特別研究報告

題目

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

指導教員
楠本 真二 教授

報告者
今里 文香

平成 25 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 25 年度 特別研究報告

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

内容梗概

コードクローン (ソースコード中の同一，あるいは類似するコード片) は，あるコード片に修正すべき箇所が見つかった場合，そのコード片と一致または類似するすべてのコード片について同様の修正を検討しなければならない．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を増大させるおそれがあると考えられている．また，コードクローンを構成する複数のコード片に修正を加える際，一部のコード片に修正をし忘れるといった“修正漏れ”が起こることもあるため，コードクローンは，ソフトウェアの保守性を低下させる恐れもある．これと同様のことが，繰り返しコードに対してもいえるのではないかと考えた．繰り返しコードとは，類似した記述が連続して出現するコード片のことである．繰り返しコードについても同じような傾向が見られるのであれば，修正に要する作業量の削減および，ソフトウェアの保守性の維持を目的とする，繰り返しコードに特化した修正支援が必要であると考えられる．

本研究では，3 つの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ついて，生成から消滅までの進化を追跡し，その進化傾向から，繰り返しコードに対してどのような修正支援が有用であるか考察を行った．

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修正が加わりやすいことなどが明らかになった．また，この結果から，繰り返しコード中の 1 つの繰り返し要素に修正が加えられた場合に，その繰り返しコードに含まれる他のすべての繰り返し要素について 1 つずつ修正の提案を行うといった支援が有用なのではないかと考えた．

主な用語

コードクローン
バージョン管理システム
ソフトウェア保守

目 次

1　まえがき

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェアの保守に要するコストが増加している．ソフトウェアの保守を困難にさせる要因の 1 つとしてコードクローンへの関心が高まっており，これまでにコードクローンに関する様々な研究が盛んに行われている [1]．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り，主にコピーアンドペーストの操作等によって発生するといわれている [2]．

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 [3]．その進化の中では，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，同じクローンセットに属する他のコード片に対しても同様の修正が加えられるといったことがしばしば起こる．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と考えられている．さらに，コードクローン中の複数のコード片に対して同様の修正を加える際，本来は修正を加えなければならないコード片に対して修正をし忘れるといった，いわゆる“修正漏れ”が起こることも少なくない．“修正漏れ”が起こった場合，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もなり得る [4][5]．このような問題を解決することを目的として，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 [6][7]．

一方，ソースコード中には，繰り返しコードが存在することが，いくつかの研究によって報告されている [8][9][10]．繰り返しコードとは，類似した記述が連続して出現するコード片のことである．代表的な例として，if-else 文や，case 文などの繰り返し記述が挙げられる．また，この他にも，様々な文による繰り返しコードが，ソースコード中には存在する．

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，複数の構成要素に対して同様の修正が加えられるといったことが起こるのではないかと考えた．この場合，コードクローンと同じく，修正に要する作業量の増大および，“修正漏れ”による新たなバグの混入が予想される．そして，それに伴い，繰り返しコード中の複数の繰り返し要素に自動で同時に修正を行うなどといった，修正支援手法の提案が期待できる．

そこで本研究では，繰り返しコードの進化について調査し，調査結果から，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．繰り返しコードの進化について調査するために，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，その生成から消滅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．

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加えられる傾向が強いこと，また，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．このこと

から，繰り返し要素数の少ない繰り返しコードについては，1 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すべての繰り返し要素について 1 つずつ修正の提案を行うといった支援が有用であると考えられる．また，繰り返し要素数の多い繰り返しコードについては，新たに繰り返し要素を挿入するといった支援が有用であると考えられる．

以降，2 節でコードクローン，コードクローン検出ツール，繰り返しコード，バージョン管理システム Subversion[11]，及び，ファイルの差分情報について述べる．3 節で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．4 節でオープンソースソフトウェアを対象とした，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実験について説明し，5 節で繰り返しコードの進化と修正支援についての考察を行う．6 節で実験の妥当性について述べ，7 節で関連研究について述べる．最後に 8 節で本研究のまとめと今後の課題について述べる．

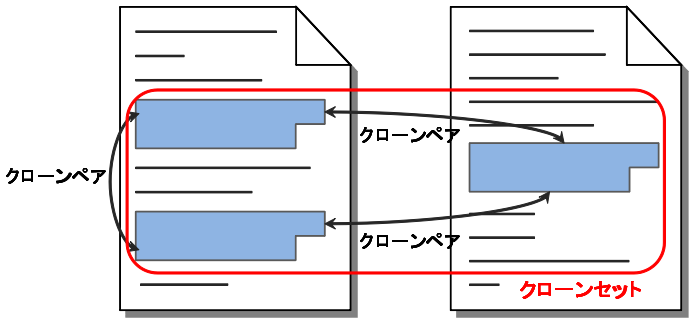


図 1: クローンペアとクローンセット

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び，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 [12]．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 [13][14]．

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．

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は，既存コードのコピーアンドペーストによる再利用，コード生成ツールの生成コード，偶然による生成などが挙げられる [15][16][17].

2.2 コードクローン検出ツール

2.2.1 コードクローン検出ツールの分類

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コードクローンをどの単位で検出するかによって，大まかに以下の 5 つに分類することができる [1][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．

行単位での検出を行うコードクローン検出ツールには，Duploc などがある [19].

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．

字句単位での検出を行うコードクローン検出ツールには，CCFinder などがある [15].

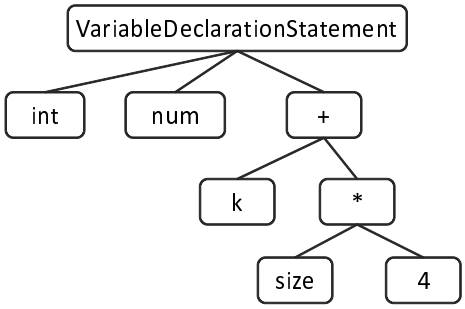


図 2: 抽象構文木の例

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．

抽象構文木を用いた検出を行うコードクローン検出ツールには，Deckard などがある [20].

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)[21] 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．

プログラム依存グラフを用いた検出を行うコードクローン検出ツールには，Duplix などがある [22].

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

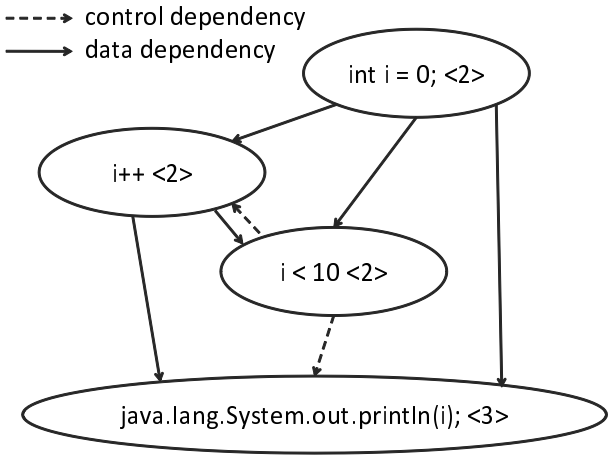


図 3: プログラム依存グラフの例

```
  fp = lookaheadset + tokensetsize;
  for (l = lookaheas(state) ; l < k ; i++) {
%     fp1 = LA + i * tokensetsize;
%     fp2 = lookaheadset;
%     while (fp2 < fp3)
%           *fp2++ |= fp1++;
  }
```

(a) コード片 1

```
  fp3 = base + tokensetsize;
  ...
  if (rp) {
      while ((j = *rp++) >= 0) {
          ...
#             fp1 = lookaheadset;
#             fp2 = LA + j * tokensetsize;
#             while (fp1 < fp3)
#                   *fp1++ |= *fp2++;
      }
  }
```

(b) コード片 2

図 4: 順序入れ替わりコードクローン

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．

メトリクスを用いた検出を行うコードクローン検出ツールには，CLAN などがある [23].

