社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソースコード中に含まれる繰り返しコードの進化に関する調査

今里　文香† 　佐々木　唯† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科

あらまし　コードクローンの進化に関する既存研究によって，ある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り返しコードが存在する．著者らは，繰り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り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，繰り返し回数が少ないほど，すべての繰り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 Research on Evolution of Repeated Code in Program Source Code

Ayaka IMAZATO†, Yui SASAKI†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University

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. Therefore 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. 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.

Key words　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伴い，形を変えながら様々な進化を遂げていく [1]．ある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．この為，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．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，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得る [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繰り返しコードが多く存在する [6]〜[8]．代表的な例として，if-else 文や，case 文などが繰り返しコードとなりやすい．このような繰り返しコードについても，コードクローンのように，一つの繰り返しコードを構成する各繰り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，繰り返しコードの進化について調査を行った．そして調査結果を元に，繰り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挙げられるか考察を行った．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り返しコードに対して，その生成から消失までの変遷を調査し，加えられた修正の内容などの情報から，繰り返しコードの進化傾向を導き出した．

2. 繰り返しコード

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，繰り返しコードと呼ぶ．なお，繰り返しコードの識別は Sasaki らが提案している手法 [8] を元に行う．Sasaki らの手法では，繰り返しコードの識別は，ソースコードから構築される AST の構造を元に行われる．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．

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

(a) 繰り返し要素を構成する文が一つ

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

(b) 繰り返し要素を構成する文が複数

図 1　繰り返しコード例

そこで，AST 上の連続する兄弟ノードについて，それぞれのノードが示す文が類似していれば，そのノード以下の部分木をを構築するコード片を繰り返しコードとみなす．文の類似性の判定基準ついては次節で述べる．

例えば図 1(a) のソースコードは，三つの変数宣言文からなる繰り返しコードである．このとき，繰り返しの単位となっているコード片を**繰り返し要素**と呼ぶこととする．この例の場合，繰り返し要素は各変数宣言文であり，繰り返し要素数は 3 である．また，図 1(b) のソースコードは，繰り返し要素が代入文とメソッド呼び出し文，繰り返し要素数が 2 の繰り返しコードである．

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みなす．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，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．

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みなす．

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みなす．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，類似する文とみなす．

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，類似する文とみなす．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．

表 1　文の種類と Java における記述形式

文の種類	記述形式
メソッド呼び出し文	object.method(...); method(...);
変数宣言文	type name; type name = *Expression*;
代入文	name = *Expression*;
return 文	**return** *Expression*;
throw 文	**throw** *Expression*;

return 文・throw 文

オペランドに現れる式の種類が同じであれば，類似する文とみなす．

3. 調 査 手 法

3.1　概　　要

ソースコード中に含まれる繰り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，以下の通りである．

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り返しコードの進化に関するデータ

3.2　調 査 手 順

調査手順は，以下の三つの STEP から成る．

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，繰り返しコードの進化を調査

STEP3　各繰り返しコードの生成から消失までの進化に関するデータを整理

この手順を図 2 に示す．

各ステップの詳細は以下の通りである．

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成るリビジョン列 $R = \{r_1, r_2, \cdots, r_n\}$ 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．例えば，r_i は，R 中の i 番目の要素を表す．

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り返しコードの進化の調査

STEP2 は以下の三つの STEP から成る．

STEP2A　繰り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} 間の差分を取り，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 に含まれる繰り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，その繰り返しコードには修正があったとみなす．

