複数メソッド上に解離したコードクローンの検出手法

高　良多朗[†]　　堀田　圭佑[†]（正員）
肥後　芳樹[†]（正員）　　井垣　　宏[††]（正員）
楠本　真二[†]（正員）

A Technique to Detect Code Clones Scattered on Multiple Methods

Ryotaro KOU[†], *Nonmember*, Keisuke HOTTA[†], Yoshiki HIGO[†], Hiroshi IGAKI[††], and Shinji KUSUMOTO[†], *Members*

[†] 大阪大学大学院情報科学研究科
〒 565–0871 大阪府吹田市山田丘 1–5
Graduate School of Information Science and Technology, Osaka University
Yamadaoka 1–5, Suita–Shi, Osaka, 565–0871 Japan
Email: { r-kou,k-hotta,higo,kusumoto } @ist.osaka-u.ac.jp
[††] 大阪工業大学情報科学部
〒 573–0196 大阪府枚方市北山 1–79–1
Faculty of Information Science and Technology, Osaka Institute of Technology
Kitayama 1–79–1, Hirakata–Shi, Osaka, 573–0196 Japan
Email: hiroshi.igaki@oit.ac.jp

あらまし　従来のコードクローン検出手法では，ソースコード上においてある程度以上の大きさでまとまったコードが重複している場合に，コードクローンとして検出される．しかし，重複したコードが複数のメソッドに解離して存在していた場合には検出できない場合が多い．本論文では，複数のメソッド上に解離して存在するコードクローンを検出する手法を提案する．

キーワード　ソフトウェア保守, ソースコード解析, コードクローン

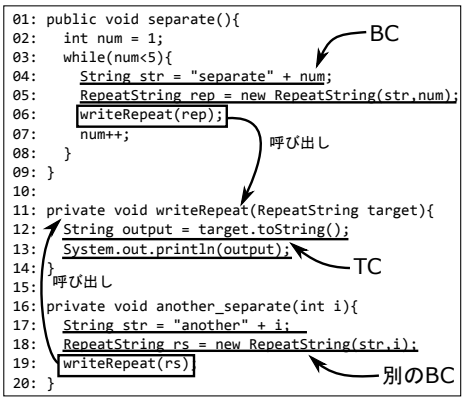
1. まえがき

近年，ソフトウェア保守作業の支援を目的とした手法としてコードクローン検出が注目されている [1]．コードクローンとはソースコード上に存在する同一または類似したコード片を指し，コード片の複製や定型処理など様々な原因で発生すると報告されている [2]．

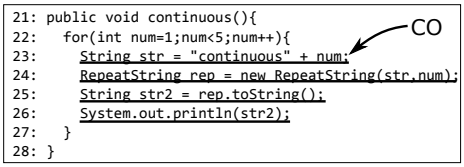
一般的に，コードクローンはソフトウェアの保守性を悪化させる可能性を持つと言われている．このような背景から，ソースコード上に存在するコードクローンを特定する技術である，コードクローン検出手法に関する研究が行われている [3]．

既存手法の多くは偶然類似したコード片の検出を防ぐため，類似したコード片の規模が一定の閾値を超えるならばコードクローンとして検出する．ところが，類似箇所がメソッド呼び出しによって分断されている場合，コード片が十分な規模にならないためにコードクローンとして検出されない可能性がある．

本研究では複数メソッド上に解離したコード片を一連のコードクローンとして特定する手法を提案する．既存手法との比較による評価実験の結果，提案手法は既存手法では検出できないコードクローンを検出可能であることを確認した．



(a) 解離したコード片



(b) 連続したコード片

図 1　解離したコードクローンの例
Fig. 1　An example with the clone of pattern 1

2. 提案手法

2.1 用語の定義

本研究では，あるメソッド呼び出しの直前のコード片と呼び出されるメソッドの先頭のコード片を一組とみなし，この組を “解離したコード片” と定義し，解離したコード片と同様の処理を行うが 1 つのメソッド上に連続して記述されたコード片を “連続したコード片” と定義する．また，解離したコード片を含むコードクローンを解離したコードクローンと定義し，以下の 2 つのパターンに分類する．