2.3 バージョン管理システム

バージョン管理システムとは，主にプログラム開発においてソースコードやその他のデータを管理するために用いるシステムである．バージョン管理システムは主に開発情報 (ソースコード，リソースなど) を開発者間で共有する機能，及び開発履歴情報を保持する機能を提供する．バージョン管理システムを用いることで離れた場所にいる開発者間で開発情報の共有が容易になるという利点があるため，商用ソフトウェア開発やオープンソースソフトウェアの開発など，多数の開発者によってソフトウェア開発が行われる際に一般的に使用されている．本研究で用いたバージョン管理システム Subversion[11] における上述の機能の概要は以下の通りである．

開発情報の共有

Subversion では，ソフトウェアの開発情報 (ソースコード，リソースなど) はリポジトリと呼ばれるデータ格納庫に保存される．開発者はソフトウェアの開発を行う際，リポジトリから開発情報を手元にコピーし (この動作をチェックアウトと呼ぶ)，修正，変更を加えた後，リポジトリに反映させる (この動作をコミット，あるいはチェックインと呼ぶ)．コミットしたとき，別の開発者が同じファイルに変更を加えていた場合は，ファイルが別の開発者によって変更されていることが通知されるので，開発者は 2 つのファイルをマージし，コミットする．これにより別の開発者が行った変更を誤って上書きすることを防ぐことができる．

開発履歴情報の保持

Subversion では，開発履歴情報をリビジョンと呼ばれる単位で保持している．リビジョンとは開発の状態を表す単位で，リポジトリがコミットを受け付けるたびに生成され，それぞれのリビジョンにはユニークな自然数 (リビジョン番号) が割り当てられる．開発履歴情報にはソースコードなどの開発情報の他，変更を加えた開発者名や日時，変更が加えられたファイル名，変更を加えた開発者の変更に関するメッセージ等のログも含まれている．開発履歴

情報を利用すれば，ソースコードに誤った変更を加えてしまった場合などに過去の状態に復元することが可能となる．

2.4　ファイルの差分情報

ファイルの差分情報とは，2 つのファイル間におけるテキストの差異に関する情報のことである．プロジェクトの保守作業のために，バージョン間の差分情報が利用されることもある．

ファイルの差分情報を出力する機能として，代表的なものに Diff が挙げられる．Diff の概要および出力されるデータの見方は，以下の通りである．

Diff の概要

Diff とは，2 つのファイルの差分情報を出力する機能である．

Diff の出力形式には，

- コンテキスト形式
- ユニファイド形式

の 2 つの種類が存在するが，以下では，本研究で用いたユニファイド形式による出力について，その見方を説明する．

出力データの見方

Diff による出力例を図 5 に示し，それを元に Diff の見方を説明する．

“Index:”で始まる行は，対象となるファイル名を表す．また，図中に橙色でマークされた“−−−”で始まる行は修正前のファイル名とリビジョンを，緑色でマークされた“+++”で始まる行は修正後のファイル名とリビジョンを表す．したがって，この例では，Javadoc.java というファイルのリビジョン 267,781 と 267,782 間の差分比較を行っているということになる．

続く“@@”で囲まれた行は，コードに修正のあった範囲を，その範囲の開始行の行番号と行数で表している．“−”の後に続く，カンマで区切られた 2 つの数は，修正前の開始行の行番号と行数を表し，“+”に後続する 2 つの数は，修正後の開始行の行番号とその行数を表す．この例では，“@@ $-158,8+169,8$@@”と書かれているので，修正前のリビジョンで 158 行目から 8 行分であった箇所が，修正後に 169 行目から 8 行分になったということになる．

```
Index: Javadoc.java
===================================
--- Javadoc.java (revision 267781)
+++ Javadoc.java (revision 267782)
@@ -158,8 +169,8 @@
    private boolean version = true;
    private DocletInfo doclet = null;
    private boolean old = false;
-   private String classpath = null;
-   private String bootclasspath = null;
+   private Path classpath = null;
+   private Path bootclasspath = null;
    private String extdirs = null;
    private boolean verbose = false;
    private String locale = null;
```

図 5: Diff の出力例 (行の修正)

```
Index: Roster.java
===========================================
--- Roster.java (revision 69)
+++ Roster.java (revision 70)
@@ -10,6 +11,7 @@
  import org.apache.log4j.Logger;
  import org.openymsg.network.ContactListType;
  import org.openymsg.network.FireEvent;
+ import org.openymsg.network.FriendManager;
  import org.openymsg.network.ServiceType;
  import org.openymsg.network.YahooUser;
  import org.openymsg.network.event.SessionEvent;
```

図 6: Diff の出力例 (行の挿入)

最後のコード部分は，修正のあった範囲に含まれる実際のコードと修正内容を表している．コードの各行に対して，削除された行には“－”，挿入された行には“＋”，修正されていない行にはスペース“ ”が行頭に付加される．また，削除行の直後に挿入行が置かれている場合は，削除行として示されている行が修正され，その修正によって，挿入行として示されている行になったということを表す．この例では，図中に青色でマークされたコード片が，修正によって，赤色でマークされたコード片になったということになる．

また，図 5 で紹介した例のほかに，行が挿入された場合の例を図 6，行が削除された場合の例を図 7 に示す．図 6 では，赤色でマークされた行が挿入され，図 7 では，青色でマークされた行が削除されている．

```
Index: Javac.java
===================================================
--- Javac.java (revision 267595)
+++ Javac.java (revision 267596)
@@ -98,9 +95,6 @@
  private String target;
  private String bootclasspath;
  private String extdirs;
- private String[] includes;
- private String[] excludes;
- private boolean useDefaultExcludes = true;

  private Vector compileList = new Vector();
  private Hashtable filecopyList = new Hashtable();
```

図 7: Diff の出力例 (行の削除)

3 繰り返しコードの識別

ソースコード中には類似した記述の繰り返しが多数存在する [8]．本研究では，このような記述の繰り返しを**繰り返しコード**と呼ぶ．Sasaki らは，メトリクス計測の前処理として，ソースコード中の繰り返しコードを識別し，折りたたむ手法を提案している [10]．本研究ではこの手法を元に繰り返しコードを識別する．

3.1 定義

本研究では，類似した文が繰り返されている場合に，繰り返しコードとみなす．文が類似しているかどうかの判定方法については後述する．例えば以下のソースコードは，3 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，繰り返されている要素を**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 1 つの変数宣言文であり，繰り返しコードを構成している繰り返し要素数は 3 である．

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

3.2 繰り返しコードの識別方法

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．繰り返しコードとみなされるパターンは，大別すると 2 種類存在する．1 つめは，繰り返し要素が 1 つの文であるような繰り返しコードである．3.1 節の例で示した繰り返しコードが，これに該当する．2 つめは，繰り返し要素が複数の文であるような繰り返しコードである．

例えば以下のソースコードは，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．

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

また，以下のソースコードは，ネスト内に 2 つのメソッド呼び出し文を含む if 文を繰り返し要素とする繰り返しコードである．

```
if (null != storepass) {
  cmd.createArg().setValue(" -storepass ");
  cmd.createArg().setValue(storepass);
}
if (null != storetype) {
  cmd.createArg().setValue(" -storetype ");
  cmd.createArg().setValue(storetype);
}
```

3.3 類似した文

繰り返しコードを識別する際，文が類似しているかどうか判定する必要がある．本研究では，基本的に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，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

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*;

メソッド呼び出し文

メソッド呼び出し文は，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によって，表 1 に示す 2 通りの記述形式が存在する．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，表 1 に示すように，初期化を行う場合と行わない場合がある．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．ただし，式の種類とは，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，二項演算式などを指す.

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．名前が類似しているかどうかの判定方法については後述する.

return 文

予約語である return に後続する式の種類が同じであれば，類似する文とみなす.

throw 文

return 文と同様，予約語である throw に後続する式の種類が同じであれば，類似する文とみなす.

代入文については，名前が類似するかどうかの判定を行う．既存研究 [24] 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．本研究でもこの方法を用いる．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．例えば，`CamelCase` は `Camel` と `Case` の 2 つの単語から構成される．2 つ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みなす.

- 分割後の単語の数（分割数）が 1 語である．例えば `x` と `y` は類似した文字列である.
- 分割数が同じであり，1 箇所以上同じ単語が出現する．例えば `strTmp` と `strMsg` は類似した文字列である.

