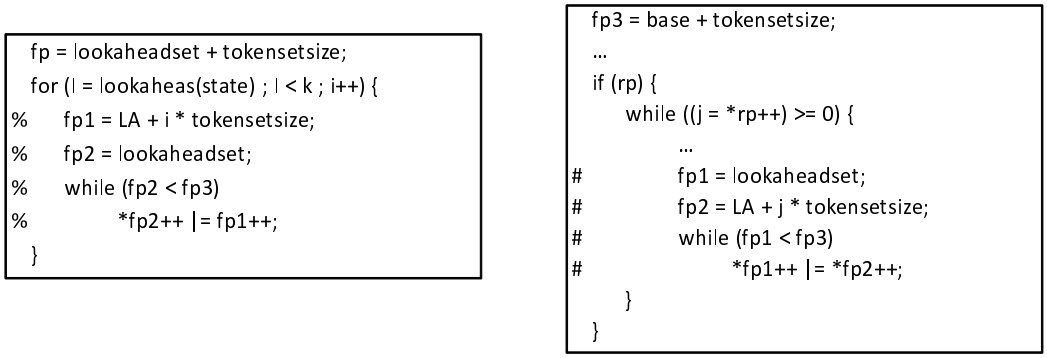
プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3)[19] 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



(a) コード片 1　　(b) コード片 2

図 4: 順序入れ替わりコードクローン

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．

2.2.2　大規模ソフトウェア群を対象とした検出手法

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，これら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．そこで，大規模なソフトウェア群から高速にコードクローンの検出を行う手法が提案されている．

ファイルクローン検出手法

佐々木らは，ファイルクローン検出ツール FCFinder を開発し，ファイルクローンの性質を調査した [3]．FCFinder はコメント文の削除や字句解析を行った後，ファイルからハッシュ値を計算しファイルクローンを検出する．また，佐々木らは総行数約 400,000,000 行の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．その結果，17 時間ほどで検出を終了し，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%はコメント文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であるファイルとそうでないファイルとで差異がなかったとも報告している．

Ossher らは，exact，FQN，fingerprint の 3 つの要素を組み合わせた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た [4]．exact は，各ソースファイルを 1 つの文字列とみなし，その文字列からハッシュ値を計算し，ハッシュ値が一致したファイルをファイルクローンとして検出する．FQN は，クラスの完全限定名が一致しているファイルをファイルクローン

として検出する．fingerprint は，ソースファイル中のメソッド名とフィールド名がどの程度等しいかを調査し，ある閾値を超えて一致しているファイルをファイルクローンとして検出する．Ossher らは，約 13,000 の Java ソフトウェアに対し上記の 3 つの要素を組み合わせて実験したところ，全ファイルの約 10%超がファイル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ときにそれ以前に開発されたソフトウェアの再利用を行うことなどが挙げられるとも報告している．

ファイルクローン検出手法では，ファイルを単位として比較を行う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場合は検出できないため，頻出する処理のライブラリ化を行う場合には，多くの見落としが発生してしまうと考えられる．

スケーラブルな細粒度コードクローン検出

Hummel らは，index を用いた増分的コードクローン検出手法を提案した [6]．彼らの手法は，ソースコード中の連続する文に対して index を計算し，その比較を行うことでコードクローンを検出する．また，彼らは 100 台のコンピュータを使用することで，730,000 行のソースコードから 36 分でコードクローンの検出が行えると報告している．

Cordy らは，入力ソースファイルと Debian のソースコード間でコードクローンを検出するためのツール DebCheck を作成した [7]．DebCheck は 1 度だけ 10 時間の準備が必要であるが，準備後は数分で入力ソースファイルと Debian のソースコード間でコードクローンを検出することが可能である．

Koschke はライセンスに違反したコードを検出することを目的として，suffix tree を用いた手法を提案した [8]．また実験により，この手法が index を用いた検出手法よりも高速に検出が可能であることを確認している．

これらの手法は大規模なデータセットに対し，実用的な時間で細粒度なコードクローンを検出することができる．しかし，検出されるコードクローンの数が膨大であり，また，プログラムの機能の一部のみからなるコードクローンが多く検出される．そのため，これらの手法をライブラリの作成に利用することは難しい．

2.3 異常処理

異常処理とはプログラムの耐故障性の向上に利用される処理である [20]．異常処理はプログラムの正常処理には影響を与えないため，正常処理が同じのメソッドであっても異常処理部分が異なることは十分考えられる．そのため，正常処理に着目してライブラリ化候補を検出する際には，異常処理が異なるメソッドであっても正常処理が同じであれば候補として検出するべきである．