ただし，リビジョン r_i にて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

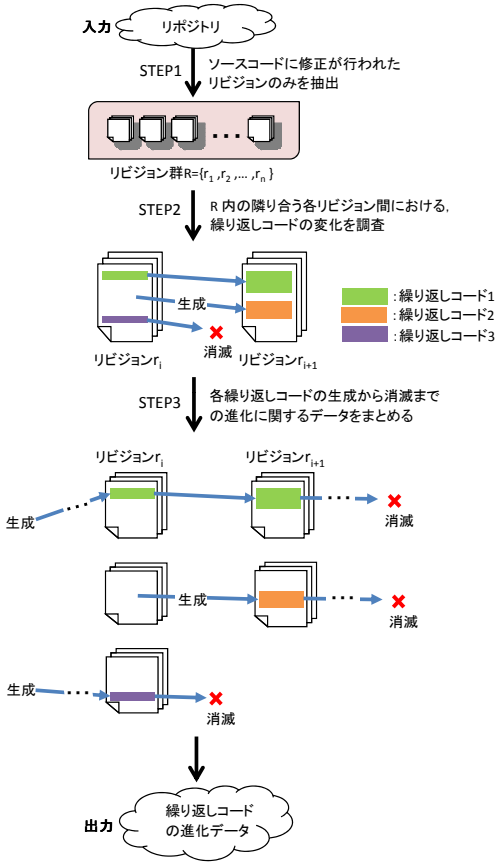


図 2　手順の流れ

うちいずれかがリビジョン r_i の繰り返しコードに含まれるならば，その繰り返しコードに修正があったとみなす．

STEP2B　繰り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 で繰り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} の繰り返しコードに含まれるならば，その繰り返しコードは修正後も繰り返しコードであるとみなす．この判定アルゴリズムを，Algorithm1 および 2 に示す．

ただし，リビジョン r_i にて繰り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} で繰り返しコードに含まれるならば，その繰り返しコードは修正後も繰り返しコードであるとみなす．

Algorithm1 および 2 は，リビジョン r_i に含まれる繰り返しコード c が修正後のリビジョン r_{i+1} でも繰り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．入力変数 c はリビジョン r_i に含まれる一つの繰り返しコードを指す．また，関数 $getRevisedLines$ は，引数として繰り返しコードを取り，リビジョン r_i におけるその繰り返しコード中の修正位置を返す．関数 $getAfterLines$ は，引数として繰り返しコードの位置を取り，その繰り返しコードのリビジョン r_{i+1} での位置を返す．関数 $getFileName$ は，引数として繰り返しコードを取り，その繰り返しコードが記述されているファイルの名前を返す．また，関数 $getStartLine$ および $getEndLine$ はそれぞれ，引数として繰り返しコードを取り，そのコード片の開始行，終了行を返す．これらのアルゴリズムでは，Algorithm1 の出力変数 flg が true ならば繰り返しコード c は修正後も繰り返しコードであり，false ならばそうではないことを表す．

Algorithm 1 繰り返しコードの修正後における状態判定

Input: c $(c \in C_{r_i})$, $C_{r_{i+1}}$
Output: flg

```
1: L_before ← getRevisedLines(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
```

STEP2C　修正がなかった繰り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り返しコードの，リビジョン r_{i+1} での位置は，リビジョン r_i おける繰り返しコードの行番号を，その繰り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り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り返しコードについて生成から消失までの進化データを導出する．

3.3 実　　装

SVNKit [10] とは，Subversion を操作するための Java ソフトウェアライブラリである．リビジョン間の差分は，このSVNKit の Diff 機能を利用して取得する．また，繰り返しコードは，2. 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，繰り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．

RQ1　繰り返しコードの存在期間と，存在期間中にその繰り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

RQ2　繰り返しコードの繰り返し要素数は，その繰り返しコードの進化に影響を与えるか

RQ3　繰り返しコードを構成する文の種類は，その繰り返しコードの進化に影響を与えるか

RQ4　繰り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，三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 RQ1 の結果

繰り返しコードの存在期間と修正回数の関係を図 3 に示す．図 3 のグラフは，各繰り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 3 より，相関係数はいずれも 0 に近い値であるため，繰り返しコードの存在期間と修正回数には相関がない．

4.4 RQ2 の結果

修正が加えられた繰り返しコードのうち，修正後も繰り返しコードであるものについて，繰り返し要素数別に修正内容の内訳を調査した．修正内容は，以下の 3 種類に分類される．

