特別研究報告

題目

コードクローンの自動集約による
削減可能なソースコード行数の測定

指導教員

楠本 真二 教授

報告者

中川 将

平成 31 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 30 年度 特別研究報告

コードクローンの自動集約による
削減可能なソースコード行数の測定

中川 将

内容梗概

コードクローンとは，ソースコード中に存在する互いに一致または類似しているコード片のことである．コードクローンを一つのメソッドやクラスに集約することにより，ソースコード行数の削減が可能である．削減可能なソースコード行数を推定する手法として，コードクローンの情報を解析する手法が既存研究では提案されている．しかし，コードクローンの中には集約するとコンパイルやテストに失敗するものが存在すると著者らは考えた．そのため，コードクローンの集約により削減可能なソースコード行数を推定するとき，コードクローンの情報のみでは正確な推定を行うことはできない．そこで本研究では，より正確な削減可能なソースコード行数を推定する手法を提案する．提案手法では，それぞれのコードクローンの集合に対して集約，コンパイル，テストを自動で行い，それらが実際に集約可能なのかを判定したうえで，削減行数を測定する．さらに提案手法を用いてオープンソースソフトウェアの削減可能なソースコード行数の測定を行った．その結果，コードクローンの情報のみで推定された削減可能な行数とは異なる数値を得た．

主な用語

コードクローン
ソフトウェア保守
リファクタリング
削減可能ソースコード行数

目次

図目次

表目次

1　まえがき

ソフトウェア開発では，完成までに人員や資金，設備などの様々なリソースが必要である．また，一度完成したソフトウェアであっても機能追加やバグ修正などの保守が必要であり，保守にもリソースが必要となる [11]．ソフトウェアの保守に必要なリソースはそのソフトウェアの規模や複雑さから算出されることが多い [13]．ソフトウェアの規模はソースコードの行数から測定できる．しかし，冗長なソースコードが多く含まれているとソフトウェアの規模を正確に測定できない．したがって，ソフトウェアの規模を測定し保守に必要なリソースを見積もるためには，冗長なソースコードを削減した場合の行数を推定する必要がある．

冗長なソースコードの一つとしてコードクローンが挙げられる．コードクローンとは，ソースコード中に存在する互いに一致または類似しているコード片のことである．コードクローンとなっているコード片を一つのモジュールに集約することで，冗長なソースコードを削減できる．

コードクローンを集約する方法の一つがリファクタリングである．リファクタリングとはソフトウェアの外部的振る舞いを保ちつつ内部の構造を改善する行為である [3]．リファクタリングの一つにメソッドの抽出という手法がある．メソッドの抽出とは，既存のソースコードから一部のコード片を切り出し新たなメソッドにすることである．メソッドの抽出を行うことで，コードクローンの集合を一つのメソッドに集約できる [21]．コードクローンの集約を行うことでソースコードの行数が削減可能である．そのため，コードクローンの情報から削減可能なソースコード行数を推定できると考えられている [16]．しかし，検出されたコードクローンの中には，集約を行うことでコンパイルやテストに失敗するコードクローンがある．また，異なるコードクローンが部分的に重なることで，どのコードクローンを集約するべきか選択する必要がある．このような問題点から，リファクタリングを行うことで削減可能なソースコード行数を推定することは容易ではない．

既存研究では，削減可能なソースコード行数を推定する手法としてコードクローンの情報を解析するという手法を提案している [19]．しかし，実際にコードクローンの集約を行っていない．そのため，算出された推定値に対して，その行数が実際に削減可能であるかどうか分からない．また，実際にコードクローンの集約を行うと，コンパイルとテストに失敗して集約を行えない可能性も考えられる．例えば，検出されたコードクローン間の対応する変数の型が異なる場合，集約を行うとコンパイルに失敗するので集約を行うことはできない．また，テストを行わなければ外部的振る舞いが保たれていることを保証できない．しかし，検出された大量のコードクローン全てに対して，手作業で実際に集約が可能かを確認することには多大な労力が必要となる．

そこで本研究では，コードクローンの検出，ソースコードの変更，コンパイル及びテストを全て自動で行い，実際に削減行数を測定することで，より正確なソースコードの削減可能行数を算出する手法を