本研究は以下の処理を異常処理として扱う．

- 例外処理
- アサーション

- ロギング処理
- プログレス表示処理
- プログラムの終了処理

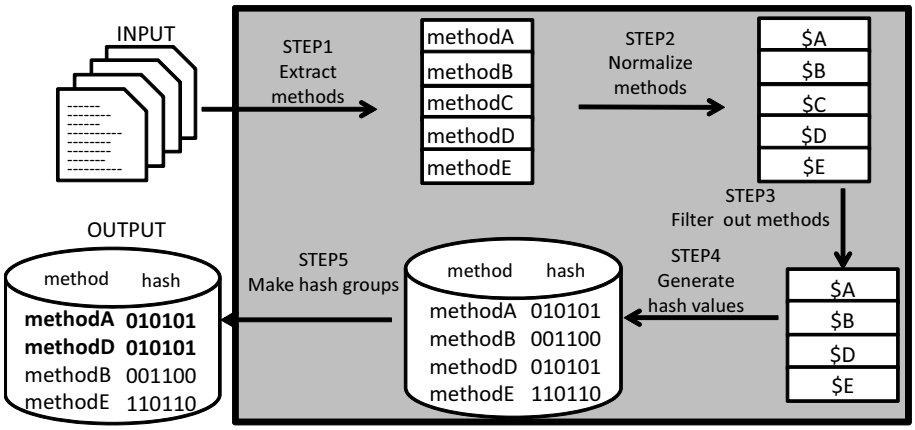


図 5: 石原らの手法の概要

3 研究の動機

3.1 メソッド単位のコードクローン検出

石原らは，開発者による共通処理のライブラリ化の支援を目的として，複数のソフトウェアを対象としたメソッド単位のコードクローン検出手法を提案した [10]．石原らの手法の概要を図 5 に示す．また，各 STEP の概要を以下に示す．

STEP1（メソッドの切り出し） 与えられたソースファイルからメソッドを切り出す．メソッドの切り出しは，ソースファイルから作成した抽象構文木上で実現する．抽象構文木を作成することで，改行や空白の違いなどが吸収される．

STEP2（正規化） STEP1 で切り出されたメソッドに対して正規化を行う．行う正規化は，変数名，リテラル，メソッド宣言部のメソッド名の特殊文字列への置換，および，修飾子，アノテーション，コメント文の削除である．

STEP3（フィルタリング） ソースファイルにはセッターやゲッター等の処理が単純であり，かつ，短いメソッドが多く存在する．このようなメソッドは多くが return 文や簡単な代入文のみで構成されているため，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処理が簡単に記述できるためライブラリ化する価値が小さく，コードクローンとしての検出は不要である．そこで，このようなメソッドはハッシュ値の計算を行わないようにフィルタリングを行う．

STEP4（ハッシュ値の計算） STEP3 を通過した各メソッドに対してハッシュ値を算出する．具体的には，メソッドを 1 つの文字列とみなしその文字列に対してハッシュ値の計算を行う．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同士はコー

ドクローンの関係にある．ハッシュ値の算出には MD5[21] を用いており，算出されたハッシュ値はメソッドごとにデータベースに格納する．

STEP5（ハッシュ値によるグループ化） STEP4 で算出されたハッシュ値の等しいメソッドをグループ化する．等しいハッシュ値を持つメソッドは等しい記述を持つ．そのため，ハッシュ値が等しいメソッドをグループ化することで，互いにコードクローンであるメソッド同士が 1 つのグループを構成する．これにより，コードクローンの関係にあるメソッドがどれか判別できるようになる．

また，石原らは 13,000 以上ものソフトウェアからなる UCI source code data sets[4][22] に対し手法を適用した．その結果，ファイル単位の検出手法では見つけることのできない約 1,160,000 ものメソッド単位のコードクローンを検出した．さらに，検出されたコードクローンの内，多くのソフトウェアに含まれている 100 のコードクローンを調査した結果，56%のコードクローンがライブラリ化に有用であったと報告している．

3.2 既存研究の問題

石原らの手法を用いることで大規模なデータセットからライブラリ化の候補を特定することができる．しかし，石原らの手法では，正常処理が同じであっても異常処理が異なるために検出されないコードクローンが存在する．異常処理を除去することで検出可能なコードクローンの例を図 6 に示す．これらのメソッドは実際のオープンソースプロジェクトに存在しているメソッドである．これらのメソッドの違いは，各メソッドにおいてハイライトで示している異常処理のみであり，異常処理は異なるが正常処理は等しい．そのため，ライブラリ化を行う候補を特定する際には，これらのメソッドを 1 つの候補として検出すべきである．しかし石原らの手法では異常処理を考慮していないため，これらをコードクローンとして検出することができない．そこで，異常処理に関するコードを除去したうえでメソッド単位のコードクローン検出を行う．

```
public static void copyFile(File in, File out) {
  if ((in == null) || (out == null)) {
    throw new IllegalArgumentException(
                  "in and out must not be null!");
  }
  try {
    FileInputStream fis = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while ((i = fis.read(buf)) != -1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  }catch(Exception e) {
     logger.error("Caught " + e.getClass().getName(), e);
  }
}
```

