情報処理学会論文誌 Vol.55 No.2 959–970 (Feb. 2014)

繰返しコードの進化に関する調査

今里 文香[1,a)] 佐々木 唯[1,b)] 肥後 芳樹[1,c)] 楠本 真二[1,d)]

受付日 2013年5月16日, 採録日 2013年11月1日

概要：コードクローンの進化に関する既存研究によって，コードクローン中のある 1 つのコード片が修正された場合に，そのコード片と類似した他のコード片に対してもしばしば同様の修正が加えられることが明らかとなっている．そのため，コードクローンはソフトウェアの修正に要する作業量の増大をもたらすとされる．一方でコードクローンの一種として，類似した記述が連続して出現する繰返しコードが存在する．また，繰返しコードには，大きさが小さいものが多く存在することが報告されている．しかし，コードクローン検出ツールでは通常は小さいコードクローンは検出されない．したがって，多くの繰返しコードは，既存の検出手法では検出することが難しい．著者らは，繰返しコードについても，コードクローンのように，複数のコード片に対して同様の修正が加えられることがあるのではないかと考えた．そこで本研究では，ソースコード中に含まれる繰返しコードの変遷を追跡し，その進化傾向について調査した．調査の結果，繰返し回数が少ないほど，すべての繰返し要素に対して同様の修正が加わりやすいことなどが明らかになった．

キーワード：コードクローン，ソフトウェア保守，バージョン管理システム

A Study about Evolution of Repeated Code in Program Source Code

Ayaka Imazato[1,a)] Yui Sasaki[1,b)] Yoshiki Higo[1,c)] Shinji Kusumoto[1,d)]

Received: May 16, 2013, Accepted: November 1, 2013

Abstract: Studies on evolution of code clone have confirmed that when one code fragment is modified, other code fragments that are similar to or identical to it also require the same modifications frequently. Hence it is said that code clone increases workload for source code modifications. On the other hand, as a kind of code clone, there is a plenty of repeated code in source code. Repeated code means consecutive code fragments that consist of the same instructions. It has been revealed that a lot of repeated code consists of a small number of tokens. But generally, existing code clone detection methods exclude code clone composed of a small number of tokens as a detection target because they are not useful. Therefore it is difficult for existing code clone detection methods to detect repeated code composed of few tokens. In this study, we tracked repeated code in source code and investigated its evolution. As a result, we revealed that, the less elements repeated code has, the more likely all the elements of it be modified simultaneously.

Keywords: code clone, software maintenance, version control system

1. はじめに

コードクローンは，ソースコードの修正にともない，形を変えながら様々な進化をとげていく [1]．コードクローン中のある 1 つのコード片が修正された場合に，そのコード片と類似した他のコード片に対しても同様の修正が加えられることがある．このため，コードクローンの存在はソフトウェアの修正に要する作業量を増大させるおそれがある．さらに，類似した複数のコード片に同様の修正を加える際，本来修正を加えるべきであったコード片に対して修正漏れが生じることも少なくない [2]．修正漏れが生じた場合，その箇所には新たなバグが発生することとなるため，

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
a) i-ayaka@ist.osaka-u.ac.jp
b) s-yui@ist.osaka-u.ac.jp
c) higo@ist.osaka-u.ac.jp
d) kusumoto@ist.osaka-u.ac.jp

```
MenuItem iSaveMenuItem = null;
MenuItem iMenuSeparator = null;
MenuItem iShowLogMenuItem = null;
```

(a) 繰返し要素を構成する文が 1 つ

```
comparator1 = new DnComparator();
cb1.schemaObjectProduced( this, "2.5.13.0", comparator1 );
comparator2 = new DnComparator();
cb2.schemaObjectProduced( this, "2.5.13.1", comparator2 );
```

(b) 繰返し要素を構成する文が複数

図 1 繰返しコード例

Fig. 1 Example of repeated code.

コードクローンはソフトウェアの保守性を低下させる原因になりうる [2], [3]．このような問題の解決を目的として，コードクローンに着目した修正支援手法が数多く提案されている [4], [5]．

一方，ソースコード中には，類似した記述が連続して出現するコード片が多く存在する [6], [7], [8]．代表的な例として，if-else 文や，case 文などが連続して記述されやすい．本研究では，このような連続して類似したコード片が繰り返されている部分のことを繰返しコードと呼ぶ．繰返しコードには，主に 2 つの発生要因がある [6]．1 つ目は，言語依存による発生である．たとえば，変数宣言文やメソッド呼び出し文，case 文などは，プログラミング言語の特性上，しばしば連続して記述されることがある．2 つ目は，アプリケーション依存による発生である．アプリケーションフレームワーク（e.g., データベース接続を行うためのフレームワーク）はしばしば記述形式が慣用化されている．こういったソースコード中に類似した記述が連続して出現した場合に，繰返しコードが発生することがある．また，繰返しコードには，大きさが小さいものが多く存在することが報告されている [6]．しかし，コードクローン検出ツールでは通常は小さいコードクローンは検出されない．したがって，多くの繰返しコードは，既存の検出手法では検出することが難しい．このような繰返しコードについても，コードクローンのように，1 つの繰返しコードを構成する各繰返し要素に対して同様の修正が加えられることが起こるのではないかと著者らは考えた．本研究では，オープンソースソフトウェアのソースコード中に含まれる繰返しコードに対して，その変遷を調査し，加えられた修正の内容などの情報から，繰返しコードの進化傾向を明らかにした．そして調査結果をもとに，繰返しコードに対する有用な修正支援としてどのようなものがあげられるか考察を行った．

2. 繰返しコード

2.1 定義

本研究では，ソースコード中において，連続して類似したコード片が繰り返されている部分を，**繰返しコード**と呼ぶ．繰返しコードは，コードクローンの一種である．なお，繰返しコードの識別は Sasaki らが提案している手法 [8] をもとに行う．Sasaki らの手法では，繰返しコードの識別は，ソースコードから構築される AST（Abstract Syntax Tree）の構造をもとに行われる．AST 上の兄弟ノードは，ソースコード上の出現順序を保持している．そこで，AST 上の連続する兄弟ノードについて，それぞれを根とする部分木が同形であれば，そのノード以下の部分木を構築するコード片を繰返しコードと見なす．部分木が同形であるための条件は，部分木に含まれるそれぞれのノードが示す文が類似していることである．文の類似性の判定基準については次節で述べる．

たとえば**図 1** (a) のソースコードは，3 つの変数宣言文からなる繰返しコードである．このとき，繰返しの単位となっているコード片を**繰返し要素**と呼ぶこととする．この例の場合，繰返し要素は各変数宣言文であり，繰返し要素数は 3 である．また，図 1 (b) のソースコードは，繰返し要素が代入文とメソッド呼び出し文，繰返し要素数が 2 の繰返しコードである．