提案する．また，提案手法を用いて Java 言語で記述されたプロジェクトに対して実験を行い，削減可能行数を算出した．その結果，既存研究とは異なる数値を得た．

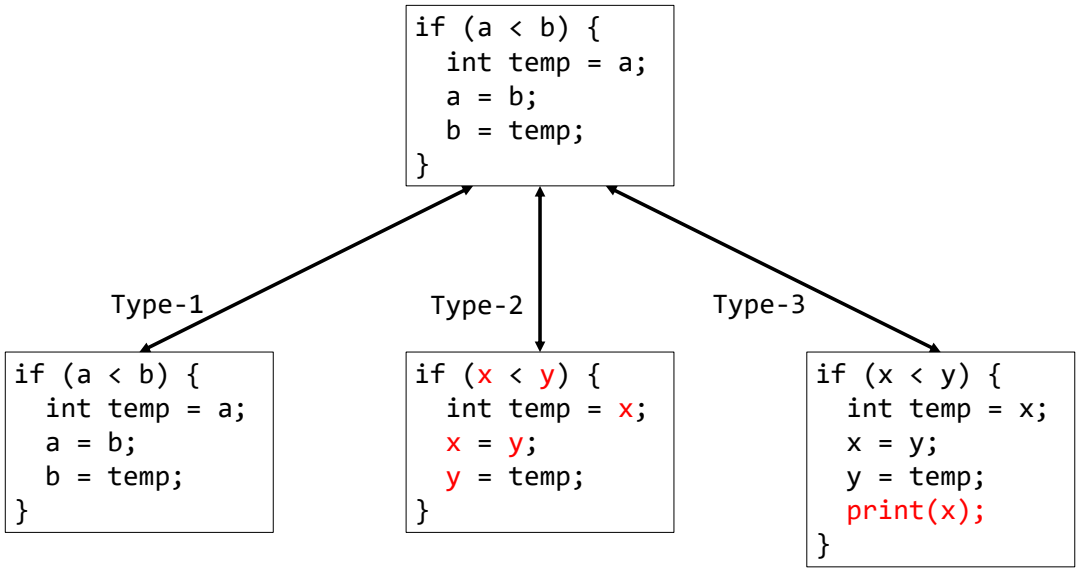


図 1　クローンの例

2　準備

本章ではまずコードクローンについて説明する．次に，コードクローンのリファクタリングについて説明する．

2.1　コードクローン

コードクローン（以下，クローン）とは，ソースコード中に存在する互いに一致または類似しているコード片のことである．一般的に，クローンはその類似度に基づいて以下の三つのタイプに分類される [1]．図 1 に例を示す．

Type-1:　空白やタブの有無，括弧の位置などを除いて，完全に一致するクローン

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，変更が行われたクローン

クローンの主な発生要因はコピーアンドペーストである [20]．コピーアンドペーストを用いることによりソフトウェアの実装速度が速くなる一方で，発生するクローンはソフトウェアの保守性が低下する原因の一つである [6]．例えば，あるコード片にバグが存在した場合，そのクローンに対しても同様のバ

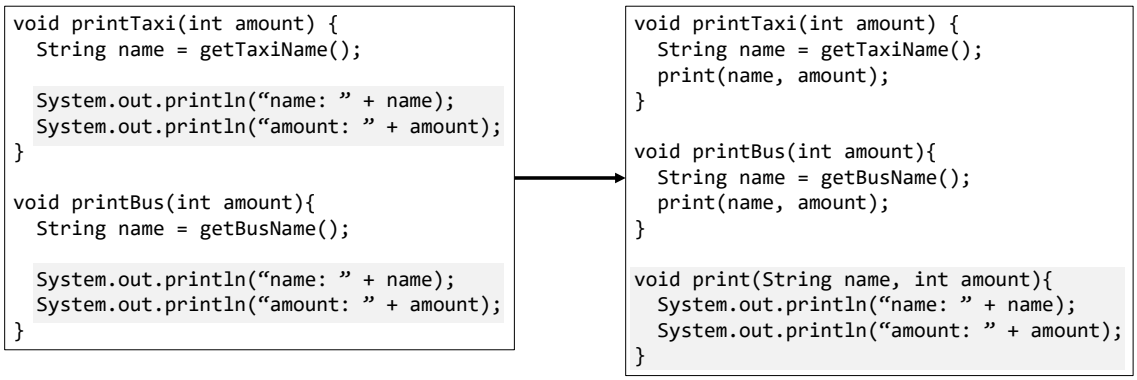


図 2　メソッド抽出の例

グが存在する可能性があり，同様の変更を検討する必要がある [9]．したがって，クローンがソースコード中のどこに存在しているか，どの程度存在しているかを把握することはソフトウェアの保守において重要である．

そのため，自動でクローンを検出する研究が盛んに行われており，多くのクローン検出ツールが開発されている [8, 7, 2]．コード片同士が一致または類似しているかを判断する基準はそれぞれの検出手法や検出ツールによって異なる [18]．

また，検出されたクローンを一つのモジュールに集約することにより，潜在的なバグの修正箇所とソースコード行数の削減が可能である．クローンを集約する手法の一つとしてリファクタリングが挙げられる．

2.2　コードクローンのリファクタリング

リファクタリングとは，外部的振る舞いを変えることなく内部の構造を改善する行為である [3]．リファクタリングを提唱した Fowler は，リファクタリングを行うべきコードとして重複したコード，即ちクローンを挙げている．これまでに様々なリファクタリング手法が考案されており，クローンの集約に有効な手法も複数存在する [21]．クローンの集約に有効なリファクタリング手法の一つがメソッド抽出である．メソッド抽出とは，既存のソースコードから一部のコード片を切り出し新たなメソッドにすることである．メソッド抽出の例を図 2 に示す．クローンとなってるコード片の集合（以下，クローンセット）に対してメソッド抽出を行うことでクローンを集約できる．