(a) exception throwing,try-catch,logging

```
static public void CopyFile(File source, File dest)
                                          throws Exception {
  if (source == null || dest == null) {
    throw new IllegalArgumentException();
  }
  pLogger.log(Level.FINEST, "CopyFile",
                "Copying file from " +source.toString()
                  + " to " + dest.toString());
  FileInputStream fis = new FileInputStream(source);
  FileOutputStream fos = new FileOutputStream(dest);
  byte[] buf = new byte[1024];
  int i = 0;
  while ((i = fis.read(buf)) != -1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(b) throws clause,exception throwing,logging

```
public void copySingleFile(File in, File out)
                          throws FileCopyException {
  try{
    FileInputStream fis  = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while((i=fis.read(buf))!=-1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  } catch (Exception ex) {
    throw new FileCopyException(
                  "IO stream Exception",ex);
  }
}
```

(c) throws clause,exception throwing,try-catch

```
static void copyFile(File in, File out)
                    throws Exception {
  try {
    FileInputStream fis = new FileInputStream(in);
    FileOutputStream fos = new FileOutputStream(out);
    byte[] buf = new byte[1024];
    int i = 0;
    while ((i = fis.read(buf)) != -1) {
      fos.write(buf, 0, i);
    }
    fis.close();
    fos.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

(d) throws clause,try-catch

```
public static void copyFile(File in, File out)
                               throws Exception{
  System.err.println("copyFile: " + in + " -> " + out);
  FileInputStream fis  = new FileInputStream(in);
  FileOutputStream fos = new FileOutputStream(out);
  byte[] buf = new byte[1024];
  int i = 0;
  while((i=fis.read(buf))!=-1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(e) throws clause,System.err.println

```
private void copyFile(File in, File out)
                              throws IOException {
  FileInputStream fis  = new FileInputStream(in);
  FileOutputStream fos = new FileOutputStream(out);
  byte[] buf = new byte[1024];
  int i = 0;
  while((i=fis.read(buf))!=-1) {
    fos.write(buf, 0, i);
  }
  fis.close();
  fos.close();
}
```

(f) throws clause

図 6: 異常処理のみが異なるメソッドの例

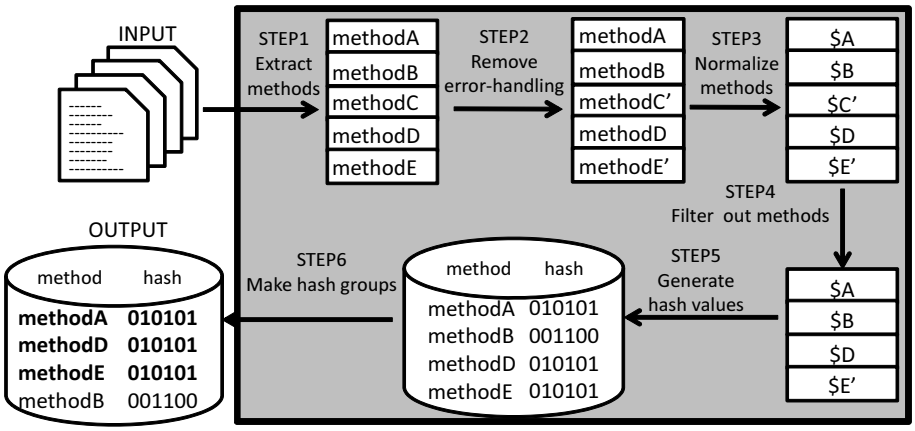


図 7: 提案手法の概要

4 異常処理を考慮したメソッド単位のコードクローン検出

4.1 概要

本研究では石原らの手法を元に異常処理の除去を行うメソッド単位のコードクローン検出手法を提案する．そのため提案手法の入出力は石原らの手法と等しく，対象とするソースファイル群を入力として受け取り，検出されたコードクローンの情報を出力として返す．提案手法の概要を図 7 に示す．本研究では，正規化処理を行う前に異常処理の除去を行う．また，本研究では Java で記述されたプログラムを対象としている．

4.2 異常処理の除去

各メソッドから異常処理を検出し，除去を行う．本研究では 2.3 節に示した異常処理を対象としており，例えば Java においては以下に示す処理が除去の対象となる．

- throw 文
- try〜catch〜finally と catch 節内部の処理
- throws 節
- return null
- assert 文
- Logger クラスに含まれるメソッド呼び出し
- 標準出力への出力

- 標準エラー出力への出力
- System.exit() メソッド

異常処理の特定および除去はメソッドごとに切りだされた抽象構文木のノード単位で行われる．このとき，抽象構文木を後順で走査することにより，複文を表すノードの全ての子ノードが異常処理として除去された場合も同時に除去を行う．ここで複文とは，複数の文から成り立つブロックを持つ文のことを指し，do，for，if，switch，synchronized，while 文を複文と定義する．

図 8，9 に異常処理除去の例を示す．図 8 では，throw 文と throws 節の除去を行っている．また，図 8(c)〜図 8(e) に抽象構文木上での除去の流れを示す．ここで，抽象構文木の各ノードは図 8(a) の各行と対応している．まず，図 8(c) から throw 文と throws 節に対応するノードが除去される（図 8(d)）．このノードの除去により，if 文を表すノードから子ノードがなくなるために除去を行い（図 8(e)），除去後のコードとなる．

図 9 では，try 文の除去を行っている．また，図 9(c)〜図 9(e) に抽象構文木上での除去の流れを示す．ここで，抽象構文木の各ノードは図 9(a) の各行と対応している．まず，図 9(c) から異常処理として catch 節内部のノードが除去される（図 9(d)）．次に，try 文を表すノードの親ノード（ノード a）を try 節および finally 節の各子ノードの新しい親ノードとして設定する（図 9(e)）．その後，try 文に対応するノードの除去を行い（図 9(f)），除去後のコードとなる．

4.3 実装

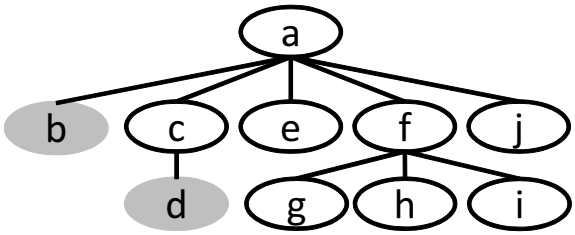
本研究では，石原らの作成したメソッド単位のコードクローン検出ツールを元に，提案手法を実現するツールを作成した．現在のところ，作成したツールの解析対象言語は Java のみである．また，抽象構文木の作成には JavaDevelopmentTools[23] を用いている．