本研究で用いる繰返しコードは，コード片を構成している文構造に着目し，一部を除き（提案手法では，メソッドおよび代入文のみ，文の類似判定のために変数名を考慮する），変数名やリテラルといった文の詳細は考慮していない．このように変数名やリテラルの差異を吸収することで，文の詳細が異なっていても，似た構造が繰り返されていれば，繰返しコードとして検出することができる．

なお，コードクローン検出手法と本研究で用いる繰返しコード検出手法の主な相違点は以下のとおりである．

- コードクローン検出では，連続したコード片・非連続のコード片のどちらも検出可能である．一方で，繰返しコード検出では連続したコード片のみを検出する．
- コードクローン検出では，通常は検出対象のコードクローンの大きさに閾値が設けられており，小さいものは検出されない．一方で，繰返しコード検出には，そのような大きさによる制限がない．

2.2 文の類似性

本研究では，文の種類が同じであれば類似した文と見なす．ただし，次の 5 種類の文については，類似していると判定するため以下の条件も考慮する．なお，これらの条件は，Sasaki らの手法 [8] によって定義されているものをそのまま利用している．また，本研究で実装したツールは解析対象を Java 言語に限定しているため，ここでは Java における文の類似性についてのみ考える．Java におけるこれらの文の記述形式を**表 1** に示す．

表 1 文の種類と Java における記述形式
Table 1 Kinds of statements and symbolic conventions in Java.

文の種類	記述形式
メソッド呼び出し文	object.[<Type,... >]method(...); [<Type,... >]method(...);
変数宣言文	{modifier} type name; {modifier} type name = *Expression*;
代入文[*1]	name = *Expression*;
return 文	**return** *Expression*;
throw 文	**throw** *Expression*;

メソッド呼び出し文

メソッド呼び出し文は，表 1 に示すように，メソッドが呼び出されるオブジェクトを指す変数名やクラス名が明記される場合とされない場合がある．この記述形式が等しく，かつ，メソッドの名前が完全一致であれば類似する文と見なす．また，型引数については考慮しない．

変数宣言文

変数宣言文は，初期化を行わない場合は，型名が一致すれば類似する文と見なす．初期化を行う場合は，型名が一致し，かつ，右辺に現れる式の種類が同じであれば，類似する文と見なす．なお，式の種類については，後述する．また，修飾子については考慮しない．

代入文

左辺に現れる名前が類似しており，かつ，右辺に現れる式の種類が同じであれば，類似する文と見なす．ただし，名前の類似判定については，既存研究 [9] で識別子名の類似性を調べるために用いられている方法を本研究でも用いる．既存研究 [9] では，識別子名が類似しているかどうか判断するため，識別子名をキャメルケースや記号で分割し，分割数や分割後の単語がどの程度一致するか調べる方法が用いられている．キャメルケースとは，文字列が複数の単語から構成される場合に，単語の区切りが分かるように各単語の先頭の文字に大文字を用いる表記方法である．たとえば，`CamelCase` は `Camel` と `Case` の 2 つの単語から構成される．2 つの文字列をキャメルケースで分割したとき，以下の条件のいずれかを満たす場合に，その文字列は類似していると見なす．

- 分割後の単語の数（分割数）が 1 語である．たとえば x と y は類似した文字列である．
- 分割数が同じであり，一カ所以上同じ単語が出現する．たとえば newStrTmp と oldStrMsg は類似した文字列である．
- 最初か最後の単語が一致する．たとえば srcPixels と dstInPixels は類似した文字列である．

return 文・throw 文

オペランドに現れる式の種類が同じであれば，類似する文と見なす．

式の種類

本研究で用いる繰返しコードの検出手法 [8] では，JDT[*2]によって生成される AST をもとに，コード片の類似判定を行っている．JDT では，代入文や変数宣言文，return 文，throw 文の *Expression*（表 1 参照）について，その詳細な種類を取得することができる（e.g., 文字列，数値，変数または定数，インスタンス生成，単項演算式，二項演算式）．

3. 調査手法

3.1 概要

ソースコード中に含まれる繰返しコードの進化について調査する手法を説明する．調査における入出力は，以下のとおりである．

入力 プロジェクトのリポジトリ
出力 プロジェクトのソースコードに含まれる繰返しコードの進化に関するデータ

出力するデータの詳細を，以下に示す．

- 繰返しコードの繰返し要素数の遷移
- 繰返しコードを構成する文の種類の遷移
- 繰返しコードを構成する繰返し要素のトークン数の遷移
- 繰返しコードの存在期間
- 存在期間中に繰返しコードに行われた修正の回数

3.2 調査対象とする繰返しコード

本研究では，ソースコード中に含まれるすべての繰返しコードを調査対象とする．ただし，繰返しコードの中にさらに繰返しコードが含まれる場合は，包含されている繰返しコードについては調査対象外とする．**図 2** は，while 文および while 文に含まれるメソッド呼び出し文を 1 つの繰返し要素とする，繰返し要素数 2 の繰返しコードである．この繰返しコードの各繰返し要素の中にはさらに，メソッド呼び出し文を 1 つの繰返し要素とする，繰返し要素数 3 の繰返しコードが含まれている．したがって，図 2 には，while 文と while 文に含まれるメソッド呼び出し文から構成される繰返しコードが 1 つ（1〜10 行目），メソッド呼び出し文から構成される繰返しコードが 2 つ（2〜4 行目および 7〜9 行目），合計 3 つの繰返しコードが存在する．本研

[*1] Java では，代入文はオペレータが “=” である式文として定義されている．したがって本研究では，式文のうち演算子が “=” であるものを代入文として扱う．

[*2] 統合開発環境 Eclipse の Java 開発ツール．JDT は Java ソースコードのコンパイルや実行，コード補完，デバッグなどの機能を提供する．

```
 1: while(...){
 2:   a.hoge();
 3:   b.hoge();
 4:   c.hoge();
 5: }
 6: while(...){
 7:   d.hoge();
 8:   e.hoge();
 9:   f.hoge();
10: }
```

図 2 包含関係にある繰返しコード
Fig. 2 Inclusion relation of repeated code.