すべて　すべての繰り返し要素に変更が加えられたもの

一部　一部の繰り返し要素に変更加えられたもの

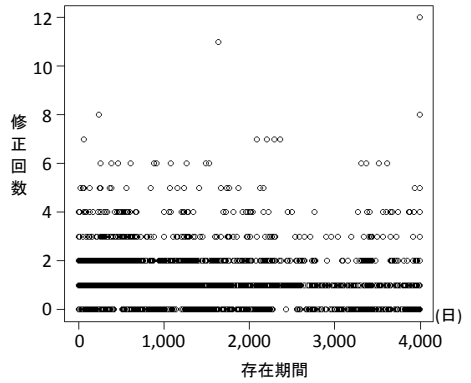
増加のみ　既存の繰り返し要素に対する変更はなかったが，その修正によって繰り返し要素数が増加したもの

ただし，繰り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．修正後も繰り返しコードであるものについて，繰り返し要素数別に修正内容の内訳をまとめたものを図 4 に示す．図 4 の横軸は繰り返しコードの繰り返し要素数を，縦軸はその繰り返し要素数からなる繰り返しコードに対する修正内容の内訳を表している．図 4 からは，以下のことがわかる．

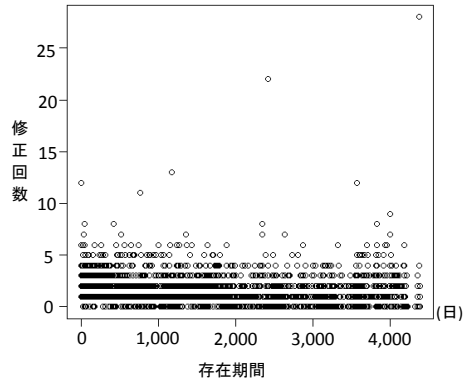
- 繰り返し要素数が少ない繰り返しコードほど ”すべて ” に該当する割合が高い
- 繰り返し要素数が多い繰り返しコードほど修正によって繰り返し要素が増加する傾向が強い

4.5 RQ3 の結果

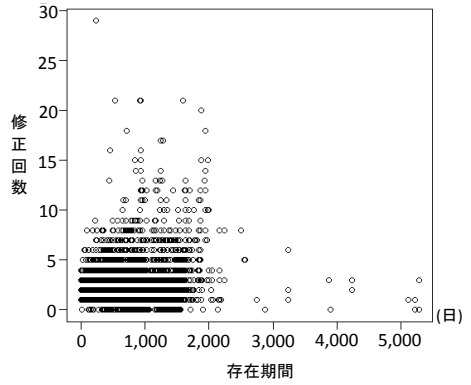
修正が加えられた繰り返しコードのうち，修正後も繰り返しコードであるものについて，文の種類別に修正内容の内訳を調査した．修正内容は，RQ2 に対する調査のときと同様に，すべて，一部，増加のみの 3 種類に分類される．



(a) jEdit($\rho = -0.3535105$)



(b) Ant($\rho = -0.1490333$)



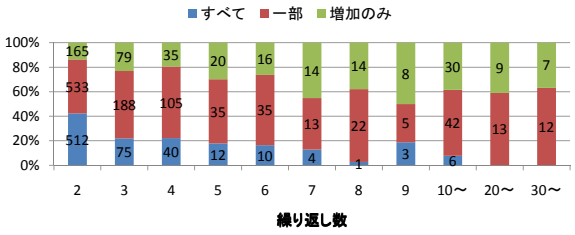
(c) ArgoUML($\rho = 0.1740145$)

図 3　各繰り返しコードに対する存在期間と修正回数

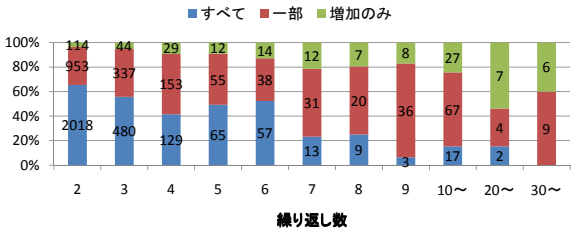
修正後も繰り返しコードであるものについて，文の種類別に修正内容の内訳をまとめたものを図 5 に示す．図 5 の横軸は