- 最初か最後の単語が一致する．例えば `srcPixels` と `dstInPixels` は類似した文字列である．

4　調査手法

この章では，ソースコードの中の繰り返しコードがどのように修正され，進化したのかを調査する方法について述べる．

4.1　調査手法の概要

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

出力するデータの詳細を，以下に示す．

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

これより，調査の流れを説明していく．

STEP1：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集合 $R = \{r_1, r_2, \cdots, r_n\}$ 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．

STEP2：リビジョン群 R 内の隣り合うリビジョン $r_i, r_{i+1} (0 < i < n)$ について，リビジョン間におけるソースコード中の繰り返しコードの変化を調べる．

STEP2A：各繰り返しコードについて，修正が加えられたかどうか判定する．

STEP2B：修正が加えられた繰り返しコードについては，修正後も繰り返しコードであるかどうか調べる．修正後も繰り返しコードである場合，修正後であるリビジョン r_{i+1} におけるその繰り返しコードの情報を取得する．

STEP2C：修正が加えられなかった繰り返しコードについては，リビジョン間で構造が変化することはない．しかし，ソースコード中の他の箇所に修正が加えられることで，その繰り返しコードのソースコード上での位置は変移する．そのため，

リビジョン間での位置の変化について調べ，繰り返しコードのリビジョン r_{i+1} での位置を取得する．

STEP3：各繰り返しコードの生成から消滅までの進化に関するデータをまとめる．

この手順を図 8 に示す．

4.2 調査手法の詳細

4.2.1 STEP1：リビジョン群 R の構築

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集合 R に追加する．

4.2.2 STEP2：隣り合うリビジョン間における繰り返しコードの変化

STEP2A：繰り返しコードへの修正の有無

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行番号が重なっていたら，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，前後の行のうちいずれかがリビジョン r_i の繰り返しコードと重なっていたら，その繰り返しコードに修正があったとみなす．

STEP2B：修正があった繰り返しコードの修正後の構造

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} における行番号を取得する．修正箇所の変更後の行番号がリビジョン r_{i+1} の繰り返しコードに内包されていたら，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，Algorithm1，2 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．

Algorithm1，2 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

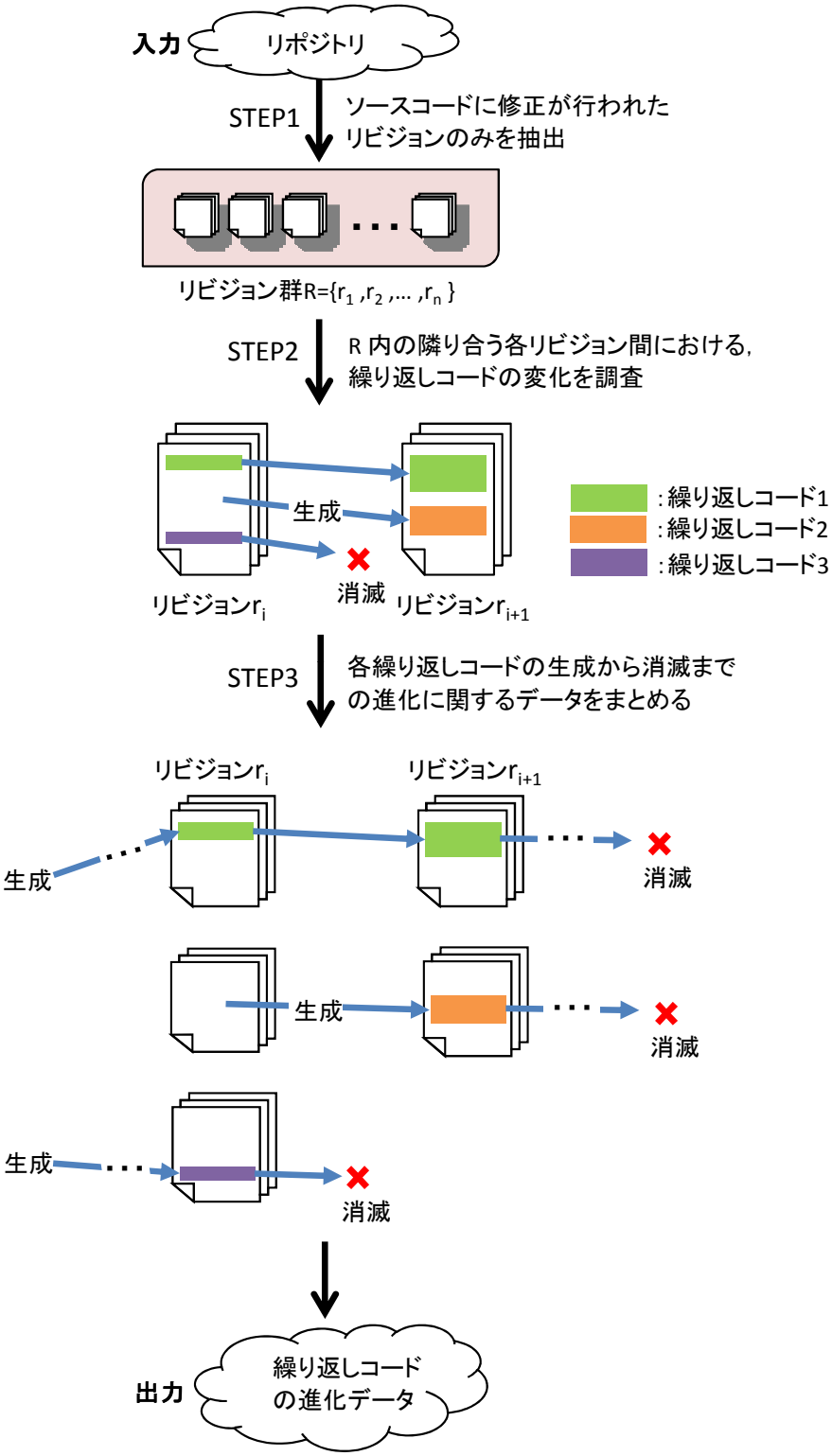

入力
リポジトリ
STEP1
ソースコードに修正が行われた
リビジョンのみを抽出
リビジョン群R={r1 ,r2 ,… ,rn }
STEP2
R 内の隣り合う各リビジョン間における,
繰り返しコードの変化を調査
生成
消滅
リビジョンri
リビジョンri+1
:繰り返しコード1
:繰り返しコード2
:繰り返しコード3
STEP3
各繰り返しコードの生成から消滅まで
の進化に関するデータをまとめる
リビジョンri
リビジョンri+1
生成
消滅
生成
消滅
生成
消滅
出力
繰り返しコード
の進化データ

図 8: 手順の流れ

r_i に含まれる 1 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行の行番号集合を返す．関数 $getAfterLines$ は，引数として行番号集合を取り，その行番号集合の修正後であるリビジョン r_{i+1} での行番号集合を返す．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 修正があった繰り返しコードの修正後の状態判定

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
```

STEP2C：修正がなかった繰り返しコードの位置の変移

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．

4.2.3 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変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滅までの進化データを導出す

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
```

る．進化データとは，調査において出力に必要なデータのことを指す．

4.3 実装

SVNKit[25] 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．SVNKit は Subversion の全機能を実装しており，主に Subversion のコピー，アクセス，リポジトリ操作を扱うための API を提供する．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，3 章で説明した手法を用いて取得する．

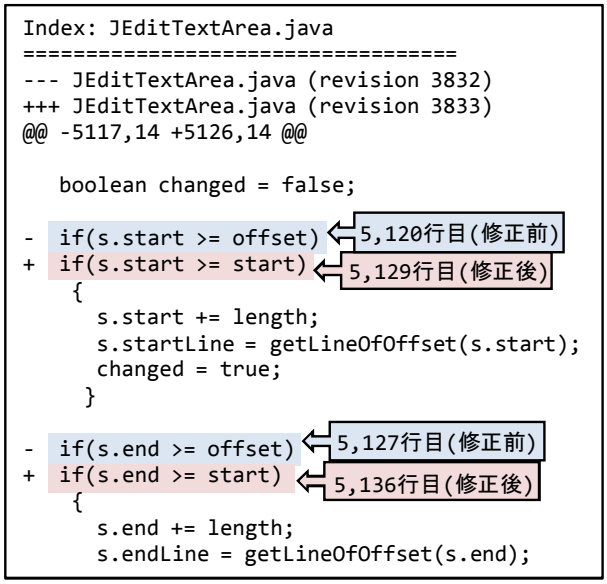
4.4 調査手法を用いて得られる繰り返しコードの進化例

繰り返しコードに修正が加えられた場合の Diff およびソースコード例を図 9 に示す．

まず図 9(a) より，リビジョン 3,832 の 5,120 行目と 5,127 行目に修正が行われていることがわかる．また，図 9(b) より，リビジョン 3,832 の 5,120 〜5,132 行目は，赤色の括弧で囲まれた部分を繰り返し単位とする繰り返し要素数 2 の繰り返しコードとなっていることがわかる．これら 2 つの修正箇所とリビジョン 3,832 における繰り返しコード (図 9(b)) の行番号は重なっているため，この繰り返しコードには，5,120 行目と 5,127 行目の 2 箇所に修正があったとみなされる．