究では，これらの繰返しコードのうち，包含されている繰返しコード，すなわちメソッド呼び出し文から構成される 2 つの繰返しコードは調査対象外とする．

繰返しコードの中にさらに繰返しコードが含まれる場合，包含されている繰返しコードは，包含している繰返しコードの繰返し要素数と同じ数だけ存在する．さらに，それらは互いに類似した構造を持つ．そのため，もし包含されている繰返しコードを調査対象に含めると，類似した構造を持つ繰返しコードについて調査結果が重複してしまうことになる．特に，包含している繰返しコードの繰返し要素数が多い場合は，包含されている繰返しコードが全体の調査結果へ与える影響が大きくなってしまうと著者らは考えた．したがって，本研究では，包含されている繰返しコードは調査対象から除外している．

ただし，繰返しコードに対する修正支援について言及した場合，繰返しコード中に含まれる繰返しコードも考慮に入れた方が，有用な修正支援を行うことができる場合がある．たとえば，図 2 の 2 行目の `a.hoge();` に変更が加えられたとき，包含されている繰返しコードについて考慮していれば，繰返しコードを構成している他の要素（3 行目の `b.hoge();` および 4 行目の `c.hoge();`）に対しても修正支援が可能である．しかし，本研究のように包含されている繰返しコードについて考慮していない場合は，こういった修正支援を行うことができない．

3.3 調査手順

調査手順は，以下の 3 つの STEP からなる．

STEP1 ソースコードに修正が行われたリビジョンのみからなるリビジョン列 R を抽出．

STEP2 R 内の隣り合う各リビジョン間における，繰返しコードの進化を調査．

STEP3 各繰返しコードの進化に関するデータを整理．

この手順を図 **3** に示す．

各ステップの詳細は以下のとおりである．

STEP1 リビジョン列 R の抽出

プロジェクトの全リビジョンのうち，ソースコードに修正が行われたリビジョンのみからなるリビジョン列 $R = \{r_1, r_2, \cdots, r_n\}$ を抽出する．ただし，R 中の各要素の添え字は，その要素の R 中における順番を示す．たとえば，r_i は，R 中の i 番目の要素を表す．

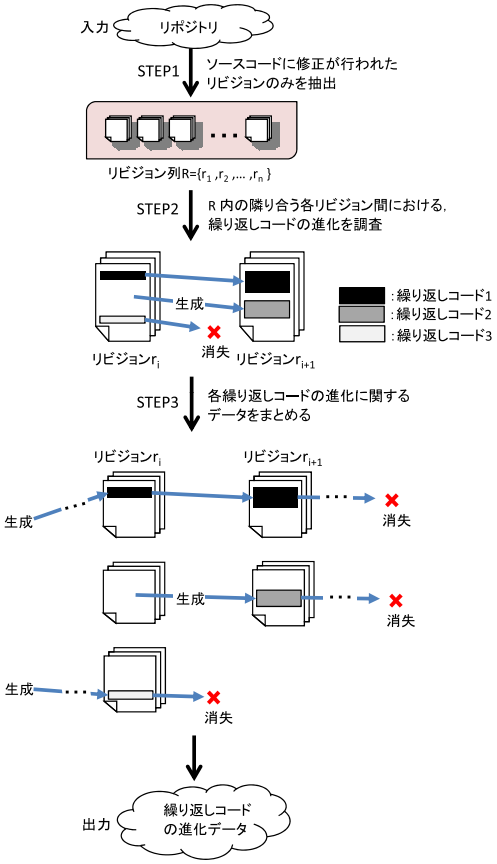


図 3 手順の流れ
Fig. 3 Overview of the proposed method.

プロジェクトの各リビジョンに対して，そのリビジョンの中でソースコードに修正が行われているかどうか調べる．その後，ソースコードに修正が行われているリビジョンを，リビジョン番号が小さいものから順にリビジョン列 R に追加する．

STEP2 隣り合うリビジョン間における繰返しコードの進化の調査

STEP2 は以下の 3 つの STEP からなる．

STEP2A 繰返しコードに対する修正の有無の調査

まずリビジョン r_i，r_{i+1} 間の差分をとり，リビジョン r_i における修正箇所を特定する．リビジョン r_i に含まれる繰返しコードについて，修正箇所と一部でも位置が重なっていたら，その繰返しコードには修正があったと見なす．

ただし，リビジョン r_i で行が挿入された場合は，まず挿入が行われた箇所の前後の行を特定する．そして，前後の行のうちいずれかがリビジョン r_i の繰返しコードに含まれるならば，その繰返しコードに修正があっ

たと見なす．

STEP2B 繰返しコードの修正後における構造の調査

リビジョン r_i で繰返しコードに対して行われた修正箇所について，差分情報からその修正後であるリビジョン r_{i+1} における位置を取得する．修正箇所の変更後の位置がリビジョン r_{i+1} の繰返しコードに含まれるならば，その繰返しコードは修正後も繰返しコードであると見なす．この判定アルゴリズムを，Algorithm1 および 2 に示す．

Algorithm1 および 2 は，リビジョン r_i に含まれる繰返しコード c が修正後のリビジョン r_{i+1} でも繰返しコードであるかどうか判定するアルゴリズムである．Algorithm 中の C_{r_i} は，リビジョン r_i に含まれる繰返しコードの集合を表す．Algorithm1 の入力変数を以下に示す．

- c
  リビジョン r_i に含まれる 1 つの繰返しコード
- $C_{r_{i+1}}$
  リビジョン r_{i+1} に含まれる繰返しコードの集合

また，Algorithm1，2 中の各関数の説明は以下のとおりである．

- *getModifiedLines*
  引数として繰返しコードをとり，その繰返しコード中の修正が行われた行の行番号集合を返す．
- *getAfterLines*
  引数として行番号集合をとり，その行番号集合の修正後の行番号集合を返す．
- *getFileName*
  引数として繰返しコードをとり，その繰返しコードが記述されているファイル名を返す．
- *judgeCondition*
  引数は以下のとおりである．
  - L_{after}
    繰返しコード c 中の修正が行われた行の修正後の行番号集合
  - f
    繰返しコード c が記述されているファイル名
  - p
    リビジョン r_{i+1} に含まれる繰返しコード

  この関数は，f と繰返しコード p のファイル名が等しく，かつ L_{after} のすべての行が繰返しコード p に含まれるならば true を返し，それ以外は false を返す．
- *getStartLine*
  引数として繰返しコードをとり，そのコード片の開始行を返す．
- *getEndLine*
  引数として繰返しコードをとり，そのコード片の終了行を返す．

Algorithm 1 繰返しコードの修正後における状態判定

```
Input: c (c ∈ C_{r_i}), C_{r_{i+1}}
Output: flg
 1: L_before ← getModifiedLines(c)
 2: L_after ← getAfterLines(L_before)
 3: f ← getFileName(c)
 4: for all p in C_{r_{i+1}} do
 5:     flg ← judgeCondition(L_after, f, p)
 6:     if flg then
 7:         break
 8:     end if
 9: end for
10: return flg
```