```
a. public static byte[] OR(byte[] a,byte[] b)
b.   throws Exception
     {
c.     if(a.length != b.length) {
d.       throw new Exception();
       }
e.     byte[] v = new byte[a.length];
f.     for(int i=0; i<v.length; i++) {
g.        int A=(a[i]<0 ? a[i]+256 : a[i]);
h.        int B=(b[i]<0 ? b[i]+256 : b[i]);
i.        v[i]=(byte)(B | A);
       }
j.     return(v);
   }
```

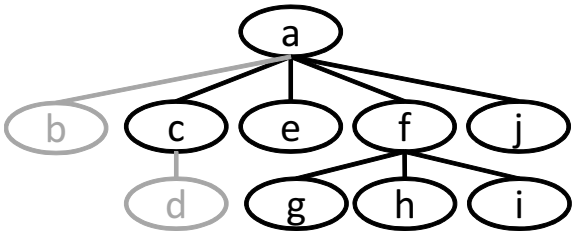
(a) 除去前のコード

```
a. public static byte[] OR(byte[] a,byte[] b)
     {
e.     byte[] v = new byte[a.length];
f.     for(int i=0; i<v.length; i++) {
g.        int A=(a[i]<0 ? a[i]+256 : a[i]);
h.        int B=(b[i]<0 ? b[i]+256 : b[i]);
i.        v[i]=(byte)(B | A);
       }
j.     return(v);
   }
```

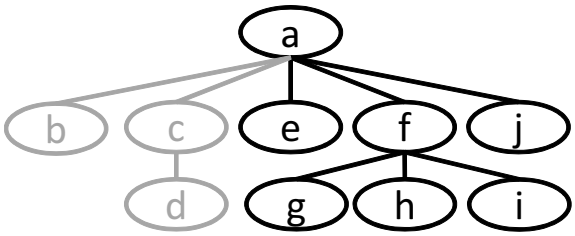
(b) 除去後のコード



(c) 除去前の抽象構文木



(d) throw 文，throws 節を除去



(e) 空になった if 文を除去

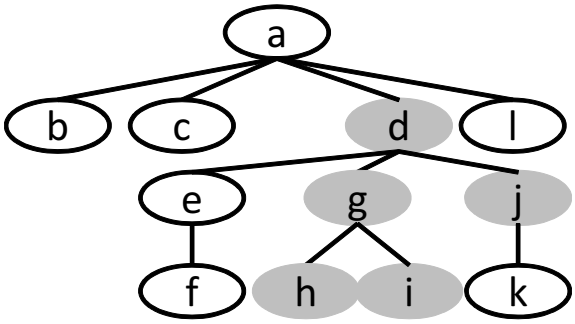
図 8: throw 文，throws 節の除去例