表 2　実験対象プロジェクト

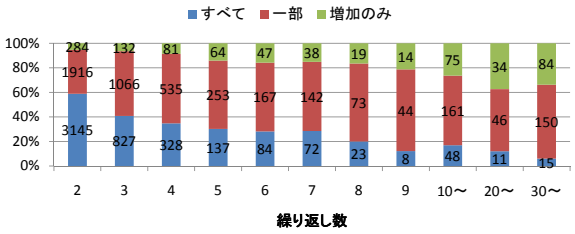
プロジェクト名	リビジョン番号		対象リビジョン数	行数		計測時間 (分)
	開始	最終		開始	最終	
jEdit	3,791	21,981	5,292	57,837	183,006	1,099
Ant	267,548	1,233,420	12,621	7,864	255,061	1,798
ArgoUML	2	19,893	17,731	20,287	369,583	3,898



(a) jEdit

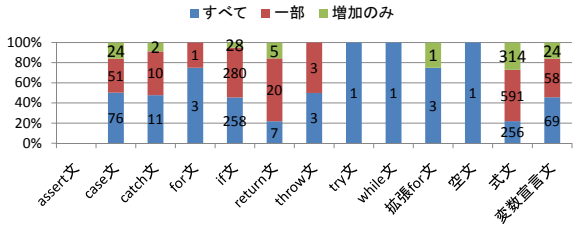


(b) Ant

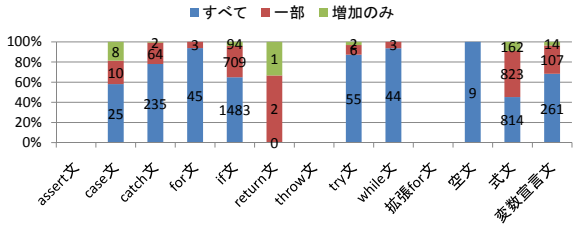


(c) ArgoUML

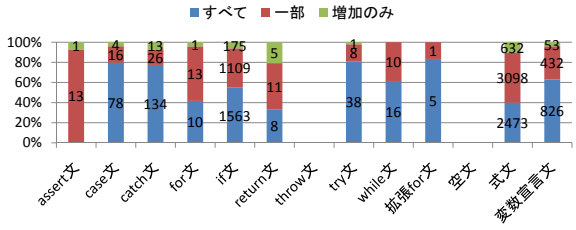
図 4　修正内容の内訳 (繰り返し要素数別)



(a) jEdit



(b) Ant



(c) ArgoUML

図 5　修正内容の内訳 (文の種類別)

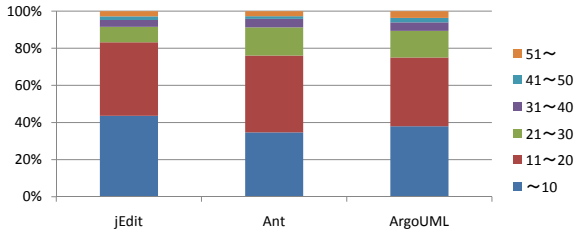


図 6　繰り返し要素のトークン数

繰り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り返しコードに対する修正内容の内訳を表している．ただし，一つの繰り返し要素に複数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，その繰り返しコードに対する修正内容を反映している．また，棒グラフがない項目については，その文を含む繰り返しコードに修正が行われなかったことを意味する．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わかる．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わかる．

4.6　RQ4 の結果

各プロジェクトに含まれる繰り返し要素のトークン数の内訳を図 6 に示す．横軸がプロジェクト名，縦軸がそのプロジェクトに含まれる繰り返し要素のトークン数の内訳を表している．図 6 からは，繰り返し要素の 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わかる．

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

繰り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，多くの繰り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，繰り返しコードに対する修正支援は必要である．

繰り返し要素数

実験により，繰り返し要素数が少ないほど，すべての繰り返し要素に同様の修正が加えられる割合が高いことがわかった．そこで，繰り返し要素数が少ないことを利用して，繰り返しコード中のある繰り返し要素に修正が加えられた場合に，他の各繰り返し要素に対して一つずつ修正箇所を提示するような支援が有用であると考えられる．