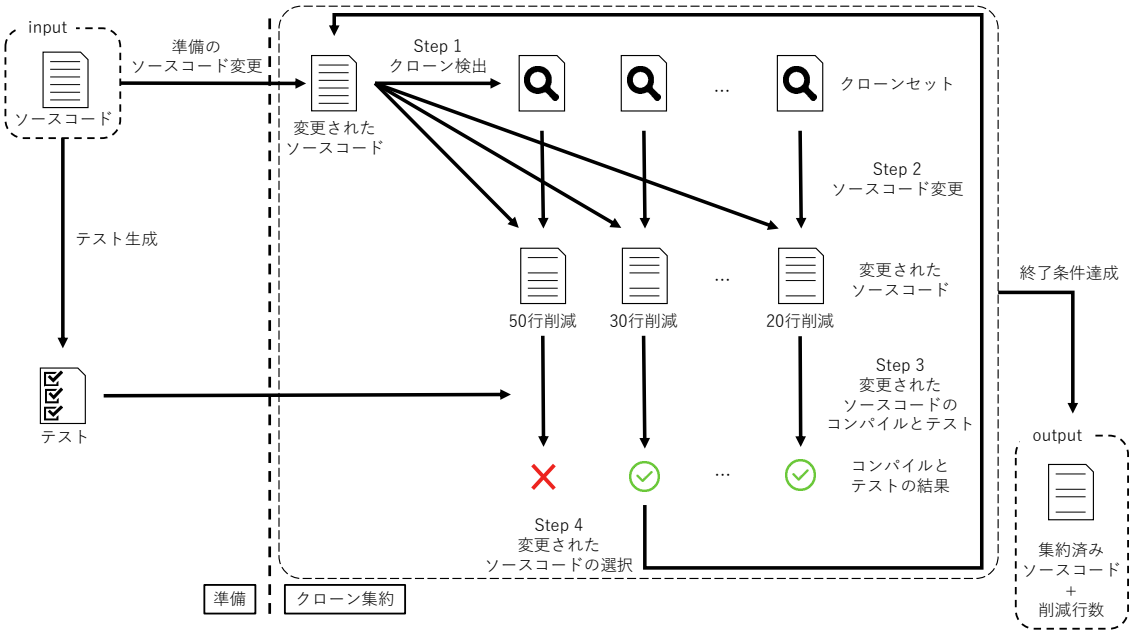


図 3　提案手法の概要

3　提案手法

本研究では，クローンの検出，ソースコードの変更，コンパイル，テストを繰り返し行いソースコードの削減可能行数を算出する手法を提案する．提案手法の概要を図 3 に示す．本研究における提案手法の入力はソースコードである．出力は集約可能なクローンを全て集約したソースコードと削減されたソースコード行数である．

本研究の提案手法は以下の二つの工程で構成される．

- 準備
- クローン集約

3.1　準備

準備ではソースコードの変更前後で外部的振る舞いが変化していないかを確認するために，対象とするソースコードの単体テストを用意する．単体テストはソースコードから自動生成する．

また，メソッド抽出で切り出されたメソッドの宣言を行うクラスを用意する．それに加えて，メソッドのアクセス修飾子及びローカル変数の未初期化によるコンパイルエラーとコーディングスタイルによる削減行数の変化を防ぐために，ソースコードに変更を加える．合計で以下の四つの変更を行う．それぞれの変更内容については 4.1 で詳しく述べる．

- 抽出されたメソッドの宣言を行うクラスの用意
- メソッドのアクセス修飾子の変更
- ローカル変数の初期化
- フォーマッタの適用

この準備はクローン集約を行う度に実行する必要はない．

3.2 クローン集約

クローン集約は次の 4 ステップから構成される．これらはすべて自動で行われる．

Step 1: クローン検出
Step 2: ソースコード変更
Step 3: 変更されたソースコードのコンパイルとテスト
Step 4: 変更されたソースコードの選択

Step 1 では，対象とするソースコード中に存在するクローンセットを検出する．Step 2 では，ソースコードに対して Step 1 で検出したクローンセットの一つを集約する変更を加える．集約の際に抽出されたメソッドは，3.1 で説明した準備で用意したクラス内に宣言する．Step 3 では，変更されたソースコードに対してコンパイルとテストを行い，外部的振る舞いが変化していないかどうかを判定する．以降，Step 2 で変更されたソースコードの中で，行数が削減されコンパイルとテストを通過するようなソースコードを**選択可能なソースコード**と定義する．Step 1 で検出したすべてのクローンセットに対して Step 2 と Step 3 を行い，Step 4 で選択可能なソースコードの中で削減行数が最も多いソースコードを選択する．そして選択したソースコードに対して繰り返し Step 1 以降を実行する．終了条件に到達した時点でクローン集約を終了し，削減行数を測定し出力する．クローン集約の終了条件は以下とする．

- クローンが検出されなくなる
- 選択可能なソースコードが存在しなくなる

```
public class FizBuzz {
  public static String apply(int num){
    if (num % 15 == 0) {
      return "FizzBuzz";
    } else if (num % 5 == 0) {
      return "Fizz";
    } else if (num % 3 == 0) {
      return "Buzz";
    } else {
      return String.valueOf(num);
    }
  }
}
```