Algorithm 2 メソッド *judgeCondition*

```
Input: L_after, f, p
 1: if f = getFileName(p) then
 2:     startLine_p ← getStartLine(p)
 3:     endLine_p ← getEndLine(p)
 4:     for all l in L_after do
 5:         if (l < startLine_p) or (endLine_p < l) then
 6:             return false
 7:         end if
 8:     end for
 9:     return true
10: end if
11: return false
```

これらのアルゴリズムでは，Algorithm1 の出力変数 *flg* が true ならば繰返しコード c は修正後も繰返しコードであり，false ならばそうではないことを表す．ただし，Algorithm1，2 の中では説明していないが，リビジョン r_i で繰返しコードに対して行の削除が行われた場合は，まず削除された箇所の前後の行を特定する．そして，前後の行のうちいずれかがリビジョン r_{i+1} で繰返しコードに含まれるならば，その繰返しコードは修正後も繰返しコードであると見なす．

STEP2C 修正がなかった繰返しコードの位置変移の調査

リビジョン r_i において修正が行われなかった繰返しコードの，リビジョン r_{i+1} での位置は，リビジョン r_i における繰返しコードの行番号を，その繰返しコードより前の行で行われた修正の行数分だけずらすことによって得ることができる．具体的には，削除行があった場合はその行数分だけ行番号を減らし，挿入行があった場合はその行数分だけ行番号を増やす．

STEP3 進化に関するデータの整理

STEP2 で得られた，各リビジョン間における繰返しコードの進化に関する情報をまとめて，プロジェクト中の各繰返しコードについて生成から消失まで（消失していない繰返しコードは，生成から最新リビジョンまで）の進化データを導出する．

3.4 実装

リビジョン間の差分は，SVNKit [10] の Diff 機能を利用して取得する．SVNKit とは，Subversion [11] を操作するための Java ソフトウェアライブラリである．また，繰返しコードは，2 章で説明した手法を用いて取得する．

4. 実験

4.1 目的

本実験は，繰返しコードがどのように進化していくか調べることを目的とする．具体的には以下の項目について調査する．

項目 1 繰返しコードの存在期間と，存在期間中にその繰返しコードに対して行われた修正の回数に相関はあるか．

項目 2 繰返しコードを構成する文の種類は，その繰返しコードの進化に影響を与えるか．

項目 3 繰返しコードの繰返し要素数は，その繰返しコードの進化に影響を与えるか．

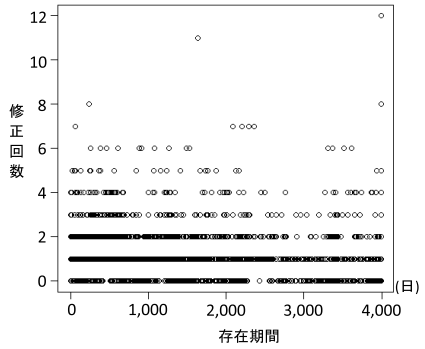
項目 4 繰返し要素のトークン数の大きさはどのくらいか．

4.2 対象

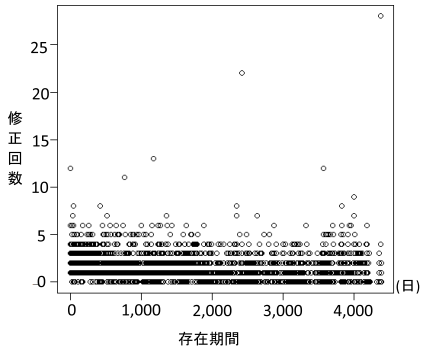
本研究で実装したツールは，Java 言語で開発されたプロジェクトのみを解析対象としている．本研究では，3 つのオープンソースプロジェクトを対象として実験を行った．対象としたプロジェクトを**表 2** に示す．

4.3 項目 1 の調査結果

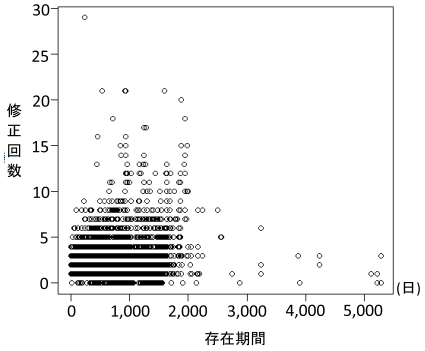
繰返しコードの存在期間と修正回数の関係を**図 4** に示す．図 4 のグラフは，各繰返しコードについて，横軸にその存在期間（日数）を，縦軸に存在期間中の修正回数をとるようプロットしている．図 4 から，存在期間が長いからといって繰返しコードの修正回数が多くなるといったことはないことが分かる．また，**表 3** に示すように，相関係数はいずれもほぼ 0 に近い値であった．したがって，繰返しコードの存在期間と，存在期間中にその繰返しコードに対して行われた修正の回数に相関はない．さらに，この実験の中では，各繰返しコードの修正回数のみに着目した調査も行った．繰返しコードを，生成されてから 1 度も修正が加えられなかったものと，1 回以上の修正が加えられたものに分類し，それぞれの割合を**表 4** にまとめた．表 4 よ



(a) jEdit



(b) Ant



(c) ArgoUML

図 4 各繰返しコードに対する存在期間と修正回数

Fig. 4 Survival period and the number of modifications on repeated code.

表 3 繰返しコードの存在期間と修正回数の相関係数

Table 3 Correlation coefficient between survival period and the number of modifications on repeated code.

プロジェクト名	相関係数
jEdit	−0.3535105
Ant	−0.1490333
ArgoUML	0.1740145

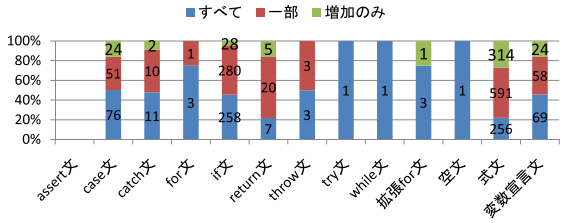
表 2 実験対象プロジェクト

Table 2 Target projects.

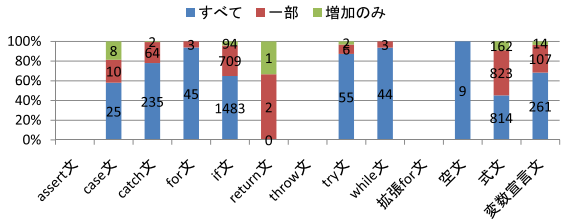
プロジェクト名	リビジョン番号		対象リビジョン数	行数		計測時間（分）
	開始	最終		開始	最終	
jEdit	3,791	21,981	5,292	57,837	183,006	1,099
Ant	267,548	1,233,420	12,621	7,864	255,061	1,798
ArgoUML	2	19,893	17,731	20,287	369,583	3,898

表 4 繰返しコードの修正回数別の割合
Table 4 Ratio of the number of modifications on repeated code.