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，様々な文の種類の繰り返しコードに対して修正が加えられていることがわかった．したがって，どの文の種類についても，修正支援が必要である．

繰り返し要素の増加

実験により，繰り返し要素数が多いほど，修正によって繰り返しコードの繰り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，このような繰り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，コーディングの効率をさらに向上させることができる．

トークン数

既存のコードクローン検出ツールでトークン数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として除外される．一方，検出対象のトークン数の大きさに制限のない繰り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り返しコードを大量に検出されていた．

また，繰り返し要素はトークン数が小さい傾向があることから，多くの繰り返しコードは，既存の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，繰り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．

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三つであり，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，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，繰り返しコードに修正が加えられた場合に，それが具体的にどのような内容であったか実際のソースコードを見て分析を行うに至っていない．したがって，繰り返しコードに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，現段階では断定することができていない．

繰り返しコードの消失と再出現

本実験では，一度消失した繰り返しコードが，その後の修正によって再び繰り返しコードとなった場合に，同一の繰り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，一度消失した後で再び出現した繰り返しコードについて，正確な存在期間を調査できていない．したがって，RQ1 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，繰り返しコードの進化について調査した．調査の結果，繰り返し要素数が少ない繰り返しコードほどすべての繰り返し要素に対して同様の修正が行われやすいことや，繰り返し要素数が多い繰り返しコードほど修正によって繰り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，この結果を元に，繰り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挙げられるか考察を行った．繰り返し要素数の少ない繰り返しコードには各繰り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，一つの繰り返し要素に修正が加えられた場合に，開発者に対し，その繰り返しコードに含まれる他の各繰り返し要素について一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，繰り返し要素数が多い繰り返しコードは繰り返し要素の追加が行われやすい．したがって，新たに繰り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通りである．

- より多くのソフトウェアに対して実験を行う．
- 繰り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．
- 一度消失した後，再び出現した繰り返しコードについて，同一の繰り返しコードとして追跡する．

謝辞 本研究は，日本学術振興会科学研究費補助金萌芽研究 (課題番号 : 24650011)，若手研究 (A)(課題番号 : 24680002) の助成を得た．

文　　献

[1] K. Miryung, S. Vibha, N. David, and M. Gail, “An empirical study of code clone genealogies,” Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp.187–196, Sept. 2005.

[2] Y. Higo and S. Kusumoto, “How Often Do Unintended Inconsistencies Happen? –Deriving Modification Patterns and Detecting Overlooked Code Fragments–,” 28th IEEE International Conference on Software Maintenance, pp.222–231, Sept. 2012.

[3] Li Z., Lu S., Myagmar S., and Zhou Y., “CP-Miner: finding copy-paste and related bugs in large-scale software code,” IEEE Transactions on Software Engineering, vol.32, no.3, pp.176–192, March 2006.

[4] M. Toomim, A. Begel, and S. Graham, “Managing Duplicated Code with Linked Editing,” Proceedings of the 2004 IEEE International Conference on Visual Languages and Human Centric Computing, pp.173–180, Sept. 2004.

[5] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, “Simultaneous Modification Support based on Code Clone Analysis,” Proceedings of the 14th Asia-Pacific Software Engineering Conference, pp.262–269, Dec. 2007.

[6] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “Method and implementation for investigating code clones in a software system,” Information and Software Technology, vol.49, pp.985–998, Sept. 2007.

[7] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, “Folding Repeated Instructions for Improving Token-Based Code Clone Detection,” IEEE International Workshop on Source Code Analysis and Manipulation, pp.64–73, Sept. 2012.

[8] Y. Sasaki, T. Ishihara, K. Hotta, H. Hata, Y. Higo, H. Igaki, and S. Kusumoto, “Preprocessing of Metrics Measurement Based on Simplifying Program Structures,” International Workshop on Software Analysis, Testing and Applications, pp.120–127, Dec. 2012.

[9] X. Wang, L. Pollock, and K. Vijay-Shanker, “Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability,” Proceedings of the 18th Working Conference on Reverse Engineering, pp.35–44, Oct. 2011.

[10] “SVNKit”. http://svnkit.com/