(a) 入力されたソースコード

```
@Test(timeout = 4000)
public void test0() throws Throwable {
  String string0 = FizBuzz.apply(-74);
  assertEquals("-74", string0);
}

@Test(timeout = 4000)
public void test1() throws Throwable {
  String string0 = FizBuzz.apply(-9);
  assertEquals("Buzz", string0);
}

@Test(timeout = 4000)
public void test2() throws Throwable {
  String string0 = FizBuzz.apply(170);
  assertEquals("Fizz", string0);
}

@Test(timeout = 4000)
public void test3() throws Throwable {
  String string0 = FizBuzz.apply(660);
  assertEquals("FizzBuzz", string0);
}
```

(b) 出力されるテストコード

図 4　EvoSuite の入出力例

4　実装

4.1　準備

まず，対象とするソースコードに対して単体テストを自動生成する．テストの自動生成ツールとして EvoSuite を用いた [4]．EvoSuite は Java ソースコードのクラスファイルを入力とし，そのソースコードに対して可能な限り分岐を網羅する最小の単体テストを出力する．EvoSuite の入出力例を図 4 に示す．例では全ての条件分岐を網羅するテストが生成されている．EvoSuite で自動生成された単体テストはソースコードの変更を行う前と行った後での外部的振る舞いに変化がないことを確認するために用いる．

次に，対象とするソースコードに 3.1 で述べた四つの変更を加える．

抽出されたメソッドの宣言を行うクラスの用意

メソッド抽出で切り出されたメソッドを宣言するクラスを新しく用意する．このクラスは新しく用意

したパッケージ内で宣言される．このクラス内で宣言されたメソッドが元のコード片があった場所から呼び出される際には完全修飾名で呼び出される．

メソッドのアクセス修飾子の変更

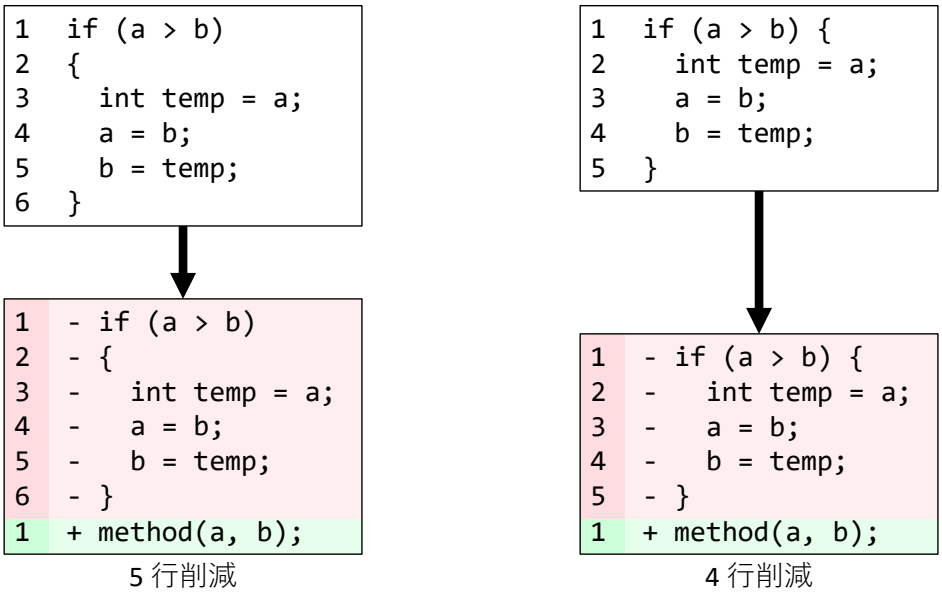
public 以外のアクセス修飾子で宣言されたメソッドは，そのメソッドが宣言されているクラスとは別のクラスやパッケージから呼び出された場合コンパイルエラーとなることがある． この問題に対して，開発者がリファクタリングを行う際には様々な対応が可能である．例えば，メソッドのアクセス修飾子を適切に変更したり，抽出されたメソッドをクローンが検出されたクラスと同じクラスに宣言したりするといった対応が考えられる．しかし，本研究ではメソッド抽出と宣言を自動で行っているため，このような対応を行うことは困難である． したがって，外部的振る舞いが変化しない範囲で，事前に全てのメソッドのアクセス修飾子を public に変更する．

ローカル変数の初期化

Java の言語規約上，初期化されていないローカル変数を参照することは禁止されている．そのため，抽出されたメソッドを呼び出すときに，初期化されていないローカル変数を引数として与えるとコンパイルエラーが発生する．開発者がリファクタリングを行う際には何らかの値で初期化してから引数として与えることが考えられる．したがって本研究では，事前に final 修飾子がついておらず，宣言時に初期化されていない全てのローカル変数を初期化する．ローカル変数が boolean 型以外のプリミティブ型の場合は “0”，boolean 型の場合は “false”，参照型の場合は “null” で初期化する．

フォーマッタの適用

コーディングスタイルの規約はプロジェクト毎に定められている．例えば，改行を多く使うような規約もあればその逆も存在する．プロジェクト内でコーディング規約に従わないコードが存在した場合，正確な削減行数を測定できない可能性がある．コードによって削減行数が異なる例を図 5 に示す． 図 3(a) ではコード片をメソッドに切り出すと 5 行削減されるのに対して，図 3(b) では 4 行の削減になる． したがって，事前にフォーマッタを利用しコーディングスタイルを統一する必要がある．本来ならプロジェクト毎のコーディング規約を再現したフォーマッタを用意するべきだが，それは困難なため，本研究では適用するフォーマッタは Eclipse の基本設定のフォーマッタで統一した．