```
a. public String read(){
b.    readLock.lock();
c.    String data=null;
d.    try {
e.       while(hasNext()){
f.          data= new String(doRead());
         }
g.    } catch(Exception e){
h.       e.printStackTrace();
i.       data=null;
j.    } finally {
k.       readLock.unlock();
      }
l.    return data;
   }
```

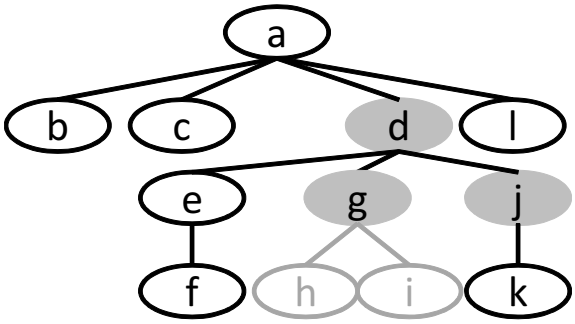
(a) 除去前のコード

```
a. public String read(){
b.    readLock.lock();
c.    String data=null;
e.    while(hasNext()){
f.       data= new String(doRead());
      }
k.    readLock.unlock();
l.    return data;
   }
```

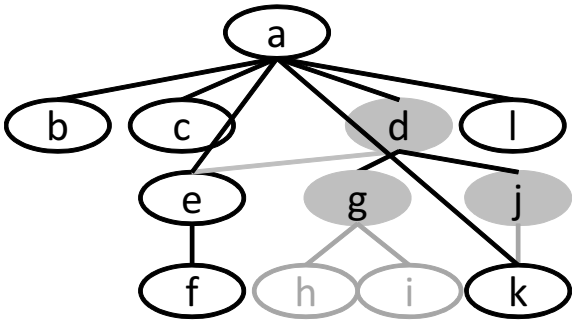
(b) 除去後のコード



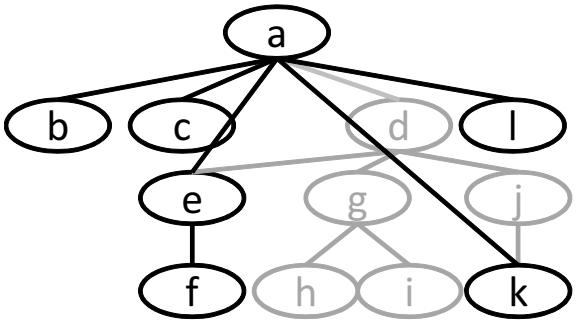
(c) 除去前の抽象構文木



(d) catch 節内部を除去



(e) 親ノードの変更



(f) try～catch～finally 文を除去

図 9: try～catch～finally 文の除去例

5　異常処理除去の精度の評価

5.1　実験の概要

本実験では，提案手法の適用により，開発者が異常処理として記述した処理をどの程度除去できているかを調査する．本実験ではまず被験者となる開発者に，自身が過去に開発したプログラムに存在するメソッドを提示し，異常処理として記述した箇所を選択してもらう．その後，各メソッドから提案手法に含まれる異常処理除去の適用により除去された箇所と被験者が異常処理として提示した箇所を比較することで，除去の適合率と再現率を調査する．

本実験の被験者はコンピュータサイエンスを専攻している大学院生 4 名である．各被験者にはそれぞれ 10 のメソッドを提示し，合計 40 のメソッドに対して異常処理除去の精度を調査した．

5.2　実験結果

被験者によって選択された 40 のメソッドに含まれる異常処理と，提案手法によって除去された処理がどの程度一致するかを調査した．この調査は，被験者によって選択された各異常処理に対して行った．実験結果を図 10 に示す．提案手法の異常処理除去により，被験者が異常処理として記述した 30 の処理の内，25 の異常処理を除去することができた．式 1，2 を用いて算出される適合率と再現率はそれぞれ，78.1%，83.3%である．ここで，正しく除去できた処理数とは被験者が異常処理として選択した処理のうち，提案手法で除去できた処理の数のことである．

$$\text{適合率} = \frac{\text{正しく除去できた処理数}}{\text{提案手法で除去した処理数}} \tag{1}$$

$$\text{再現率} = \frac{\text{正しく除去できた処理数}}{\text{被験者によって選択された異常処理数}} \tag{2}$$

被験者が異常処理として記述したが，提案手法では除去できなかった例を図 11 に示す．このメソッドにおいて，被験者はハイライトされた箇所を異常処理として記述している．しかし提案手法では，setAttribute() メソッドを用いたメッセージ送信を異常処理とみなしていない．そのため，この処理を異常処理として除去することができなかった．

また，被験者は異常処理と判定しなかったが除去してしまった処理を含むメソッドを図 12 に示す．このメソッドでは特定の条件を満たすパスを出力する処理を行っており，ハイライトされた出力処理は正常処理の一部であり，除去すべきでない．しかし，提案手法では標準出力への出力処理を異常処理として扱っているため，除去してしまっている．

被験者	選択された異常処理	提案手法の適用結果			適合率	再現率
		正しく除去できた処理	除去できなかった処理	異常処理ではないが除去してしまった処理		
A	2	2	0	0	100	100
B	14	9	5	1	90.0	64.3
C	8	8	0	4	66.7	100
D	6	6	0	2	75.0	100
合計	**30**	**25**	**5**	**7**	**78.1**	**83.3**

図 10: 調査結果