パターン 1　解離したコード片と連続したコード片からなるコードクローン

パターン 2　2 つの解離したコード片からなるコードクローン

解離したコードクローンの例を図 1 に示す．図 1(a) ではある処理をメソッド separate とメソッド writeRepeat の組合わせで実装している一方で，図 1(b) では図 1(a) と類似処理を単一のメソッド continuous で実装している．これらのコード片はパターン 1 に相当する．一方で，図 1(a) ではメソッド writeRepeat の呼び出しがメソッド separate とメソッド another_separate の二

```
01: public void adapter(){
02:   int num1 = 15;
03:   int num2 = 25;
04:   int num3 = num1 * num2;
05:   Divisor divi = new Divisor(num3);
06:   adaptee(divi.getArray());
07: }
08:                          2つのメソッド呼び出しが存在
```

(a) 正規化前

```
01: public void adapter(){
02:   int num1 = 15;
03:   int num2 = 25;
04:   int num3 = num1 * num2;
05:   Divisor divi = new Divisor(num3);
06:   int[] var0 = divi.getArray();
07:   adaptee(var0);
08: }                          メソッド呼び出しを切り離す
```

(b) 正規化後

図 2　1 文中に複数のメソッド呼び出しを含むコードの例
Fig. 2　An Example with the code including multiple methods.

箇所で行われているが，どちらもメソッド呼び出しの直前のコード片が類似している．これらのコード片はパターン 2 に相当する．

以降，本稿では以下の用語を使用する．

BC(BeforeCall)　メソッド呼び出しの直前のコード片

TC(TopofCallee)　呼び出し先メソッドの先頭のコード片

CO(COntiguous)　*BC* および *TC* が同順で連続しているコード片

2.2　解離したコードクローンが発生する原因

本研究の検出対象である解離したコードクローンは一連の処理を行うコード片が複数のメソッド上に存在するものである．このようなコード片が発生する原因はリファクタリングの漏れが考えられる．ソースコード上に長いメソッドや複雑なメソッドが存在する場合はそれらの一部を切り出して別のメソッドにする，メソッド抽出と呼ばれるリファクタリングを行うことがある．

このとき，リファクタリング対象にしたメソッドとコードクローンの関係にあるメソッドが存在しているにもかかわらず，開発者がそのようなメソッドを見落とした場合に解離したコードクローンが発生することがある．この場合，メソッド抽出を適用したメソッドと適用していないメソッドで一致するコード長が短くなるため既存手法での検出が難しくなるが，提案手法を用いることでこれらのメソッドを検出できる．

また，全てのコードクローンを見落とすことなくリファクタリングを適用したとしても，適用前と類似するコード片が後から追加される可能性が考えられる．この場合でも提案手法ならば容易に特定可能である．

```
01: public void separate(){
02:   for(int num=1;num<5;num++){                BC
03:     String str = "separate" + num;
04:     writeWithNum(str, num);      メソッド呼び出し
05:   }
06: }
07: private void writeWithNum (String str, int num){
08:   write(str);
09:   System.out.println(num);  メソッド呼び出し
10: }
11: private void write(String str){
12:   System.out.println(str);
13: }                                            TC
```

図 3　複数回のメソッド呼び出しに対応した例
Fig. 3　An example with adapting to multiple method call.

提案手法は最初から実装が異なるようなソースコードのほか，上述の例のようにリファクタリングなどが原因で一貫性を失ったソースコードに対して，類似処理を行うコード片を検出するために利用できる．したがって，本手法はソースコードの実装方法を統一したい場合などに有効である．

2.3　手法の概要

提案手法は複数のメソッド上に解離したコードクローンを検出する．コードクローンの検出は字句単位で行い，コード片の合計一致字句長が閾値以上であればそれらのコード片をコードクローンとして提示する．図 1 の例では図 1(a) の 4-5 行目と 12-13 行目に対して，図 1(b) の 23-26 行目がパターン 1 の解離したコードクローンとして提示される．一方で，図 1(a) では 6 行目と 19 行目で writeRepeat が呼び出されているため，4-5 行目および 17-18 行目に 12-13 行目を合わせたコード片の組がパターン 2 の解離したコードクローンとして提示される．