(a) (b)

図 5　コードによって削減行数が異なる例

4.2　クローン集約

Step 1: クローン検出

クローン検出の例を図 6 に示す．クローンはブロック単位で検出する．本研究では，Eclipse JDT[15] で Statement として定義されている構文定義の内，以下をブロックとして検出する．

- Block
- DoStatement
- EnhancedForStatement
- ForStatement
- IfStatement
- SwitchStatement
- SynchronizedStatement
- TryStatement
- WhileStatement

ブロック単位で検出されたクローンは字句単位で検出されたクローンと比べると粗粒度であり検出され

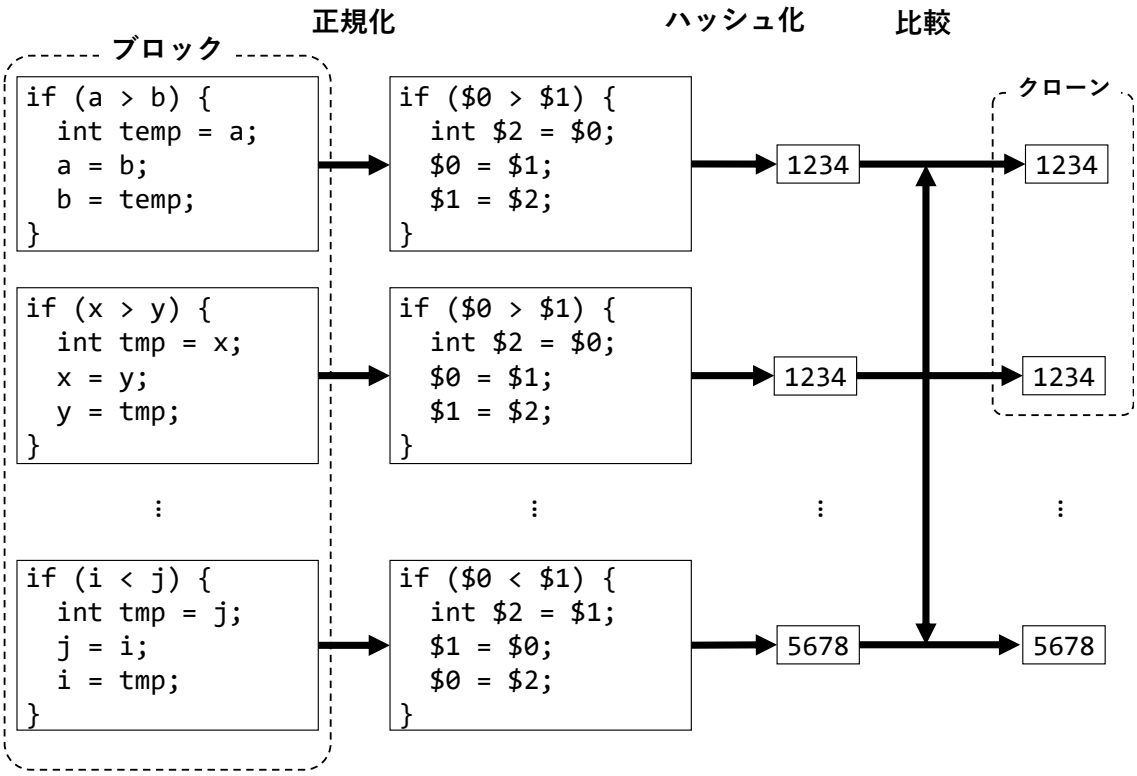


図 6　クローン検出の例

るクローンの数は少なくなるが，構造的なまとまりで検出されメソッドとして抽出することが容易であるという特徴がある [17]．本研究では，Eclipse JDT を用いて構文解析を行いソースコード中のブロックを検出する．この時，return 文を含むコード片をメソッドとして抽出することは難しいとされているため，そのようなブロックは検出を行わない [10]．

識別子の正規化を行うことでより多くのクローンを検出できるため，検出したブロックに対して以下のルールで正規化を行う．

- 変数名は “$” + “数字” で正規化する
- 同一の変数名は同一の名前で正規化する
- リテラルは全て “$” で正規化する
- 修飾された変数名は一つの変数として正規化する
- クラス名は正規化しない
- メソッド名は正規化しない