さらに，図 9(a) より，図 9(b) で示した繰り返しコードに対する修正箇所は，修正後のリビジョン 3,833 ではそれぞれ 5,129 行目と 5,136 行目になっていることがわかる．また，図 9(c) より，リビジョン 3,833 の 5,129 〜5,141 行目は，赤色の括弧で囲まれた部分を繰り返し単位とする繰り返し要素数 2 の繰り返しコードとなっていることがわかる．これら 2 つの

修正箇所の修正後の行番号はリビジョン 3,833 における繰り返しコード (図 9(c)) に内包されているので，図 9(b) で示した繰り返しコードは，図 9(c) で示した繰り返しコードとなって，リビジョン 3,833 でも繰り返しコードであり続けるとみなされる．



(a) リビジョン 3,832，3,833 間の Diff

```
5120 if(s.start >= offset)
5121  {
5122   s.start += length;
5123   s.startLine = getLineOfOffset(s.start);
5124   changed = true;
5125  }
5126
5127 if(s.end >= offset)
5128  {
5129   s.end += length;
5130   s.endLine = getLineOfOffset(s.end);
5131   changed = true;
5132  }
```

(b) リビジョン 3,832 のソースコードの一部

```
5129 if(s.start >= start)
5130  {
5131   s.start += length;
5132   s.startLine = getLineOfOffset(s.start);
5133   changed = true;
5134  }
5135
5136 if(s.end >= start)
5137  {
5138   s.end += length;
5139   s.endLine = getLineOfOffset(s.end);
5140   changed = true;
5141  }
```

(c) リビジョン 3,833 のソースコードの一部

図 9: 繰り返しコードに修正がある場合の例

5 実験

5.1 実験目的

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検証することで，繰り返しコードの進化に見られる傾向を調査する．

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

5.2 実験対象

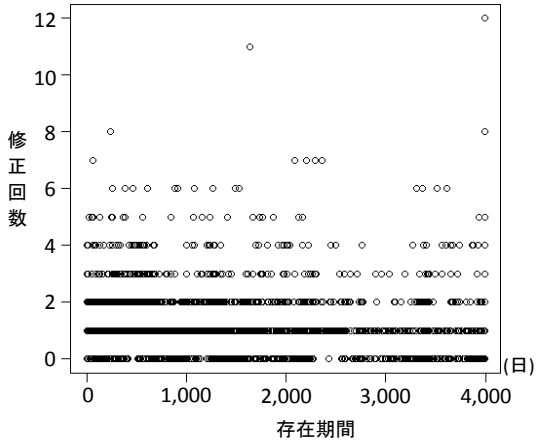
本研究では，3 つのオープンソースプロジェクトを対象として実験を行った．対象としたプロジェクトを表 2 に示す．これらのプロジェクトを選択した基準は以下の通りである．

1. バージョン管理システム Subversion を用いて開発を行っている．
2. 開発に Java のみを用いている．

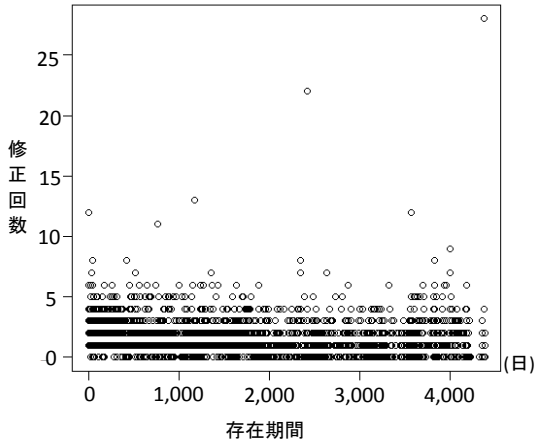
SVNKit は Java で開発されたプロジェクトのみに対応しているため，実験対象を Java プロジェクトに限定した．

表 2: 実験対象プロジェクト

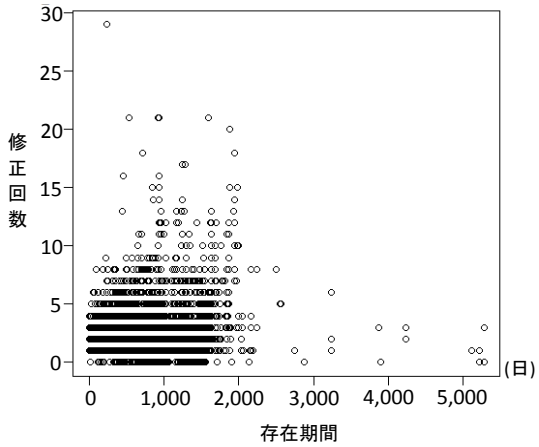
プロジェクト名	リビジョン番号		対象リビジョン数	行数		計測時間 (min)
	開始	最終		開始	最終	
jEdit	3,791	21,981	5,292	57,837	183,006	1,099
Ant	267,548	1,233,420	12,621	7,864	255,061	1,798
ArgoUML	2	19,893	17,731	20,287	369,583	3,898



(a) jEdit



(b) Ant



(c) ArgoUML

図 10: 各繰り返しコードに対する存在期間と修正回数

表 3: 繰り返しコードの存在期間と修正回数の相関係数

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

5.3 実験結果

5.3.1 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

繰り返しコードの存在期間と修正回数の相関図を図 10 に示す．図 10 では，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 10 から，繰り返しコードの存在期間と修正回数には明らかに相関がない．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ない．

5.3.2 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

RQ2 について検証するために，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修正が加えられたもの

一部： 一部の繰り返し要素に修正が加えられたもの

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

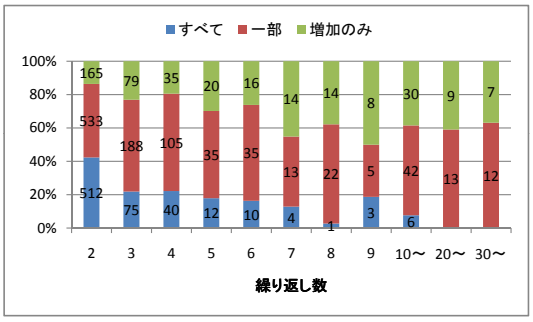
ただし，繰り返し要素に修正が加えられ，かつ，修正によって要素数が変化したものについては，すべてもしくは一部に含まれる．

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 11 に示す．図 11 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 11 から，繰り返し要素数が少ない繰り返しコードほどすべてに修正が加わる割合が高いことがわかる．

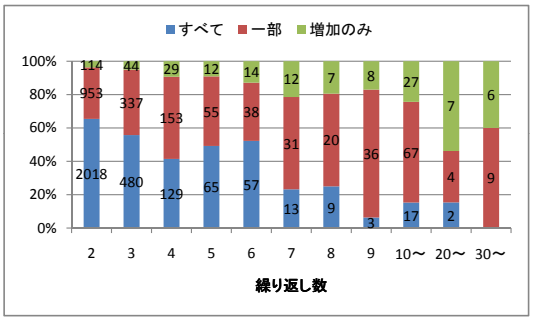
次に，修正後も繰り返しコードであるものを，以下のように更に 3 つに分類した．

1. 修正前後で繰り返し要素数が変化しなかったもの
2. 修正によって繰り返し要素数が増加したもの
3. 修正によって繰り返し要素数が減少したもの

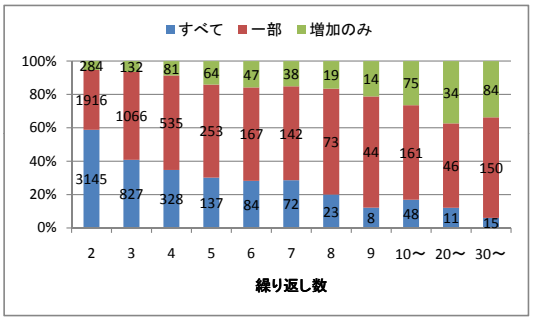
このうち，1. および 2. について，それぞれ修正内容の内訳を繰り返し要素数別に調査した．