2.4　手法の流れ

提案手法は対象ソフトウェアのソースコードを入力に受け取り，解離したコードクローンを検出する．手法の流れは以下に示す 5 つのステップで構成される．

ステップ 1

ソースコード中の各メソッド呼び出しについて，呼び出される可能性のあるメソッドをすべて特定する．

ステップ 2

以下の法則に従いソースコードを正規化する．

- 1 文中に複数のメソッド呼び出しが含まれる場合，1 文中に現れるメソッド呼び出しの数が高々1 つになるようにソースコードを書換える．図 2 の例においては，図 2(a) の 6 行目中に 2 つのメソッドが存在しているため，図 2(b) の 6-7 行目のように書換える．
- 変数名やリテラルをすべて特殊文字に置換する．

ステップ 3

各メソッド呼び出しについて，*BC* および *TC* を 1 行ずつ抽出して組にする．図 3 のように呼び出されるメソッドの先頭の文が別のメソッドを呼び出す場合，さらに呼び出されるメソッドを参照して *TC* を抽出する．

ステップ 4

ステップ 3 で抽出した *BC* と *TC* が連続して現れる箇所を *CO* として特定する．この組合せがパターン 1 に相当する．特定した *CO* について，*BC* と *TC* およびそれらの近傍を調べて，連続して一致する字句長の合計が閾値を超えているならば検出結果に含める．

ステップ 5

ステップ 3 で抽出した *BC* と *TC* に対して，*BC* 同士，*TC* 同士がそれぞれ一致するような組合せを特定する．この組合せがパターン 2 に相当する．特定後はステップ 4 と同様に一致字句長を調べ，閾値以上ならば検出結果に含める．

3. 評価実験

既存手法での検出が困難なコードクローンを提案手法で検出可能かを評価するために，コードクローン検出ツールである CCFinderX (注1) との比較実験を行った．

3.1 実験方法

はじめに，実験対象ソフトウェアのソースコードに対して最小一致字句長すなわち閾値を 30 から 60 まで 5 刻みに設定して提案手法を適用する．次に，提案手法で正規化したソースコードに対して CCFinderX を適用する．最後に，提案手法が検出した解離したコードクローンを構成する各コード片が CCFinderX で検出されたかを調査する．

調査においては，提案手法のステップ 3 で抽出した *BC* および *TC* と，ステップ 4 で特定した *CO* が CCFinderX で検出されたコードクローンのいずれかに含まれる場合は同じコード片を検出したとみなし，それらの行を含むコードクローンが存在しなかった場合は同じコード片を検出しなかったとみなす．ここで，*BC* と *TC* は連続したコード片ではないため，調査は *BC* 側と *TC* 側に分けて行う．したがって，比較結果は以下の 4 パターンに分類される．

（1）*BC* と *TC* が共に検出される場合
（2）*BC* のみが検出される場合
（3）*TC* のみが検出される場合
（4）*BC* と *TC* のどちらも検出されない場合

本実験では，オープンソースソフトウェアの Apache Ant 1.9.4 (注2) を実行対象に用いた．本ソフトウェアの行数は 218,415，ファイル数は 856 である．

3.2 実験結果

パターン 1 の実験結果を表 1 に，パターン 2 の実験結果を表 2 に示す．結果より，最低でも 80%以上の解離したコードクローンが CCFinderX で検出不可能であることが読み取れる．解離したコードクローンの検出数については，パターン 1 に比べるとパターン 2 が非常に多く，パターン 1 とパターン 2 の両パターンにおいて，*BC* よりも *TC* の方が既存手法で検出されやすい傾向にあることが読み取れる．

4. 考察

始めに，大半の解離したコードクローンが既存手法で検出できなかった理由としては，*BC* および *TC* は小規模なものが多いことが考えられる．提案手法では *BC* と *TC* の字句長を足して閾値以上ならば検出結果に含めるが，一方，既存手法では片方の字句長のみで閾値を満たす必要がある．特に，検出メソッドが小規模ならばメソッド全体で一致した場合でも一致字句長が小さくなるため，既存手法では検出結果から除外され易いと考えられる．

