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.

修正実績に基づく重複コード除去支援の試み

肥後　芳樹† 　楠本　真二†

† 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　〒 565-0871 吹田市山田丘 1-5
E-mail: †{higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　これまでに重複コードに対するさまざまなリファクタリング支援手法が提案されている．それらは，現在のソースコードの状態を元に，リファクタリング可能な重複コードを特定し，ユーザに提示する．しかし，必ずしもリファクタリングする価値がある重複コードを提示できているとはいえない．本研究では，過去に同様の修正が加わった実績がある重複コードをリファクタリング候補として提示する．過去に同様の修正を必要としたという事実は，重複コードをリファクタリングする動機の一つになると考えられる．また，小規模な実験を行い，修正実績に基づいて提示された重複コードがリファクタリング対象として適切な場合が多いことを確認した．
キーワード　重複コード，リファクタリング，ソフトウェアリポジトリマイニング

An Approach to Support Duplicate Code Removal Based on Their Evolutional Information

Yoshiki HIGO† and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
E-mail: †{higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　Previous research studies have proposed a variety of refactoring support techniques for duplicate code. They identify duplicate code as refactoring candidates and suggest how to refactor them by analyzing the latest source code. However, they often identify duplicate code that are not suited for refactoring. In this paper, we propose a new technique to identify duplicate code that should be refactored. The proposed method considers the past modification record of duplicate code by analyzing the software repository of the target system. We have conducted a small experiment on an open source system and confirmed that most of the identified duplicate code were appropriate for refactoring.
Key words　Duplicate code, Refactoring, Mining Software Repositories

1. はじめに

重複コードの存在は，ソースコードの保守を困難にする場合がある [8]．その理由の一つとして，ある重複コードに対して修正が行われた場合に，それと対応する重複コードに対しても同様の修正を行うかどうかを考慮しなければならないことが挙げられる．著者らは，実際に重複コードに対する修正漏れが起こっていることを実験により示した [10]．このような理由から，重複コードは，リファクタリング対象の一つとして位置づけられている [3]．

その一方で，重複コードをリファクタリングすることが必ずしもよいとは限らないという研究報告もある．Kim らは，オープンソースソフトウェアに対して重複コードの進化を分析し，下記の二点を明らかにした [7]．

- 短期間しか存在しない重複コードが存在する．それらは重複コードではなくなった後に別々の進化をたどる．
- 長期間存在する重複コードは，プログラミング言語の制限上存在し続けている場合があり，それらを一つにまとめることは必ずしも容易ではない．

また，Kapser らは，デバイスドライバはコピーアンドペーストにより作成することが一般的であり，それらは非常に安定しているコードであるため，コピーアンドペースト後にバグ修正が発生することはほとんどないと報告している [5]．

これらの先行研究より，全ての重複コードがリファクタリングを必要とするわけではないが，一部の重複コードは同時に同様の修正を要求することがあり，リファクタリングすべきであるといえる．本研究では，過去において同様の修正が行われた重複コードをリファクタリングの対象として特定する手法を

```
115   if (org.argouml.model.ModelFacade.isAInstance(target)) {
116     MInstance inst = (MInstance) target;                    DELETED
117 //  ((MInstance) target).setClassifier((MClassifier) element);
118
119     // delete all classifiers
120     Collection col = inst.getClassifiers();                 DELETED
121     if (col != null) {
122   Iterator iter = col.iterator();
```

(a) 修正前のソースコード

```
115   if (org.argouml.model.ModelFacade.isAInstance(target)) {
116     Object inst = /*(MInstance)*/ target;                   ADDED
117 //  ((MInstance) target).setClassifier((MClassifier) element);
118
119     // delete all classifiers
120     Collection col = ModelFacade.getClassifiers(inst);      ADDED
121     if (col != null) {
122   Iterator iter = col.iterator();
```

(b) 修正後のソースコード

図 1　オープンソースソフトウェアにおける実際の修正例 (同時に三つのファイルが同様に修正されていた)

提案する．提案手法は，たとえソースコード中に重複コードが存在していてもそれらに対して過去に同様の修正が行われていない場合は，リファクタリング候補として提示しない．よって，提案手法により提示されたリファクタリング候補は，最新バージョンのソースコードを解析することによって提示されたリファクタリング候補に比べて，開発者または保守管理者がリファクタリングを行うモチベーションが高いと著者らは考えた．

以下は本研究の貢献である．

- 修正実績に基づいてリファクタリング候補を特定する手法を提案した．提案手法はスケーラブルであり，実規模ソフトウェアからでも短時間で候補を検出できる．
- オープンソースソフトウェアに対して実験を行い，提案手法によって特定されたリファクタリング候補がその後実際にリファクタリングされていること，および，リファクタリングされていない場合でもリファクタリング候補として適切であることを確認した．