(a) jEdit



(b) Ant



(c) ArgoUML

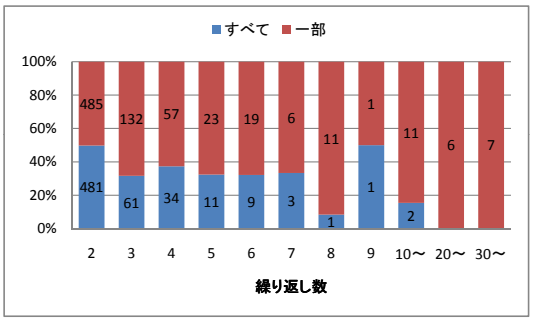
図 11: 修正内容の内訳 (繰り返し要素数別)

修正前後で繰り返し要素数が変化しなかったもの

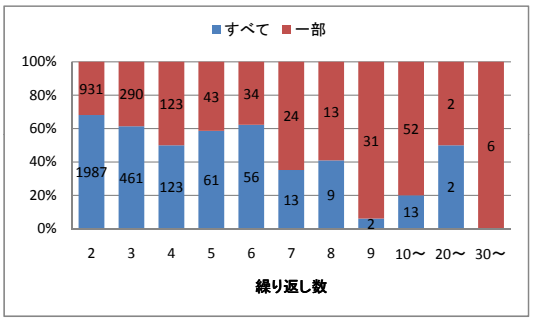
修正前後で繰り返し要素数が変化しなかったものについて，その修正内容の内訳を繰り返し要素数別にまとめた．その結果を図 12 に示す．図 12 の横軸は繰り返しコードの繰り返し要素数を，縦軸はその繰り返し要素数からなる繰り返しコードのうち，修正後も繰り返し要素数が変化しなかったものに対する修正内容の内訳を表している．図 12 からは，図 11 と同様に，繰り返し要素数が少ない繰り返しコードほどすべてに修正が加わる割合が高いということがわかる．

修正によって繰り返し要素数が増加したもの

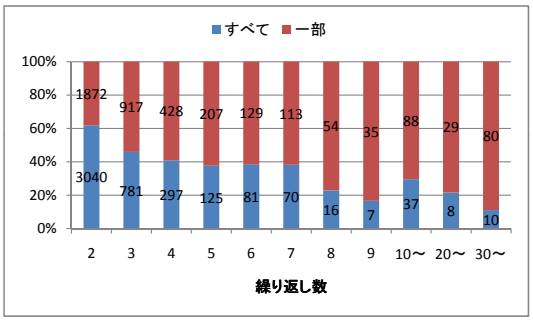
修正によって繰り返し要素数が増加したものについて，その修正内容の内訳を繰り返し要素数別にまとめた．その結果を図 13 に示す．図 13 の横軸は繰り返しコードの繰り返し要素数を，縦軸はその繰り返し要素数からなる繰り返しコードのうち，修正によって繰り返し要素数が増加したものに対する修正内容の内訳を表している．図 13 から，どの繰り返し要素数についても，6 割以上が増加のみの修正であることがわかる．また，繰り返し要素数が少ない繰り返しコードほどすべてに修正が加わる割合が高いということもわかる．



(a) jEdit

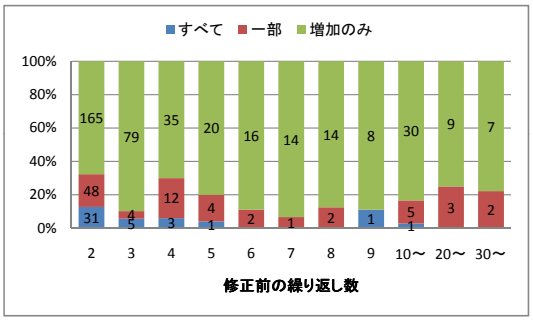


(b) Ant

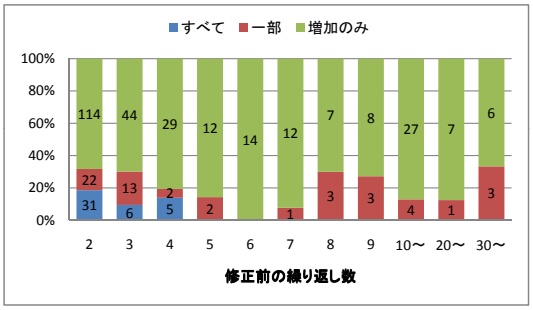


(c) ArgoUML

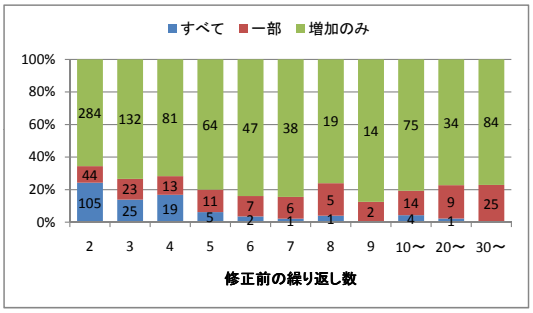
図 12: 修正前後で繰り返し要素数が変化しない修正内容の内訳 (繰り返し要素数別)



(a) jEdit

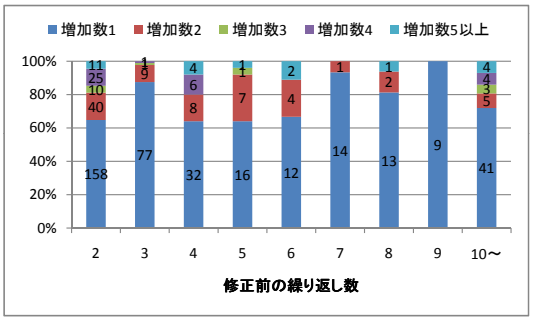


(b) Ant

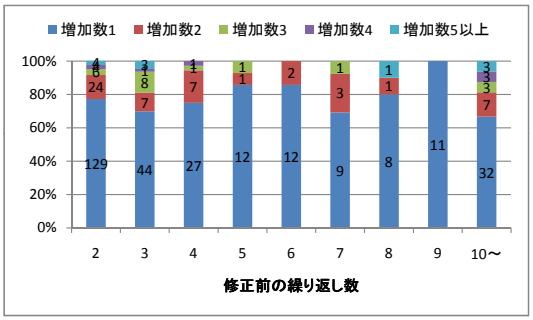


(c) ArgoUML

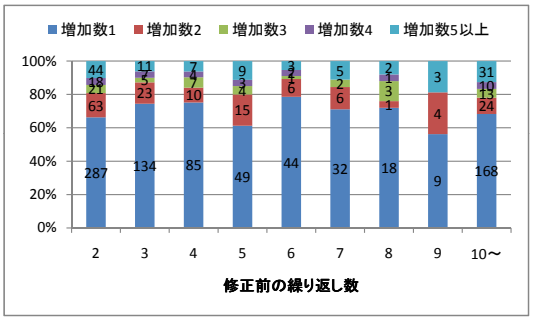
図 13: 修正によって繰り返し要素数が増加する修正内容の内訳 (繰り返し要素数別)



(a) jEdit



(b) Ant



(c) ArgoUML

図 14: 繰り返しの増加数 (繰り返し要素数別)

さらに，修正によって繰り返し要素数が増加したものについて，そのときの繰り返し要素の増加数の内訳をまとめた．その結果を図 14 に示す．図 14 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の，修正による繰り返し要素の増加数を表している．繰り返し要素の増加数については，図 14 より，全体的に増加数 1 のものが多いが，どの繰り返し要素数についても，2 割ほどは繰り返し要素が 2 以上増加していることがわかる．