```
public String unzipAndRegist(File file,
       String projectName, String labelName)
                          throws IOException {
  try {
    uncompress.uncompress(file.getCanonicalPath());
    String savePath = file.getCanonicalPath().substring(0,
                file.getCanonicalPath().lastIndexOf(sep));
    synchronized (session) {
      session.setAttribute("info",
                 "uploaded file:" + savePath);
      session.setAttribute("totalFileNum",
            countFiles(new File(savePath)));
    }
    SQLHelper.getSQLHelper().setSavepoint();
    registToDB(new File(savePath), projectName, labelName);
    …
```

図 11: 異常処理の除去に失敗したメソッド

```
private static void searchFile(File f) {
    File[] files = f.listFiles();
    for (File fileName : files) {
      if (fileName.isDirectory()) {
        // ディレクトリ時の処理
        if (!(fileName.getName().endsWith("tags")
              || fileName.getName().endsWith("branches"))) {
            if (fileName.getName()
                    .endsWith("content.zip.dir")) {
              System.out.println(fileName.getParent());
            }
            searchFile(fileName);
        }
      } else if (fileName.getName().endsWith("java")) {
    …
```

図 12: 誤って除去してしまった処理を含むメソッド

6 石原らの手法との比較

6.1 実験の概要

提案手法の有用性を示すために以下の 3 つの項目を調査する．

調査項目 A 異常処理の除去を行う場合と行わない場合で，検出されるコードクローンにどの程度違いがあるか

調査項目 B 提案手法によって検出されるコードクローンがライブラリ化に有用であるか

調査項目 C 異常処理の除去を行う場合と行わない場合で検出時間がどの程度異なるか

本研究では実験対象として，石原らが実験に用いた”UCI source code data sets”[4][22] を用いた．石原らと同様にデータセット内で trunk，tags，branches を同時に含むソフトウェアは trunk 内のソースファイルのみを，バージョンの異なる同一のソフトウェアは最新バージョンのソフトウェアのみを検出対象とし，ソースコード自動生成ツールによって生成されたファイルを検出対象から除外した．本実験における実験対象の構成を表 1 に示す．本実験は CPU 数 2，計 8core（2.27GHz），メモリ 32GByte の環境で行い，検出結果を格納するデータベースは SSD 上に作成した．また，追加した除去処理の影響のみを調査するため，石原らの手法と提案手法においてフィルタリング処理条件を 30 トークン以下のメソッドを除去するように統一した．

本実験おいて，クローンセットに含まれるメソッドが分布しているソフトウェアの数を NoS(the Number Of Software system) と定義する．ここで，クローンセットとは互いにコードクローン関係にあるメソッドの集合のことである．多くのソフトウェアで使用されているメソッドは，クローン

セットとして検出されたときに NoS の値が大きくなる．したがって，NoS の値が大きいクローンセットはライブラリ化すべきであると考えられる．

また，クローンセットに含まれる各メソッドに対して代入が行われたフィールド数を算出し，その最大値を NAF(the Number of Assignment Fields) として定義する．また，クローンセットに含まれるメソッドに返り値が存在する場合，NAF 値に 1 を加算する．NAF 値が 2 以上のクローンセットは，メソッドの呼び出し元に対して複数の値を返さなければならないため，メソッド単体で抽出することが難しい．それに対して，NAF が 0 もしくは 1 の場合は，メソッド単位で抽出することが可能であり，容易ライブラリ化を行うことができると考えられる．

6.2 調査項目 A：異常処理除去の有無による検出結果の比較

実験対象に対し，異常処理除去を行った場合と行わなかった場合の検出結果を表 2 に示す．

また，異常処理の除去によりハッシュ値が変化したメソッドや，要素が変化したクローンセットがどの程度存在するか調査した．その結果を表 3，4 に示す．ここで，調査したメソッドは異常処理を含んだ状態でトークン数 30 以上のメソッドである．

表 1: 実験対象の構成

ソフトウェア数	13,193
ファイル数	2,072,490
メソッド数	19,416,603
総行数	361,663,992
全容量	30.6GByte

表 2: 検出結果

	異常処理除去あり	異常処理除去なし
フィルタリングを通過したメソッド数	6,939,000	7,425,897
検出されたクローンセット数	943,761	1,004,718
クローンセットに含まれるメソッド数	3,351,615	3,573,676

表 3: 異常処理除去によるメソッドへの影響

ハッシュ値が変化したメソッド数	3,403,104(45.8%)
ハッシュ値が変化しなかったメソッド数	4,022,793(54.2%)

この結果より，約 45%のメソッドが何らかの異常処理を行っていることがわかる．また，異常処理の除去により，プログラムの正常処理は同じであるが異常処理の異なるメソッド群を，1 つのクローンセットとして検出できるようになった．これにより，82,639 のメソッドが新たにコードクローンとして検出された．

また，検出されなくなったクローンセットが存在するが，これは異常処理を除去したためにトークン数が減少し，フィルタリングを通過しなくなったメソッドが存在するためである．

6.3 調査項目 B：ライブラリ化に有用な候補を検出できているか

検出されたコードクローンがライブラリ化に有用であるかの調査を行う．しかし，検出されるクローンセットの数が膨大になるため，すべてのクローンセットを分析することは現実的に困難である．そこで，NAF 値が 1 以下のクローンセットを NoS 値でソートしたものの内，上位 100 を実際に目で見て調査した．

本実験では，石原らと同様に，コードクローンの発生原因を以下の 3 つに分類した．

- 抽象クラスの継承，インターフェイスの実装
- ソースコードの流用
- 汎用的な処理を行うメソッド

その結果を表 5 に示す．ここで合計が 100 を超えているが，これは発生原因に重複があるためである．本実験では石原らと同様に汎用的な処理を行うメソッドに分類されたクローンセットを除く，65 のクローンセットをライブラリ化すべきであると判断した．

図 13 にライブラリ化すべきであると判断したメソッドの一例を示す．この 2 つのメソッドは，ともに JTable のソートを実行するメソッドであり，ハイライトされた箇所のロギング処理のみが異なっ

表 4: 異常処理除去によるクローンセットへの影響

新たに検出されたクローンセット数	16,820
要素数が増えたクローンセット数	19,938
消失したクローンセット数	69,218

表 5: クローンセットの分類結果

抽象クラスの継承，インターフェイスの実装	17
要素数が増えたクローンセット数	64
汎用的な処理を行うメソッド	35