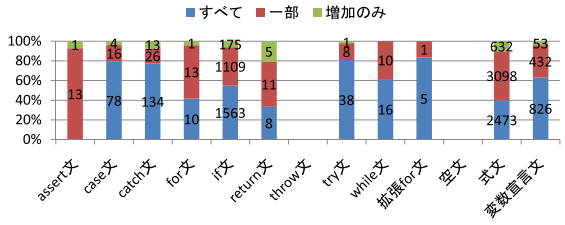
プロジェクト名	修正回数	
	0 回	1 回以上
jEdit	27%	73%
Ant	15%	85%
ArgoUML	11%	89%



(a) jEdit



(b) Ant



(c) ArgoUML

図 5 修正内容の内訳（文の種類別）
Fig. 5 Ratio of modification types by focusing on instruction types.

り，全体の 73〜89%の繰返しコードが，少なくとも 1 回以上修正されていることが分かる．

4.4 項目 2 の調査結果

修正が加えられた繰返しコードのうち，修正後も繰返しコードであるものについて，文の種類別に修正内容の内訳を調査した．修正内容は，以下の 3 種類に分類した．

すべて すべての繰返し要素に変更が加えられたもの
一部 一部の繰返し要素に変更が加えられたもの
増加のみ 既存の繰返し要素に対する変更はなかったが，その修正によって繰返し要素数が増加したもの

修正後も繰返しコードであるものについて，文の種類別に修正内容の内訳をまとめたものを**図 5** に示す．図 5 の横軸は繰返しコードを構成する文の種類を，縦軸はその文を含む繰返し要素に対する修正内容の内訳を表している．ただし，1 つの繰返し要素に複数の種類の文が含まれる場合は，それらすべての文の項目について，その繰返し要素に対する修正内容を反映している．たとえば，図 5 では，空文の項目に棒グラフがある．これは，空文と空文以外の文から構成される繰返し要素に対する修正内容が空文の項目に反映されたためであり，空文に対して修正が加えられたのではない．

また，棒グラフがない項目については，その文を含む繰返しコードに修正が行われなかったことを意味する．なお，横軸における文の種類の選定基準は以下のとおりである．

- JDT で文として定義されている．
- いずれかの実験対象プロジェクトにおいて，その文から構成される繰返しコードが存在する．

たとえば，図 5 の横軸には switch 文が出現していないが，これは，いずれの実験対象プロジェクト中の繰返しコードにも，switch 文が含まれていなかったことを表す．図 5 から，式文は “一部” への修正が 5 割以上と，高い割合を占めることが分かる．一方，case 文，catch 文，try 文，while 文，変数宣言文は “すべて” への修正が 5 割以上と，高い割合を占めることが分かる．

4.5 項目 3 の調査結果

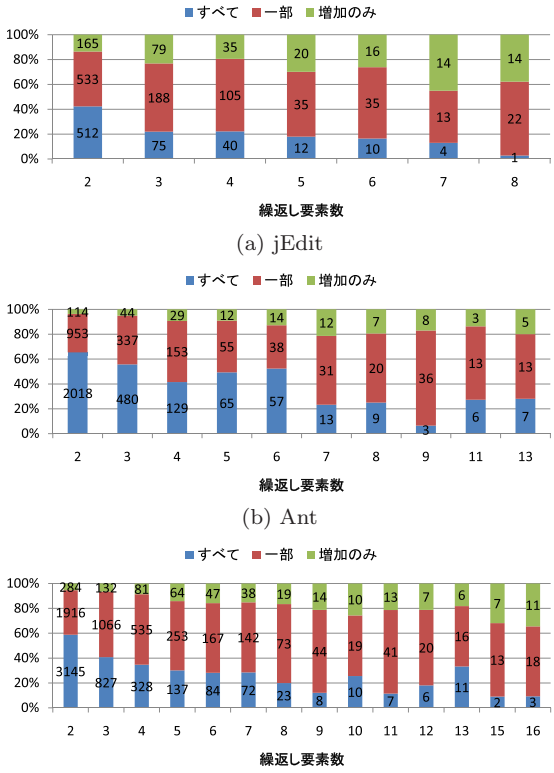
修正が加えられた繰返しコードのうち，修正後も繰返しコードであるものについて，繰返し要素数別に修正内容の内訳を調査した．修正内容は，項目 2 に対する調査のときと同様に，すべて，一部，増加のみの 3 種類に分類した．ただし，繰返し要素に変更が加えられ，かつ，その修正によって要素数が変化したものについては，すべてもしくは一部に含まれる．

修正後も繰返しコードであるものについて，繰返し要素数別に修正内容の内訳をまとめたものを**図 6** に示す．図 6 の横軸は繰返しコードの繰返し要素数を，縦軸はその繰返し要素数からなる繰返しコードに対する修正内容の内訳を表している．なお，横軸には，その繰返し要素数からなる繰返しコードに対して修正が 20 回以上行われているもののみを記載している．図 6 からは，以下のことが分かる．

- 繰返し要素数が少ない繰返しコードほど “すべて” に該当する割合が高い．
- 繰返し要素数が多い繰返しコードは，繰返し要素数が少ないものに比べて “増加のみ” に該当する割合が高い傾向がある．

このような結果が得られた要因について追究するために，実験対象プロジェクト中の繰返しコードの一部を目視により確認した．それにより，以下のような傾向があることが分かった．

- 繰返し要素数が少ない繰返しコードほど，繰返し要素間の関連が強い．
- 繰返し要素数が多い繰返しコードほど，繰返し要素間の関連が弱い．



(a) jEdit

(b) Ant

(c) ArgoUML

図 6 修正内容の内訳（繰返し要素数別）

Fig. 6 Ratio of modification types by focusing on the number of repetitions.