次に，パターン 1 に比べてパターン 2 の検出数が膨大になっている理由としては，呼び出されるメソッドの候補が多数存在した場合に起こる組合せ数の増加が考えられる．ある抽象メソッドをオーバーライドした

表 1　パターン 1 の比較結果
Table 1　An experimental result of pattern 1

閾値	総数	*TC* と *BC*	*BC* のみ	*TC* のみ	検出なし
30	300	5	1	0	292
35	133	5	1	0	127
40	78	5	1	0	72
45	62	4	1	0	57
50	42	0	1	4	37
55	39	0	1	4	34
60	30	0	1	4	25

表 2　パターン 2 の比較結果
Table 2　An experimental result of pattern 2

閾値	総数	*TC* と *BC*	*BC* のみ	*TC* のみ	検出なし
30	68,995	218	1,314	2,233	65,230
35	34,158	182	820	2,193	30,963
40	14,461	147	235	1162	12,917
45	9,543	129	137	909	8,368
50	7,052	111	44	772	6,125
55	6,050	107	43	694	5,206
60	3,428	92	50	269	3,027

（注1）：CCFinderX,http://www.ccfinder.net/

（注2）：apache ant 1.9.4, http://ant.apache.org/

メソッドが a 個存在し，そのメソッドが b 箇所で呼び出される場合，組合せの数は a × b 通りとなる．このメソッド呼び出しから複数の *BC* および *TC* が抽出された場合，抽出された分だけパターン 2 の組合せが生じる．特に，ファイル操作や入出力処理など，汎用的に用いられるメソッドをオーバーライドして実装していた場合，組合せの数が膨大になり易いと考えられる．

最後に，*BC* と比較して *TC* の方が既存手法で検出されやすい理由としては，上述のように，ファイル操作や入出力処理などを行うメソッドは具体的な内容に関わらず類似した実装になり易く，メソッドの大半または全体がコードクローンとして検出されやすいためだと考えられる．先の考察と合わせて，*TC* はオーバーライドされた分だけ組合せが増加し，それぞれが通常のコードクローンとして検出されやすいと考えられる．

5. 実験の妥当性について

本実験では，提案手法と CCFinderX との比較を行ったが，提案手法で検出されたコードクローンのみを調査しており，CCFinderX で検出されたコードクローンが提案手法で検出できるかは未調査である．提案手法はメソッド呼び出しを境に解離したコードクローンのみを検出対象としているため，単純に連続したコード片同士は検出対象としていない．したがって，本実験は提案手法が既存手法での発見が困難なコードクローンを検出するために有用であることを示すのみで，既存手法に対する全面的な優位性を示すものではない．同じく，実験対象に選択したソフトウェアはクラスの継承を多用しているためにパターン 2 の検出数が膨大になったが，実験対象が継承をほとんど行わないようなソフトウェアであった場合，パターン 1，パターン 2 共に検出数がどれほどになるかは未知である．

6. あ と が き

本研究では，複数のメソッド上に解離して存在する不連続なコード片を一連のコードクローンとして検出する手法を提案した．

提案手法の評価実験として，単一のソフトウェアに対する既存手法の検出結果と提案手法の検出結果を比較したところ，提案手法から検出されたコード片の多数は，既存手法での検出が困難であることを確認した．

今後の課題として，提案手法によって検出されるコードクローンと一般的なコードクローンの特徴の違いの調査などが挙げられる．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号 : 25220003)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の支援を受けて行われた．

文　　献

[1] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” Information and Software Technology, vol.55, no.7, pp.1165–1199, 2013.

[2] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier, “Clone detection using abstract syntax trees,” Proc. International Conference on Software Maintenance, pp.368–377, 1998.

[3] 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．

（平成 xx 年 xx 月 xx 日受付）

Abstract An existing code clone detection techniques identify the massed and duplicated code fragments as code clone. However, it is difficult for these techniques to identify duplicated code fragments scattered on multiple methods. This paper proposes a technique which identifies code clones scattered on multiple methods.

Key words Software Maintenance, Source Code Analysis, Code Clone