```
public void  sort(Object sender)
{
   checkModel();
   compares = 0;
   shuttlesort((int[])indexes.clone(),
                  indexes, 0, indexes.length);
   System.out.println("Compares: "
                   +compares);
}
```

(a) sun

```
private void sort(Object sender)
{
   checkModel();
   compares = 0;
   shuttlesort((int[])indexes.clone(),
                  indexes, 0, indexes.length);
   logger.debug("Compares: "
                   + compares);
}
```

(b) BasicQuery

図 13: ライブラリ化すべきと判断したコードクローンの例

```
public static String
       normaliseNewlines(String input)
{
   input = input.replaceAll("¥r¥n", "¥n");
   input = input.replaceAll("¥r", "¥n");
   return input;
}
```

(a) jbootcat

```
public static String
          convertToEscapes(String message)
{
   message = message.replaceAll("<", "&lt;");
   message = message.replaceAll(">", "&gt;");
   return message;
}
```

(b) YAWL

図 14: ライブラリ化すべきでないと判断したコードクローンの例

ている．このメソッドを含むクローンセットは，96 のソフトウェアにまたがって存在する．このような処理は，今後も多くのソフトウェアで行われる可能性があるため，ライブラリ化を行うべきであると判断した．

また，ライブラリ化には適していないと判断したコードクローンの一例を図 14 に示す．この 2 つのメソッドはユーザ定義名と文字列リテラルのみが異なるためコードクローンとして検出されている．このようなメソッドを含むコードクローンは多くのソフトウェアに存在するが，処理が単純であり，かつ，ソフトウェアごとに異なる目的で用いられる可能性が高いためライブラリ化には適さない．

また，石原らの手法において NoS メトリクス値の上位に存在していた size() や close() 等のメソッドを含むコードクローンは検出されなかった．これらのメソッドは多くのクラスで定義されているが，単純で汎用的な処理を行っているためライブラリ化には適さない．しかし，このようなメソッドは，特定の変数が null や 0 であるといった異常処理を伴う条件判定と併用されやすいため，石原らの手法では検出されている．提案手法ではこのような異常処理が除去されたことにより，メソッドサイズが小さくなり，フィルタリングを通過しなくなったため検出されていない．

6.4 調査項目 C：実行時間

作成したツールを対象のデータセットに適用した結果，異常処理の除去を行う場合は 1 時間 50 分，行わない場合は 1 時間 51 分をそれぞれ要した．表 6 に各処理に要した時間を示す．

異常処理の除去を行う場合と行わない場合での実行時間を比較すると，除去を行う場合ではメソッ

ドの切り出しからハッシュ値の計算までの処理にわずかに時間がかかっている．これは，異常処理の除去を追加しているためであるが，時間の増加はわずかであり，この処理にはほとんど時間がかかっていない．また，ハッシュ値のグループ化に要する時間が減少しているが，これは表 2 で示したようにフィルタリングを通過したメソッド数が減少したためである．実験結果より，作成したツールでは，約 360,000,000 行のソースコードに対し 2 時間以内で検出を行うことができ，実用的な時間で実行可能であるといえる．

表 6: 解析時間

処理内容	除去あり	除去なし
メソッドの切り出し 異常処理の除去 正規化 フィルタリング ハッシュ値の計算	1 時間 29 分	1 時間 28 分
ハッシュ値によるグループ化	21 分	23 分
合計	1 時間 50 分	1 時間 51 分

7　妥当性への脅威

7.1　ハッシュ値の衝突に関して

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の判定にハッシュ値を使用している．そのためコードクローン検出の際に文字列が異なるにも関わらずハッシュ値が偶然一致するハッシュ値の衝突が起こった場合，本来コードクローンでないメソッドがコードクローンとして検出されるおそれがある．しかし，本研究ではハッシュ値の計算に 128bit の MD5 アルゴリズムを使用しているため衝突確率は極めて低いと考えられる．また本研究では，同じハッシュ値を持つメソッドすべてについて，異常処理の除去と正規化処理を行った後の文字列が一致するかどうかを調べるプログラムを作成し，ハッシュ値の衝突がなかったことを確認した．ただし，より多くのソフトウェアに対し提案手法を適用した場合，ハッシュ値の衝突が起こる可能性がある．

7.2　正規化，フィルタリングに関して

本研究は石原らの手法を元にしており，以下の正規化処理を行っている．

- 変数名，リテラル，メソッド宣言部のメソッド名は特殊文字列に置換する．
- 修飾子，アノテーション，コメント文は削除する．

しかし，型名の正規化の有無やフィルタリングを行うトークン数の変更等により，本研究では検出できていないコードクローンを検出できる可能性がある．

8 おわりに

本論文では，異常処理のみが異なるメソッド群を 1 つのライブラリ化候補として検出することを目的として，異常処理の除去を追加したメソッド単位のコードクローン検出手法を提案した．また，提案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，異常処理除去の精度調査，および，13,000 以上のソフトウェアからなる大規模なソフトウェア群に対して適用実験を行った．

精度調査の結果，開発者が異常処理として記述した処理の多くを提案手法によって正しく除去できることを確認した．異常処理除去の適合率は 78.1%，再現率は 83.3%であった．