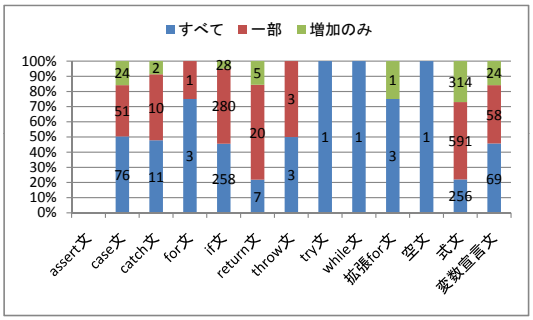
5.3.3 RQ3 : 繰り返しコードを構成する文の種類は，その繰り返しコードの進化に影響を与えるか

RQ3 について検証するために，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．

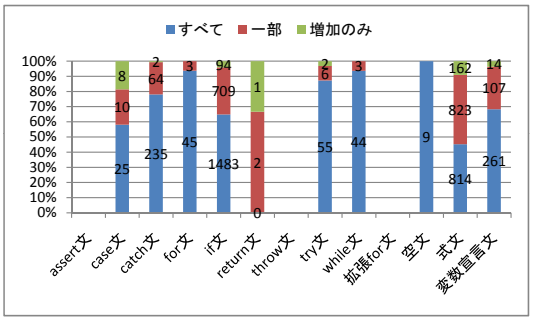
修正内容は，RQ2 に対する検証のときと同様，以下の 3 種類に分類される．

すべて： すべての繰り返し要素に修正が加えられたもの

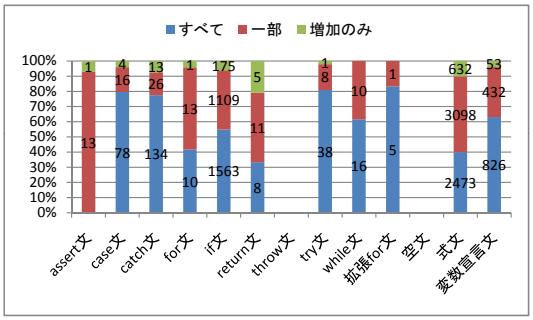
一部： 一部の繰り返し要素に修正が加えられたもの



(a) jEdit



(b) Ant



(c) ArgoUML

図 15: 修正内容の内訳 (文の種類別)

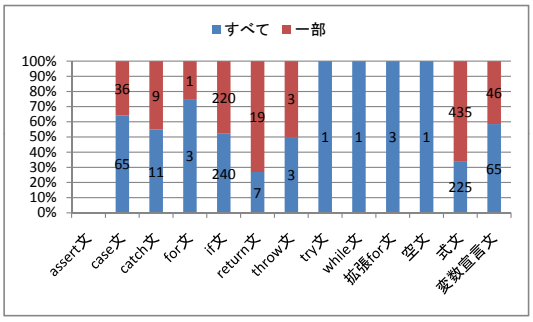
増加のみ：既存の繰り返し要素に対する修正はなかったが，修正によって繰り返し要素数が増加したもの

ただし，繰り返し要素に修正が加えられ，かつ，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．

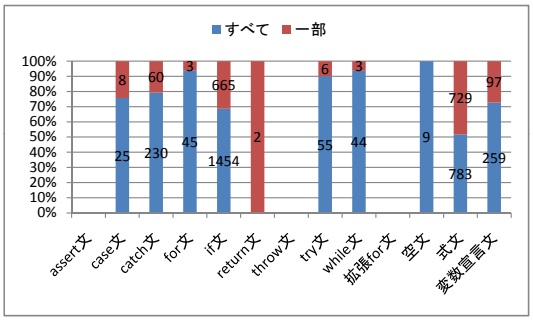
修正後も繰り返しコードであるものについて，文の種類別に修正内容の内訳をまとめたものを図 15 に示す．図 15 の横軸は繰り返しコードを構成する文の種類を，縦軸はその文を含む繰り返しコードに対する修正内容の内訳を表している．ただし，1 つの繰り返しコードに複数の文が含まれる場合は，それらすべての文の項目について，その繰り返しコードに対する修正内容を反映している．また，棒グラフがない項目については，その文を含む繰り返しコードに修正が行われなかったことを意味する．図 15 から，式文は一部への修正が 5 割以上と，高い割合を占めることがわかる．一方，case 文， catch 文， try 文， while 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．

次に，修正後も繰り返しコードであった繰り返しコードを，RQ2 に対する検証のときと同様，以下のように更に 3 つに分類した．

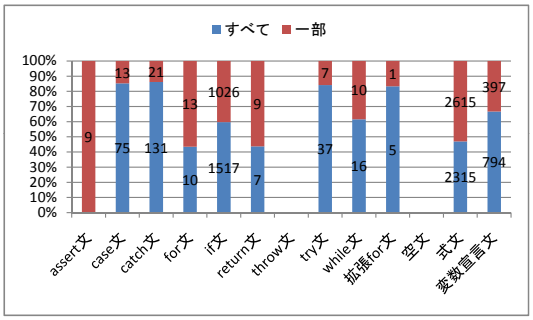
1. 修正前後で繰り返し要素数が変化しなかったもの



(a) jEdit



(b) Ant



(c) ArgoUML

図 16: 修正前後で繰り返し要素数が変化しない修正内容の内訳 (文の種類別)

2. 修正によって繰り返し要素数が増加したもの

3. 修正によって繰り返し要素数が減少したもの

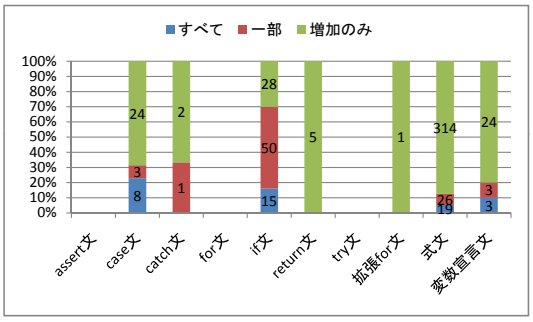
このうち，1. および 2. について，それぞれ修正内容の内訳を文の種類別に調査した．

修正前後で繰り返し要素数が変化しなかったもの

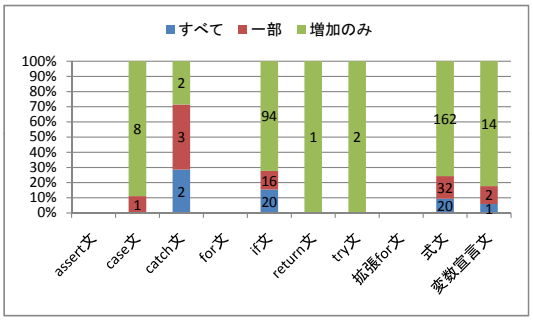
修正前後で繰り返し要素数が変化しなかったものについて，その修正内容の内訳を文の種類別にまとめた．その結果を図 16 に示す．図 16 の横軸は繰り返しコードを構成する文の種類を，縦軸はその文を含む繰り返しコードのうち，修正後も繰り返し要素数が変化しなかったものに対する修正内容の内訳を表している．図 16 からは，図 15 と同様に，式文には一部に修正が加わりやすい傾向があり，case 文， catch 文， try 文， while 文， 変数宣言文にはすべてに修正が加わりやすい傾向があるということがわかる．

修正によって繰り返し要素数が増加したもの

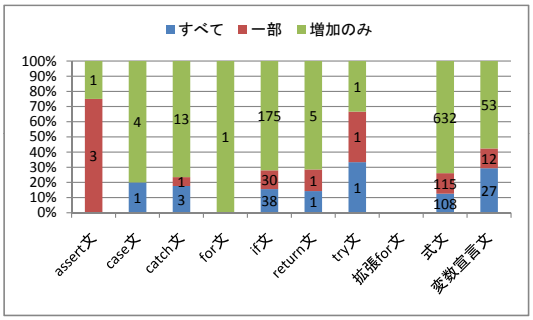
修正によって繰り返し要素数が増加したものについて，その修正内容の内訳を文の種類別にまとめた．その結果を図 17 に示す．図 17 の横軸は繰り返しコードを構成する文の種類



(a) jEdit



(b) Ant



(c) ArgoUML

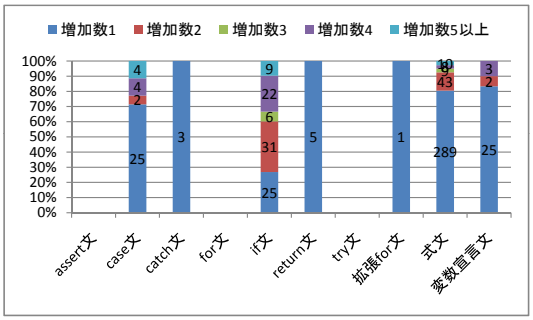
図 17: 修正によって繰り返し要素数が増加する修正内容の内訳 (文の種類別)