繰返し要素数が少ない繰返しコードの例として，**図 7** が見つかった．この繰返しコードでは，各繰返し要素において，変数 *s.start* および *s.end* に対して同様の処理が行われている．*s.start* と *s.end* はそれぞれ，テキストエリア中における選択部分の開始箇所と終了箇所を表す変数であり，必ず対になって処理が行われており関連度が高かった．実際に，この繰返しコードには，すべての繰返し要素に同様の修正が加えられることがあった．また，繰返し要素数が多い繰返しコードの例として，**図 8** が見つかった．この繰返しコードの各繰返し要素は，オプション操作に関する GUI の部品をグループに追加するという共通点を持つが，追加したそれぞれの GUI には，同様の処理を担うなどの関連はなかった．実際に，この繰返しコードには，一部の繰返し要素にのみ修正が加えられることがあった．

繰返し要素数が少ない繰返しコードほど “すべて” に該当する割合が高く，繰返し要素数が多くなるにつれて “一部” に該当する繰返しコードの割合が高くなるのは，以上のような理由のためではないかと著者らは考えた．

また，“一部” に該当するものには，本来ならばすべての繰返し要素に修正を加えるべきであったが，開発者のミスなどにより特定の繰返し要素への修正が見落とされているといった，いわゆる修正漏れの事例が含まれる．実際に，修正漏れの事例として**図 9** のようなものが存在した．図中の強調表示箇所は，2 つのリビジョン間で修正が行われた

```
if(s.start >= offset)
{
 s.start += length;
 s.startLine = getLineOfOffset(s.start);
 changed = true;
}

if(s.end >= offset)
{
 s.end += length;
 s.endLine = getLineOfOffset(s.end);
 changed = true
}
```

図 7 繰返し要素数が少ない繰返しコード

Fig. 7 Repeated code including not many repeating elements.

```
addOptionPane(new GeneralOptionPane(), jEditGroup);
addOptionPane(new LoadSaveOptionPane(), jEditGroup);
addOptionPane(new EditingOptionPane(), jEditGroup);
addOptionPane(new ModeOptionPane(), jEditGroup);
addOptionPane(new TextAreaOptionPane(), jEditGroup);
addOptionPane(new GutterOptionPane(), jEditGroup);
addOptionPane(new ColorOptionPane(), jEditGroup);
addOptionPane(new StyleOptionPane(), jEditGroup);
addOptionPane(new ShortcutsOptionPane(), jEditGroup);
addOptionPane(new DockingOptionPane(), jEditGroup);
addOptionPane(new ContextOptionPane(), jEditGroup);
addOptionPane(new ToolBarOptionPane(), jEditGroup);
addOptionPane(new AbbrevsOptionPane(), jEditGroup);
addOptionPane(new PrintOptionPane(), jEditGroup);
addOptionPane(new BrowserOptionPane(), jEditGroup);
```

図 8 繰返し要素数が多い繰返しコード

Fig. 8 Repeated code including many repeating elements.

```
45:       Model.getCoreFactory().buildClass("A", ns1);
46:       Model.getCoreFactory().buildClass("A", ns1);
47:       Model.getCoreFactory().buildClass("B", ns1);
48:       Model.getCoreFactory().buildClass("A", ns2);
```

(a) リビジョン 11,063

```
45:  c1 = Model.getCoreFactory().buildClass("A", ns1);
46:  c2 = Model.getCoreFactory().buildClass("A", ns1);
47:  c3 = Model.getCoreFactory().buildClass("B", ns1);
48:       Model.getCoreFactory().buildClass("A", ns2);
```

(b) リビジョン 11,064

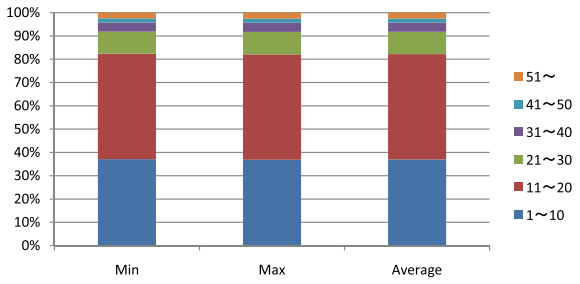
```
45:  c1 = Model.getCoreFactory().buildClass("A", ns1);
46:  c2 = Model.getCoreFactory().buildClass("A", ns1);
47:  c3 = Model.getCoreFactory().buildClass("B", ns1);
48:  c4 = Model.getCoreFactory().buildClass("A", ns2);
```

(c) リビジョン 11,065

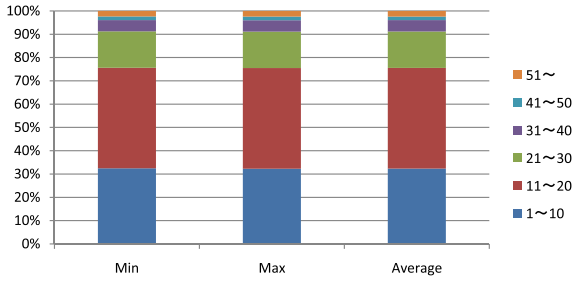
図 9 修正漏れの実例（ArgoUML-TestCrNameConflict.java）

Fig. 9 Example of modification overlooking.

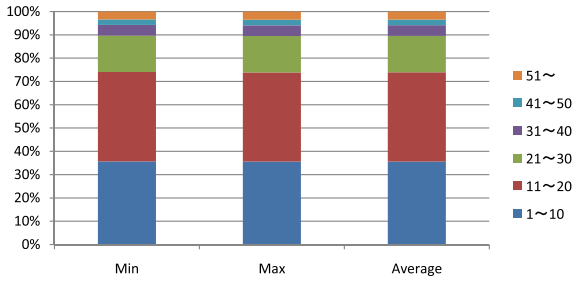
部分を表している．この例では，リビジョン 11,063 において，各メソッド呼び出し文を 1 つの繰返し要素とする，繰返し要素数 4 の繰返しコードが構成されている．そして，リビジョン 11,063，11,064 間で，この繰返しコードを構成する 4 つの繰返し要素のうち，上の 3 つに対してのみ同様の修正が加えられている．さらにその後，リビジョン 11,064，11,065 間で，残りの 1 つの繰返し要素に対しても同様の修正が加えられている．これらの修正は，本来ならばすべて同時に行われるはずであったが，何らかの理由で一部に漏れが生じたものと推測される．



(a) jEdit



(b) Ant



(c) ArgoUML

図 10 繰返し要素のトークン数
Fig. 10 Token size of repeating element.

4.6 項目 4 の調査結果

繰返しコードを構成する繰返し要素のトークン数の内訳を**図 10** に示す．繰返し要素のトークン数は，繰返しコードの進化にともない変動するため，一定の値で表すことができない．そこで，実験では，繰返しコードを構成する繰返し要素のトークン数の内訳について，以下の 3 つの項目に分けて調査した．

Min 繰返しコードの存在期間中における繰返し要素の最少トークン数

Max 繰返しコードの存在期間中における繰返し要素の最大トークン数

Average 繰返しコードの存在期間中における繰返し要素の平均トークン数

図 10 から，繰返し要素のトークン数については，Min，Max，Average にほとんど差がないことが分かる．また，繰返し要素は 9 割以上がトークン数 30 以下という比較的小さな単位で構成されているということが分かる．

5. 考察

5.1 修正支援の有用性

実験により得られた結果をもとに，修正支援の有用性について考察する．

存在期間と修正回数

繰返しコードの存在期間と修正回数の間に相関はなかった．しかし，73〜89%の繰返しコードには少なくとも 1 回は修正が行われているので，繰返しコードに対して修正支援を行うことで，修正のコストを下げることができる可能性がある．