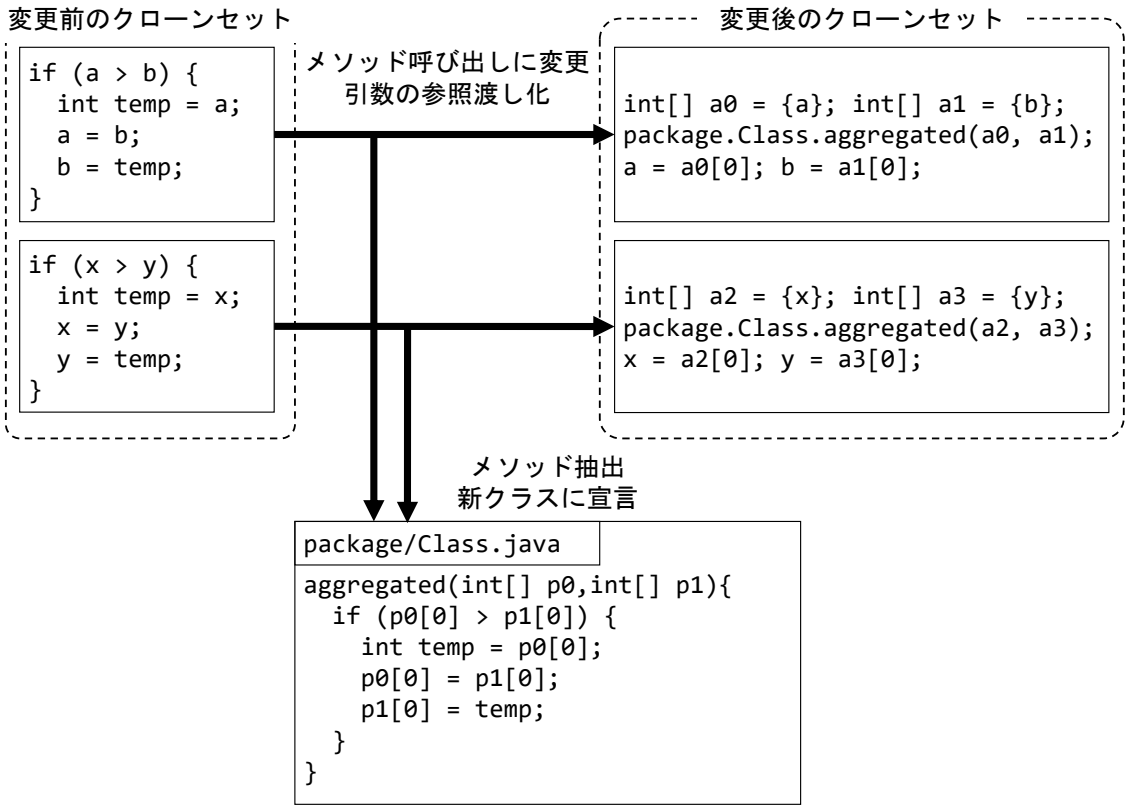


図 7　ソースコード変更の例

同一の変数名を同一の名前で正規化することで誤検出を防ぐ．クラス名とメソッド名を正規化しない理由は，クラスとメソッドはメソッドの引数として与えることができないので，クローンとして検出されるのを防ぐためである．

正規化を行った後にブロックのハッシュ化を行う．ハッシュ化には MD5[14] を利用する．MD5 は 128 ビットのハッシュ値を出力するため，ハッシュ値の衝突の可能性は十分に低いと考えられる．

最後にそれぞれのブロックのハッシュ値を比較し，同じ値のブロックをクローンとして検出する．

Step 2: ソースコード変更

Step 1 で検出されたクローンセットの一つに対してメソッド抽出を用いてソースコードに変更を加える．ソースコード変更の例を図 7 に示す．クローンをメソッドに抽出することで，それぞれのクローンをメソッド呼び出しに変更することができ，ソースコード行数を減らすことができる．抽出されたメソッドは 3.1 で作成したクラスに宣言する．このメソッドは完全修飾名で呼び出される．

また，抽出されたメソッド内で引数として与えられた変数に変更がある場合，メソッド呼び出し以降

の外部的振る舞いに影響が出る可能性がある．そこで，抽出されたメソッドに参照渡しで引数を与えることで外部的振る舞いを保つようにする．引数を参照渡しにするために配列を利用する．メソッド呼び出し前にそのメソッド内で利用される変数の型の配列を定義する．これらの配列を各変数で初期化した後にそれらの配列をメソッドに引数として与える．抽出されたメソッド内では，引数として与えられた配列の添字 0 の要素を参照する．メソッド呼び出し後に元の変数に配列の添字 0 の要素を代入する．

Step 3: 変更したソースコードのコンパイルとテスト

Step 2 で変更した結果，行数が削減したソースコードに対してコンパイルとテストを行う．テストには 4.1 で生成したテストを用いる．初めにコンパイルを行い，コンパイルに成功すると次にテストを行う．コンパイルとテストのどちらにも成功した場合，変更されたソースコードを選択可能なソースコードとして記録し，ソースコードを変更を加える前の状態に戻し，次のクローンセットに対して Step 2 から繰り返す．

コンパイルあるいはテストのどちらかに失敗した場合，そのソースコードは記録せず，成功した場合と同様にソースコードを変更を加える前の状態に戻し，次のクローンセットに対して Step 2 から繰り返す．

Step 4: 変更したソースコードの選択

Step 1 で検出されたすべてのクローンセットに対して Step 2，Step 3 の処理を行った後に Step 4 に移る．Step 4 では，選択可能なソースコードの中から最も削減行数が大きいソースコードを選択する．選択されたソースコードに対して，Step 1 から処理を繰り返す．

5 実験

本章では，提案手法を用いて行った実験と，その実験結果について述べる．

5.1 実験概要

オープンソースソフトウェア（以下，OSS）に対して提案手法を適用することで，削減可能なソースコード行数を算出する．今回は，4.2 で述べたクローン検出の際に正規化を行った場合と，正規化を行わなかった場合の二通りの実験を行った．また，算出した値に対して既存研究と削減行数の比較を行った．