また，大規模データセットへの適用実験の結果，データセットに含まれる 360,000,000 行のソースコードに対し，2 時間程度でコードクローンの検出を行うことができた．さらに，検出されたコードクローンを調査した結果，82,639 のメソッドが新たにコードクローンとして検出されていることを確認した．また，多くのソフトウェアに存在するコードクローンの内，65%のコードクローンがライブラリ化に有益であった．

今後の課題として，

- 本研究とは異なる正規化やフィルタリング処理を行った場合に，検出されるコードクローンどの程度違いがあるかの調査
- 対象となるファイルを Java 以外にも拡張し，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の調査

等が挙げられる．

謝辞

本研究を行うにあたり，理解ある親身なご指導を賜り，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，適切なご指導を賜り，有益かつ的確なご助言を多数頂きました 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，親切なご指導を頂き，多くのご助言，ご助力を頂きました 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.

また，本研究に関して多くのご助言を頂くとともに，様々な面において親切なご助力，ご協力を頂きました楠本研究室の皆様に心より感謝致します.

参考文献

[1] C. Braun. Nato standard for the development of reusable software components. In *NATO Communications and Information Systems Agency*, 1992.

[2] 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.

[3] 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, Aug. 2011.

[4] 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.

[5] Y. Higo, K. Tanaka, and S. Kusumoto. Toward identifying inter-project clone sets for building useful libraries. In *Proc. of the 4th International Workshop on Software Clones*, pp. 87–88, May 2010.

[6] 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, Sep. 2010.

[7] J. R. Cordy and C. K. Roy. Debcheck: Efficient checking for open source code clones in software systems. In *Proc. of the 19th International Conference on Program Comprehension*, pp. 217–218, June 2011.

[8] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proc. of the 16th European Conference on Software Maintenance and Reengineering*, pp. 309–318, Mar. 2012.

[9] W. Shang, B. Adams, and A. E. Hassan. An experience report on scaling tools for mining software repositories using mapreduce. In *Proc. of the 25th International Conference on Automated Software Engineering*, pp. 275–284, Sep. 2010.

[10] 石原知也, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二. 大規模なソフトウェア群を対象とするメソッド単位でのコードクローン検出. 情報処理学会論文誌（掲載予定）, Vol. 54, No. 2, Feb. 2013.

[11] 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, Sep. 2001.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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 D, Vol. J91-D, No. 6, pp. 1465–1481, June 2008.

[18] 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.

[19] 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, pp. 97–104, Sep. 2009.

[20] C. Fu and B. G. Ryder. Navigating error recovery code in java applications. In *Proc. of the OOPSLA workshop on Eclipse Technology eXchange*, pp. 40–44, Oct. 2005.

[21] R. Rivest. The md5 message-digest algorithm. RFC 1321 (Informational), April 1992. Updated by RFC 6151, `http://www.ietf.org/rfc/rfc1321.txt`.

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

[23] Java development tools. available at `http://www.eclipse.org/jdt/`.