文の種類

実験により，修正内容の傾向の違いはあるものの，様々な種類の文の繰返しコードに対して修正が加えられていることが分かった．したがって，特定のコードではなく，様々な種類の文についても，修正支援が必要であろう．

繰返し要素数

実験により，繰返し要素数が少ないほど，すべての繰返し要素に同様の修正が加えられる割合が高いことが分かった．そこで，繰返し要素数が少ないことを利用して，繰返しコード中のある繰返し要素に修正が加えられた場合に，他の各繰返し要素に対して 1 つずつ修正箇所を提示するような支援が有用であろう．

類似コードはしばしば一貫した修正を必要とする．しかし，そのような一貫した修正が必要な類似コードについて，あるコード片が修正されたとき，そのコード片と類似したコード片に対して修正漏れが起こる可能性が考えられる．実際に，ソースコード中に存在する類似コードは一貫性の保持が困難であることが既存研究によって報告されている [2]．この既存研究では，あるコード片が修正されたとき，その後しばらく経過してから，そのコード片と類似したコード片に対しても同様の修正が加えられることがあると述べられている．このように，類似した複数のコード片に同様の修正を加える際に，同じタイミングで修正を加えるべきであったコード片に対して修正漏れが生じると，類似コードの一貫性が損なわれてしまう．これと同様のことが，繰返しコードに対してもいえるのではないかと著者らは考えた．そこで，繰返しコードのある繰返し要素が修正された場合に，他の各繰返し要素に対して 1 つずつ修正箇所を提案することは，修正すべき繰返し要素への修正漏れの防止策の 1 つとして有効であろう．

実際に，このような修正支援が有効な事例として，**図 11** に示す進化が見つかった．図 11 (a) の強調表示箇所は修正によって削除されたコード片を，図 11 (b) の強調表示箇所は修正によって追加されたコード片を表している．この進

```
if ( m_Version != null) {
    cmd.createArgument().setValue(FLAG_VERSION + m_Version);
} else if ( m_Date != null) {
    cmd.createArgument().setValue(FLAG_VERSION_DATE + m_Date);
} else if (m_Label != null) {
    cmd.createArgument().setValue(FLAG_VERSION_LABEL + m_Label);
}
```

(a) 修正前

```
if ( m_Version != null) {
    cmd.createArgument().setValue(FLAG_VERSION);
    cmd.createArgument().setValue(m_Version);
} else if ( m_Date != null) {
    cmd.createArgument().setValue(FLAG_VERSION_DATE);
    cmd.createArgument().setValue(m_Date);
} else if (m_Label != null) {
    cmd.createArgument().setValue(FLAG_VERSION_LABEL);
    cmd.createArgument().setValue(m_Label);
}
```

(b) 修正後

図 11 繰返しコードの修正事例 1
Fig. 11 Example of modification on repeated code1.

化事例では，修正によって，各繰返し要素に対してメソッド呼び出し文が新たに追加され，かつ，元々存在していたメソッド呼び出し文の引数の一部が移動し，追加されたメソッド呼び出し文の引数となっている．図 11 のように，各繰返し要素に対して同様の修正を行わなければならない場合，一部の繰返し要素への修正漏れによりバグが発生する可能性がある．そこで，ある繰返し要素に修正が行われた際に，他のすべての繰返し要素に対して 1 つずつ修正の提案を行う支援を導入することで，修正漏れによるバグの混入を回避することができると著者らは考える．

繰返し要素の増加

実験により，繰返し要素数が多い繰返しコードは，修正によって繰返し要素が増加する傾向が強いことが明らかとなった．そのため，このような繰返しコードについては，自動的に新たな繰返し要素を挿入する修正支援を導入することで，開発効率をさらに向上させることができるであろう．

また，すでにソースコード中にあるコード片について，そのコード片と類似した新たなコード片を追加する場合，しばしばコピーアンドペーストが用いられる．しかし，コピーアンドペーストによって新たに生成されたコード片は，リテラルの置き換え忘れなどによってバグの原因となることがあると既存研究で報告されている [3]．これと同様のことが，繰返しコードに新たな繰返し要素を追加する場合にも起こるのではないかと考えられる．そこで，繰返しコードに対して自動的に新たな繰返し要素を挿入するだけでなく，繰返し要素間でリテラルが異なる箇所についてリテラルの入力をユーザに促す支援を導入することで，リテラルの置き換え忘れによるバグの防止が期待できる．

実際に，繰返し要素の自動挿入が可能な事例として，**図 12** に示す進化が見つかった．図 12 (b) の強調表示箇所は修正によって追加されたコード片を表している．この進

```
case 'n':
 if(escape){
   buf.append('¥n');
   escape = false;
   break;
 }
case 't':
 if(escape){
   buf.append('¥t');
   escape = false;
   break;
 }
```

(a) 修正前

```
case 'n':
 if(escape){
   buf.append('¥n');
   escape = false;
   break;
 }
case 'r':
 if(escape){
   buf.append('¥r');
   escape = false;
   break;
 }
case 't':
 if(escape){
   buf.append('¥t');
   escape = false;
   break;
 }
```

(b) 修正後

図 12 繰返しコードの修正事例 2
Fig. 12 Example of modification on repeated code2.

化事例では，修正によって，新たな繰返し要素が 1 つ追加されている．図 12 のように，新たな繰返し要素を追加するためには，ユーザは，コピーアンドペーストなどにより，既存の繰返し要素を複製しなければならない．しかし，この例では，挿入された繰返し要素は，既存の繰返し要素と一部リテラルが異なっている．このようなリテラルについて，置き換え忘れがあった場合，バグが発生する可能性がある．そこで，リテラルの入力を促す支援を導入することで，このようなリテラルの置き換え忘れによるバグの混入を回避することができると著者らは考える．

トークン数

コードクローン検出ツールで大きさの小さいコードクローンを検出する場合，偶然の一致によるコードクローンが多く検出されてしまい，実用的ではない．したがって，通常はそのような小さいコードクローンは検出対象にはならない．一方，検出対象の大きさに制限のない繰返しコード検出では，上記の理由のために通常のコードクローン検出では除外されることの多かった小さな繰返しコードが大量に検出されていた．

また，繰返し要素は大きさが小さい傾向があることから，多くの繰返しコードは，コードクローン検出手法では検出することが難しい．よって，繰返しコードの検出に特化した手法およびツールが必要である．