を，縦軸はその文を含む繰り返しコードのうち，修正によって繰り返し要素数が増加したものに対する修正内容の内訳を表している．図 17 から，繰り返し要素が増加する場合は全体的に増加のみの修正が多いが，どの文とも，既存の繰り返し要素に修正が加わっている場合もあるということがわかる．

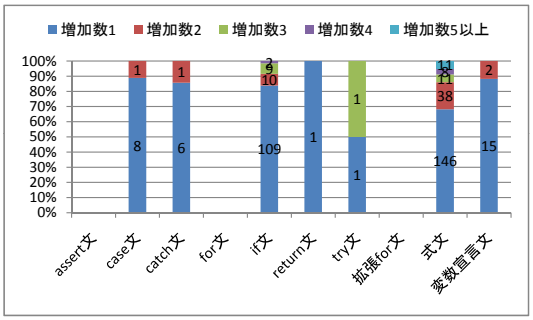
さらに，修正によって繰り返し要素数が増加したものについて，そのときの繰り返し要素の増加数をまとめた．その結果を図 18 に示す．図 18 の横軸は繰り返しコードを構成する文の種類を，縦軸はその文を含む繰り返しコードの，修正による繰り返し要素の増加数を表している．繰り返し要素の増加数については，図 18 より，全体的に増加数 1 のものが多いが，式文や if 文のように，どのプロジェクトについても繰り返し要素が 2 以上増加しているものが 2 割ほどある文も存在することがわかる．

5.3.4 RQ4：繰り返し要素のトークン数はどのくらいの大きさか

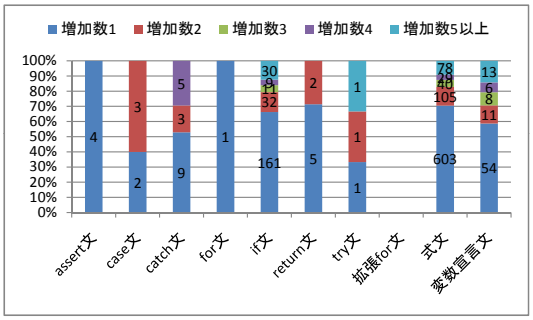
各プロジェクトに含まれる繰り返し要素のトークン数の内訳を図 19 に示す．横軸がプロジェクト名，縦軸がそのプロジェクトに含まれる繰り返し要素のトークン数の内訳を表している．図 19 からは，繰り返し要素の 7 割以上がトークン数 20 以下という比較的小さな単位で構成されているということがわかる．



(a) jEdit



(b) Ant



(c) ArgoUML

図 18: 繰り返し要素の増加数 (文の種類別)

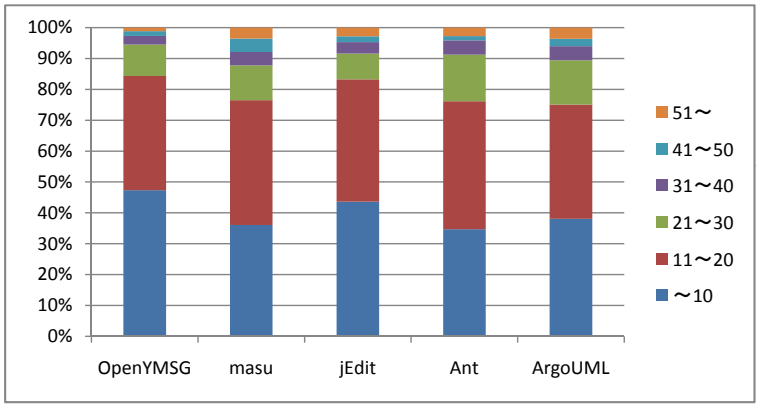


図 19: 繰り返し要素のトークン数

5.3.5 RQ に対する回答

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**

実験では，繰り返しコードの存在期間の長さと修正回数との間に特筆すべき相関関係を見出すことはできなかった．このことから，繰り返しコードは，長く存在しているからといって修正が多く加えられるわけではないということがいえる．

また，RQ1 に関する実験の中で，繰り返しコードの多くは最低でも 1 回以上の修正が加わっているということが明らかになった．そこで，これについて更に追求したところ，実際に 65 〜89 %の繰り返しコードが，進化の中で 1 回以上修正されているという結果が得られた．このことから，多くの繰り返しコードは，ソフトウェアの進化に伴いそれ自身も進化を遂げているということがいえる．

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**

実験から，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修正が加わる傾向が強いことがわかった．このことから，繰り返し要素数が少ない繰り返しコードは，1 つの繰り返し要素に修正が加えられた場合に，他のすべての繰り返し要素に対しても同様の修正が行われやすいといえる．

また，どの繰り返し要素数についても，修正後に新たに繰り返し要素が 5 以上増加する割合は 1 割に満たないことが明らかになった．このことから，繰り返しコードに対して，一度の修正で大量に機能が追加されることは稀であるということが考えられる．

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**

実験から，case 文や while 文，変数宣言文などのように，すべての繰り返し要素に修正が加わりやすい文の種類が多く存在することがわかった．このことから，様々な文の繰り返しコードについて，1 つの繰り返し要素に修正が加えられた場合に，他のすべての繰り返し要素に対しても同様の修正が加えられる事例が存在するといえる．

RQ4：**繰り返し要素のトークン数はどのくらいの大きさか**

実験により，繰り返し要素のトークン数は，小さな値をとる傾向が強いことがわかった．

6 考察

6.1 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．また，実際に手動でいくつかの繰り返しコードについてその進化を追っていき，その中から修正支援が有用であると考えられる進化事例を紹介する．

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 1 回以上の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と考えられる．

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 1 つずつ修正箇所を提示するような支援が有用なのではないかと考えられる．

実際に，このような修正支援が有効な事例として，図 20 に示す進化が見つかった．図 20(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 20(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．この進化事例では，修正によって，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．このような操作は，コピーアンドペーストによって 1 ステップで実現することが不可能であり，ユーザは，各繰り返し要素に対して同様の修正を行うために，複数回のコピーアンドペーストやコーディングを行わなければならない．これは，コストがかかる上に，タイプミスや修正漏れによりバグが発生する可能性がある．一方で，ある繰り返し要素に修正が行われた際に，他のすべての繰り返し要素に対して 1 つずつ修正の提案を行う支援を導入した場合は，ユーザは，システム側が提案する修正を各要素に対して 1 回ずつ承認するだけで，自動的に修正を加えることができる．したがって，繰り返し要素数が少ない繰り返しコードに対しては，このような修正支援が有用であるといえる．

```
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 20: 繰り返しコードの修正事例 1

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，修正支援が必要なのではないかと考えられる．

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のではないかと考えられる．

実際に，繰り返し要素の自動挿入が可能な事例として，図 21 に示す進化が見つかった．図 21(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．この進化事例では，修正によって，新たな繰り返し要素であるメソッド呼び出し文が 1 つ追加されている．このような修正を手動で行う場合，ユーザは，既存箇所をコピーし，それをペーストによって貼り付け，その後，引数を入力するという 3 ステップの処理を行う必要がある．また，この操作では，引数の修正漏れによりバグが発生する可能性もある．一方で，繰り返し要素を自動的に追加するといった支援を導入した場合は，ユーザは，繰り返し要素を追加する機能を選択するだけで，新たな繰り返し要素を追加することができる．新たに加えられた繰り返

```
addJavaFiles(files, TASKDEFS_ROOT);
addJavaFiles(files, new File(TASKDEFS_ROOT, "compilers"));
addJavaFiles(files, new File(TASKDEFS_ROOT, "condition"));
addJavaFiles(files, DEPEND_ROOT);
addJavaFiles(files, new File(DEPEND_ROOT, "constantpool"));
addJavaFiles(files, TYPES_ROOT);
addJavaFiles(files, FILTERS_ROOT);
addJavaFiles(files, UTIL_ROOT);
addJavaFiles(files, new File(UTIL_ROOT, "depend"));
addJavaFiles(files, ZIP_ROOT);
addJavaFiles(files, new File(UTIL_ROOT, "facade"));
```

(a) 修正前