5.2 実験対象

実験対象は Java 言語で記述された OSS である．本研究では，既存研究で対象となったプロジェクトの中から EvoSuite でのテスト生成に成功した jEdit，JFreeChart，JRuby の三つと，GitHub[5] 上で公開されている fastjson，Checkstyle の二つを加えた合計五つのプロジェクトを対象として実験を行った．fastjson と Checkstyle は，GitHub 上で公開されており以下の三つの条件を満たすプロジェクトの中で，スター数が上位二つのプロジェクトである．

- プロジェクトが Apache Maven[12] で管理されている
- 対象とするソースコードの行数が 50,000 行以上である
- EvoSuite でのテストの生成に成功する

Maven で管理されているプロジェクトを選んだ理由は，EvoSuite は Maven のプラグインとして提供されているため，テストの生成が容易であるからである．それに加えて，Maven を用いることでコンパイルとテストを容易に行うことができる．

対象としたプロジェクトの名前，バージョン及びソースコードの総行数を表 1 に示す．バージョン

表 1　実験対象の OSS（∗ 単位は K LoC）

プロジェクト名	バージョン	∗ 行数
jEdit	5.4.0	162
JFreeChart	1.0.19	236
JRuby	1.7.27	332
fastjson	1.2.54	50
Checkstyle	8.15	81

は，既存研究と同じプロジェクトに関しては同じバージョン，異なるプロジェクトは実験時の最新リリースのバージョンである．なお，プロジェクトのテストやチュートリアルなどのソースコードは実験対象に含まない．ソースコードの行数は 4 で説明したフォーマッタを適用した後の行数である．

5.3 実験結果

実験結果を表 2 に示す．実験結果から分かるように，検出されたクローンセットの中で，集約することで行数が減り，コンパイルとテストを通過するクローンセットは全体の 5 - 10 % ほどとそれほど多くない．また，集約可能なクローンセット数と削減行数から，一つのクローンセットあたり 10 行程度行数が削減されていることが分かる．

次に既存研究との比較を行う．比較対象は jEdit，JFreeChart，JRuby の三つのプロジェクトである．既存研究と本研究の提案手法で算出した削減行数を表 3 に示す．なお，既存研究は対象のプロジェクトに対してフォーマッタを適用していないため，フォーマッタを適用していないプロジェクトに対して提案手法で削減行数を測定した結果を記載する．

表から分かるように，既存研究と本研究とでは得られた削減行数は大きく異なる．jEdit では既存研究と比べると提案手法がより多い削減行数を算出したが，jEdit 以外では既存研究と比較して非常に少ない削減行数を算出した．

表 2　実験結果（∗ 単位は LoC）

プロジェクト名	正規化を行った場合			正規化を行わなかった場合		
	検出された クローンセット数	集約可能な クローンセット数	∗ 削減行数	検出された クローンセット数	集約可能な クローンセット数	∗ 削減行数
jEdit	397	19	213	273	17	139
JFreeChart	872	43	377	607	24	217
JRuby	802	40	326	515	44	238
fastjson	491	44	600	189	6	400
Checkstyle	100	5	36	51	2	1

表 3　算出された削減可能行数の比較結果（∗ 単位は LoC）

プロジェクト名	∗ 既存研究	∗ 提案手法
jEdit	136	362
JFreeChart	9,700	757
JRuby	2,161	222

6 考察

本章では，5 で述べた実験の考察を行う．

6.1 実験結果に対する考察

実験の結果から，集約することで行数が減り，なおかつコンパイルとテストを通過するクローンセットは検出されたクローンセット全体の内 5 - 10 % ほどであることが分かった．コンパイルに失敗する主な原因として以下が挙げられる．

- クローン間で対応する変数の型が異なる
- super を使用して親クラスを参照している
- ブロック内で例外を throw しているがその例外の対応はブロック外で行っている
- ブロック内で break や continue しているが繰り返し処理はブロック外で行っている

これらの中には，提案手法の実装方法によってはコンパイルが可能であるものも存在する．例えば，super を使用して親クラスを参照しているコードを含むクローンが同じクラス内にのみ存在した場合，抽出されたメソッドをそのクラスに宣言することでコンパイルを通過する可能性がある．

6.2 既存研究との比較に対する考察

既存研究との比較を行った結果，算出された削減可能行数が大きく異なるという結果になった．このような結果となった理由として以下の要素が考えられる．

- クローン検出法の違い
- コンパイル，テストの有無
- 削減行数の計測法

クローン検出法の違い

本研究ではブロック単位でのクローン検出を行っているが，既存研究では字句単位でのクローン検出を行っている．字句単位でのクローン検出はブロック単位でのクローン検出より多くのクローンの検出が可能である．クローン検出法の違いから，集約対象となるクローンの数に差が出ることが考えられる．

コンパイル，テストの有無

本研究では実際にクローンを集約したソースコードに対してコンパイルとテストを行い集約可能かを判定している．しかし，既存研究では行っていない．したがって，既存研究で集約可能と判断されたクローンセットは，実際は集約できない可能性があり，それが原因で削減行数に差が出ていることが考えられる．