5.2 修正支援の優位性

一般的なコード修正支援と比べて，繰返しコードに対する修正支援にはどのような優位性があるか考察する．

繰返し要素の挿入

一般的なコード修正支援では，すでにソースコード中にあるコード片について，そのコード片と類似した新たなコード片を自動的に挿入することはできない．そのようなコード片を挿入するには，通常はコピーアンドペーストなどにより手動でコード片を複製しなければならない．また，コピーアンドペーストによって新たなコード片を追加した場合，リテラルの置き換え忘れなどによりバグが混入する可能性がある．一方で，本研究で提案した修正支援では，繰返しコードに対して自動的に新たな繰返し要素を挿入することができる．さらに，リテラルの入力をユーザに促すことで，リテラルの置き換え忘れによるバグの回避に役立つと考えられる．

大きさの小さい類似コードへの修正

コードクローン検出手法では，通常小さいコードクローンは検出することができない．そのため，一般的なコード修正支援では，このような小さいコードクローンに対して修正支援を行うことが難しい．一方で，繰返しコードにはその大きさによる制限がない．したがって，繰返しコードに対する修正支援を導入することで，コードクローン検出手法では検出することが困難な小さいコードクローンに対しても，それらが連続していさえすれば，修正支援を行うことができるようになると期待できる．

6. 結果の妥当性

計測対象

本実験で対象としたソフトウェアは 3 つであり，Java で開発されたオープンソースのソフトウェアに限定して調査を行った．このため，より多くのソフトウェアを対象として実験を行った場合や，異なる言語で記述されたソフトウェアや商用ソフトウェアを対象とした場合は，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．

コード分析

本実験では，繰返しコードの識別とその進化に関するデータの取得は自動で行うが，それらのコードに対する具体的な分析は行えていない．たとえば，繰返しコードに修正が加えられたという事実は確認できるが，それが具体的にどのような修正であったか，実際のソースコードを見て確認するに至っていない．したがって，繰返しコードに対して加えられた修正ついて，それが実際に修正支援の適用が有用かどうか，現段階では特定することができない．このため，修正支援の有用性について十分な考察が行えていない可能性がある．

繰返しコードの消失と再出現

本実験では，修正が加えられた繰返しコードについて，以下に示す条件のいずれかを満たした場合に，その繰返しコードが “消失” したと見なしている．

- 繰返しコードを構成していたコード片は残存しているが，修正によって繰返しコードではなくなった．
- 繰返しコード自体が削除された．

しかし，前者の場合は，この後の修正によって再び繰返しコードとなる可能性があるので，本来ならば “消失” とは切り離して考えるのが適切である．したがって，“繰返しコードの消失” についての定義が適切ではない本実験では，1 度消失した後，再び出現した繰返しコードについて，同一の繰返しコードとして追跡することができていない．そのため，1 度消失した後で再び出現した繰返しコードについて，正確な存在期間を調査できておらず，項目 1 の調査において妥当な結果が得られていない可能性がある．

7. あとがき

本研究では，繰返しコードの進化について調査した．調査の結果，繰返し要素数が少ない繰返しコードほどすべての繰返し要素に対して同様の修正が行われやすいことや，繰返し要素数が多い繰返しコードは，修正によって繰返し要素が増加する傾向が強いことなどが明らかとなった．

さらに，この結果をもとに，繰返しコードに対する有用な支援としてどのようなものがあげられるか考察を行った．繰返し要素数の少ない繰返しコードには各繰返し要素に対して同様の修正が行われやすい．したがって，1 つの繰返し要素に修正が加えられた場合に，ユーザに対し，その繰返しコードに含まれる他の各繰返し要素について 1 つずつ修正の提案を行う支援が有用であると考えた．また，繰返し要素数が多い繰返しコードは繰返し要素の追加が行われやすい．したがって，新たに繰返し要素を挿入する支援が有用であると考えた．

今後の課題は以下のとおりである．

- より多くのソフトウェアに対して実験を行う．
- 繰返しコードに加えられた修正に対して，それがどのような内容なのか，実際にコードを見て具体的に調査する．
- 1 度消失した後，再び出現した繰返しコードについて，同一の繰返しコードとして追跡する．

参考文献

[1] Kim, M., Sazawal, V., Notkin, D. and Murphy, G.: An Empirical Study of Code Clone Genealogies, *Proc. 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*,

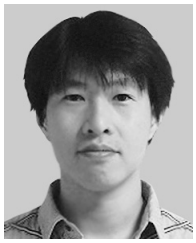
pp.187–196 (2005).
[2] 肥後芳樹，楠本真二：コード修正履歴情報を用いた修正漏れの自動検出，情報処理学会論文誌，Vol.54, No.5, pp.1686–1696 (2013).
[3] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-paste and Related Bugs in Large-scale Software Code, *IEEE Trans. Softw. Eng.*, Vol.32, No.3, pp.176–192 (2006).
[4] Toomim, M., Begel, A. and Graham, S.: Managing Duplicated Code with Linked Editing, *Proc. 2004 IEEE International Conference on Visual Languages and Human Centric Computing*, pp.173–180 (2004).
[5] Higo, Y., Ueda, Y., Kusumoto, S. and Inoue, K.: Simultaneous Modification Support Based on Code Clone Analysis, *Proc. 14th Asia-Pacific Software Engineering Conference*, pp.262–269 (2007).
[6] Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: Method and Implementation for Investigating Code Clones in a Software System, *Information and Software Technology*, Vol.49, pp.985–998 (2007).
[7] Murakami, H., Hotta, K., Higo, Y., Igaki, H. and Kusumoto, S.: Folding Repeated Instructions for Improving Token-Based Code Clone Detection, *IEEE International Workshop on Source Code Analysis and Manipulation*, pp.64–73 (2012).
[8] Sasaki, Y., Ishihara, T., Hotta, K., Hata, H., Higo, Y., Igaki, H. and Kusumoto, S.: Preprocessing of Metrics Measurement Based on Simplifying Program Structures, *International Workshop on Software Analysis, Testing and Applications*, pp.120–127 (2012).
[9] Wang, X., Pollock, L. and Vijay-Shanker, K.: Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability, *Proc. 18th Working Conference on Reverse Engineering*, pp.35–44 (2011).
[10] SVNKit, available from ⟨http://svnkit.com/⟩.
[11] Subversion, available from ⟨http://subversion.apache.org/⟩.

今里 文香

平成 25 年大阪大学基礎工学部情報科学科卒業．現在，同大学大学院情報科学研究科博士前期課程在学中．コードクローン分析に関する研究に従事．

佐々木 唯

平成 23 年大阪大学基礎工学部情報科学科卒業．平成 25 年同大学大学院博士前期課程修了．在学時，コードクローン分析に関する研究に従事．



肥後 芳樹 （正会員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．平成 19 年同大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

楠本 真二 （正会員）

昭和 63 年大阪大学基礎工学部卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．