```
addJavaFiles(files, TASKDEFS_ROOT);
addJavaFiles(files, new File(TASKDEFS_ROOT, "compilers"));
addJavaFiles(files, new File(TASKDEFS_ROOT, "condition"));
addJavaFiles(files, DEPEND_ROOT);
addJavaFiles(files, new File(DEPEND_ROOT, "constantpool"));
addJavaFiles(files, TYPES_ROOT);
addJavaFiles(files, FILTERS_ROOT);
addJavaFiles(files, UTIL_ROOT);
addJavaFiles(files, new File(UTIL_ROOT, "depend"));
addJavaFiles(files, ZIP_ROOT);
addJavaFiles(files, new File(UTIL_ROOT, "facade"));
addJavaFiles(files, INPUT_ROOT);
```

(b) 修正後

図 21: 繰り返しコードの修正事例 2

し要素の第 2 引数は他の繰り返し要素のものと関連がないため，修正を行う際は，ユーザ自身が入力を行う必要があるが，それを含めても，機能の選択と引数の入力という 2 ステップの処理で，追加を行うことが可能になる．また，新たな繰り返し要素の挿入後，システム側が引数への入力を促すことで，引数の修正漏れを防ぐことができる．したがって，繰り返し要素を追加する修正には，このような支援が有用であるといえる．

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，偶然の一致によりコードクローンとなってしまう関連性の低いクローンペアが多く検出されてしまい，実用的ではない．したがって，通常はそのような規模の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をつけてい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することができた．

また，繰り返し要素はトークン数が小さいという傾向があることから，多くの繰り返しコードは，既存の検出手法では検出することが難しい．したがって，このようなものに対し

ても修正支援を可能にするために，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．

7　妥当性への脅威

7.1　対象ソフトウェアの数について

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．

7.2　コード分析について

本実験では，繰り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．例えば，繰り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，繰り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．

7.3　繰り返しコードの消滅について

本実験では，修正が加えられた繰り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，その繰り返しコードが“消滅”したとみなしている．

- 繰り返しコードを構成していたコード片は残存しているが，修正によって繰り返しコードではなくなった
- 繰り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り返しコードとなる可能性があるので，本来ならば“消滅”とは切り離して考えるのが適切である．したがって，“繰り返しコードの消滅”についての定義が適切ではない本実験では，一度消滅した後，再び復活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ことができていないため，妥当な結果が得られていない可能性がある．

8　関連研究

大森らは，ソフトウェア進化研究の動向を調査するため，独自の基準を設けてそれらの分類を行った [26]．この中で著者らは，コードの進化を扱ったいくつものソフトウェア進化研究が行われていることを述べており，特に，静的コード解析・動的プログラム解析などの言語モデルや式に基づく解析的分野や，データ処理・リポジトリマイニングなどの統計や事例に基づく実証的分野において盛んに行われていると報告している．また，ソフトウェアが進化を続ける以上，それに関連する研究は今後ますます増加するであろうと主張している．

Kim らは，オープンソースソフトウェアの複数のバージョンに対して，コードクローンの進化に関する調査を行った [3]．この中で著者らは，ソースコードに含まれるコードクローンについて以下のことを報告している．

- あるバージョンでコードクローンになったが，その後異なった修正が加えられたことにより，コードクローンでなくなる場合がある．
- コードクローンの 36 〜38 % は，各コードクローンに対して同時に修正が行われる．
- 長く存在するコードクローンほど，各コードクローンに対して同時に修正が行われやすい．

川口らは，バージョン管理システムを用いたコードクローン進化の分析手法を提案している [27]．この研究で提案されている手法は，コードクローン分析を過去の時点に遡って順次適用し，各時点間のコードクローンについてその変更履歴を抽出するというものである．なお，コードクローン検出には，CCFinder が用いられている．研究の中で著者らは，実際に提案手法を PostgreSQL に適用し，コードクローンがどのような進化を遂げてきたのかを特定することに成功している．

Duala-Ekoko らは，CloneTracker というコードクローンの進化を追跡・管理するエディタツールを Eclipse のプラグインとして開発している [7]．Clone Tracker は，ソースコードの修正に伴うコードクローンの進化を管理し，あるコード片に対して修正が行われたときに，それとコードクローン関係にあるコード片への同時修正を提案するため，そのコード片が属するクローンセットの情報をユーザに向けて通知する．

Toomim らは，あるコード片を修正すると，それとコードクローン関係にあるすべてのコード片に対して，同様の修正を自動的に施すエディタを開発している [6]．しかし，現段階では，このエディタはユーザの入力を，対応する各コード片に対してそのまま反映させるため，変数名や関数名等もまったく同じである，完全一致のコードクローンに対してしか適用することができない．

9　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，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．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．この結果から，繰り返し要素数の少ない繰り返しコードについては，1 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すべての繰り返し要素について 1 つずつ修正の提案を行うといった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については，新たに繰り返し要素を挿入するといった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，コード分析を用いて具体的に調査する．
- 一度消滅した後，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

本研究を行うにあたり，理解あるご指導を賜り，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，有益かつ的確なご助言を頂きました岡野浩三准教授に深く感謝申し上げます.

本研究に多大なるご助言およびご指導を頂きました井垣宏特任准教授に深く感謝致します.

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.

本研究を行うにあたり，多大なるご助言，ご助力を頂きました，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博士後期課程 1 年の堀田圭佑氏に深く感謝申し上げます.

本研究を行うにあたり，適切なご助言，ご助力を頂きました，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博士前期課程 2 年の佐々木唯氏に深く感謝申し上げます.

その他，楠本研究室の皆様のご助言，ご協力に心より感謝致します．また，本研究に至るまでに，講義，演習，実験等で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，この場を借りて心から御礼申し上げます.

参考文献

[1] 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. 91, No. 6, pp. 1465–1481, Jun. 2008.

[2] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 83–92, Aug. 2004.

[3] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 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, Sep. 2005.

[4] Yoshiki Higo and Shinji Kusumoto. How Often Do Unintended Inconsistencies Happen? –Deriving Modification Patterns and Detecting Overlooked Code Fragments–. In *28th IEEE International Conference on Software Maintenance*, pp. 222–231, Sep. 2012.

[5] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 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, Mar. 2006.

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

[7] Duala-Ekoko, Ekwa, Robillard, and Martin P. Clonetracker: tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering*, pp. 843–846, May. 2008.

[8] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, Vol. 49, pp. 985–998, Sep. 2007.

[9] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Folding Repeated Instructions for Improving Token-Based Code Clone Detection.

IEEE International Workshop on Source Code Analysis and Manipulation, pp. 64–73, Sep. 2012.

[10] Yui Sasaki, Tomoya Ishihara, Keisuke Hotta, Hideaki Hata, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Preprocessing of Metrics Measurement Based on Simplifying Program Structures. In *International Workshop on Software Analysis, Testing and Applications*, pp. 120–127, Dec. 2012.

[11] Subversion. `http://subversion.apache.org/`.

[12] H. Sajnani J. Ossher and C. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, pp. 283–292, Sep. 2011.

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

[14] 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.

[15] 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, Jul. 2002.

[16] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pp. 368–377, Mar. 1998.

[17] 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.

[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] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 109–118, Aug. 1999.

[20] Lingxiao Jiang, G. Misherghi, Zhendong Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105, May. 2007.

[21] 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 2009, pp. 97–104, Sep. 2009.

[22] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working conference on Reverse Engineering*, pp. 301–309, Oct. 2001.

[23] J. Mayrand, C. LeBlanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th IEEE International Conference on Software Maintenace*, pp. 244–253, Nov. 1996.

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

[25] SVNKit. `http://svnkit.com/`.

[26] 大森隆行, 丸山勝久, 林晋平, 沢田篤史. ソフトウェア進化研究の分類と動向. コンピュータソフトウェア, Vol. 29, No. 3, pp. 3–28, Jul. 2012.

[27] 川口真司, 松下誠, 井上克郎. 版管理システムを用いたクローン履歴分析手法の提案. 電子情報通信学会論文誌 D, Vol. 89, No. 10, pp. 2279–2287, Oct. 2006.