削減行数の計測法

本研究では実際にクローンの集約を行い，集約の前後でのソースコード行数の変化から削減行数を測定している．しかし，既存研究では jEdit 以外のプロジェクトに対して実際の削減行数の測定は行っていない．したがって，実際の削減行数が既存研究の推定値とは異なっている可能性がある．

7　妥当性への脅威

本章では，本研究に対する妥当性への脅威について述べる．

本研究では 5 つの OSS を対象に実験を行った．しかし，他のプロジェクトに対して実験を行った場合，本研究で得られた結果とは異なる結果を得る可能性がある．

本研究では，ハッシュ値の比較を行うことでクローンを検出している．ハッシュ値の衝突が発生した場合，誤ったクローンが検出される可能性がある．しかし，本研究では 128 ビットのハッシュ値を出力する MD5 を用いており，ハッシュ値の衝突の可能性は十分に低い．

本研究では，ソースコードの変更前後で外部的振る舞いが変化していないことを確認するために EvoSuite によって自動生成された単体テストを用いた．しかし，テストが不十分で外部的振る舞いの変化を正確に確認できていない可能性がある．

8　あとがき

本研究では，削減可能なソースコードの行数を算出する手法として，自動でクローンの検出，集約，コンパイル，テストを行い実際に削減行数を測定する手法を提案した．また，OSS に対して提案手法を用いて削減可能行数を算出し，既存研究との比較を行った．その結果，既存研究が算出した値とは大きく異なる値が算出されるという結果となった．

今後の課題としては次のようなものが考えられる．

既存研究とのより正確な比較

5.3 で提案手法と既存手法の算出した削減可能行数を比較した．しかし，6.2 で述べたように，算出された値は検出されたクローンの数に依存している可能性がある．そのため，同じクローンセットの集合に対して提案手法と既存手法が算出する削減可能行数を比較することが必要だと考えている．

複数のリファクタリング手法の採用

本研究ではクローンの集約手法としてメソッドの抽出のみを使用した．しかし，クローンに対するリファクタリング手法はメソッドの抽出だけでなく，多くの手法が提案されている．例えば，メソッドの引き上げやテンプレートメソッドの形成などが挙げられる．これらの手法を採用することで，より正確な削減行数の算出が可能になることが期待できる．

実装の改善

6.1 で述べたように，本研究ではコンパイルエラーとなったが，実装を工夫することでコンパイルが可能になるようなクローンセットが存在すると考えられる．このようなクローンセットに合わせて実装を改善することで，より正確な削減行数の算出が可能になることが期待できる．

謝辞

本研究を行うにあたって理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝を申し上げます．

本研究の全過程を通して熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝を申し上げます．

本研究に関して有益かつ的確なご助言を頂きました，柗本真佑助教に深く感謝を申し上げます．

本研究を進めるにあたって様々な形で励ましやご協力を頂きました，楠本研究室の皆様に心より感謝を申し上げます．

本研究に至るまでの間，講義，演習，実験などで日々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より御礼申し上げます．

最後に，22 年に渡り様々な面で支援していただき，辛いときは支えてくれた両親に心より感謝申し上げます．

参考文献

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, Sep 2007.

[2] J. R. Cordy and C. K. Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pp. 219–220, June 2011.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series. Pearson Education, 1999.

[4] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416–419, 2011.

[5] GitHub. https://github.com/.

[6] Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, Takashi Fujinami, and Takashi Hoshino. Correlation analysis between code clone metrics and project data on the same specification projects. In *Proc. of the 12th International Workshop on Software Clones*, pp. 37–43, 2018.

[7] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105. IEEE Computer Society, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic 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.

[9] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, Vol. 30, No. 5, pp. 187–196, Sep 2005.

[10] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension*, pp. 33–42, May 2003.

[11] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[12] Apache Maven. https://maven.apache.org/.

[13] Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley Publishing, 1st edition, 1996.

[14] R. Rivest. The md5 message-digest algorithm, Apr 1992.

[15] Java Development Tools. https://www.eclipse.org/jdt/.

[16] Norihiro Yoshida, Takuya Ishizu, Bufurod Edwards, III, and Katsuro Inoue. How slim will my system be?: Estimating refactored code size by merging clones. In *Proceedings of the 26th Conference on Program Comprehension*, pp. 352–360, 2018.

[17] 堀田圭佑, 楊嘉晨, 肥後芳樹, 楠本真二. 粗粒度なコードクローン検出手法の精度に関する調査. 情報処理学会論文誌, Vol. 56, No. 2, pp. 580–592, Feb 2015.

[18] 井上克郎, 神谷年洋, 楠本真二. コンピュータソフトウェア, Vol. 18, No. 5, pp. 529–536, Sep 2001.

[19] 石津卓也, 吉田則裕, 崔恩瀞, 井上克郎. コードクローンに対するリファクタリング可能性に基づいた削減可能ソースコード量の調査. Technical Report 7, 大阪大学, 名古屋大学, 奈良先端科学技術大学院大学, 大阪大学, Nov 2017.

[20] 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, 情報・システム = The IEICE transactions on information and systems (Japanese edition), Vol. 91, No. 6, pp. 1465–1481, Jun 2008.

[21] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. コンピュータソフトウェア, Vol. 28, No. 4, pp. 43–56, Oct 2011.