以下，本稿は次のように構成されている．2. では，本研究の動機を実際のコード修正例を用いて述べる．次に 3. では提案手法を説明する．4. では，オープンソースソフトウェアに対して行った実験について述べる．5. では，関連研究を紹介し，最後に 6. で本稿をまとめる．

2. 研究の動機

図 1 は，あるオープンソースソフトウェアにおける修正の例を表している．この修正では，116 行目と 120 行目に存在している二つの文が変更されている．また，同様の変更が三つのファイルにおいて行われていた．これら三つのファイルに対する変更は一つのコミットにおいて行われていた．

同様の修正を必要とすることは重複コードが存在することによる弊害の一つであるといわれている [10]．もし三つのうち，一つでも開発者が見逃してしまった場合には，ソースコード中に意図しない不整合が発生する．よって，同様の修正を必要とした重複コードはリファクタリングにより一つにまとめることがよいと著者らは考えた．一つにまとめることにより，今後同時に複数箇所に対して修正を行う必要がなくなるため，意図しない不整合は発生しなくなる．

本稿では，過去に同様の修正が行われた重複コードをリファクタリング候補として特定する手法を提案する．ここで「同様の修正が行われた重複コード」とは，修正前に重複関係を持ち，修正が加わった後でもその重複関係が維持されているコード片のグループを指す．

これまでにも重複コードのリファクタリングを支援する手法は提案されている [9]．提案手法と既存手法との違いを下記に示す．

- 既存手法は，あるバージョン (多くの場合は最新バージョン) に存在している重複コードをリファクタリングの候補として提示する．
- 提案手法は，過去に同様の修正が行われた重複コードをリファクタリングの候補として提示する．

提案手法が考慮している「同様の修正が行われた」ことが，提示された候補を開発者が実際にリファクタリングする動機につながると著者らは考えた．

修正の内容を取得するために，Unix の diff コマンドが用いられる場合が多い．しかし，ソースコードにおける修正の内容の取得には，diff コマンドでは不十分な場合がある．例えば，図 1 では，二つの文が修正されているがそれらの間にはコメント行および空白行がある．そのため，diff コマンドを用いると 116 行目の変更と 120 行目の変更が別の変更になってしまう．本研究では，このような場合でも 116 行目に対する変更と 120 行目に対する変更が一つの変更として特定される手法を提案する．

提案手法の特徴は，以下の二つである．

- 過去に同様の修正が行われたコードをリファクタリング候補として提示する．
- コメントや空白行を無視した上で過去の修正の内容を特定する．

3. 提 案 手 法

3.1 修正パターン

本研究では，リファクタリング候補を検出するために**修正パターン** [10] を用いる．修正パターンとは，ある修正によってあるコード片が異なるコード片に変化したことを表す変化のパターンである．

修正パターンは修正前のコード片および修正後のコード片の情報を持つ．なお，文献 [10] では，修正パターンのコード片は連続した行として定義されているが，本研究では，コード片は字句列とする．

また，効率的にリファクタリングする価値がある重複コードを特定するために，修正パターンに対して下記のメトリクスを定義する．

- **支持度** その修正パターンが過去に出現した回数．
- **修正後文数** 修正後コード片に含まれるプログラム文の数．
- **出現期間** その修正パターンが初めて出現した日時と最後に出現した日時の差．この値が大きいほど，同様の修正が長期にわたって出現したことを意味する．
- **ファイル数** その修正パターンが出現したファイルの数．

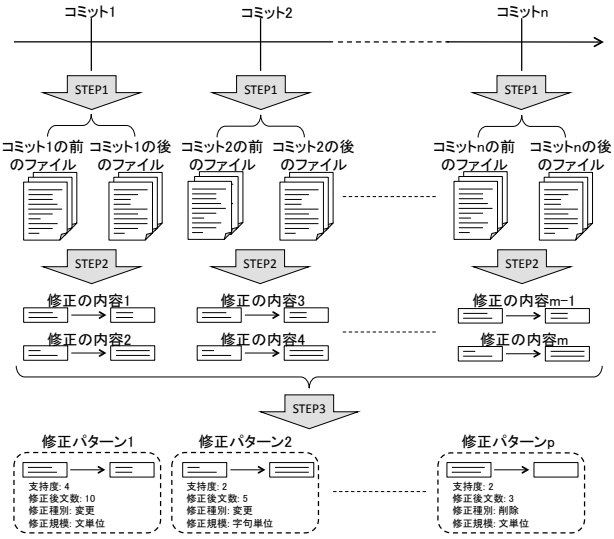

図 2　提案手法の概要

- **修正種別** その修正パターンが，**変更**，**追加**，**削除**のどれなのかを表す．修正前コード片と修正後コード片が共に空でない場合は変更である．修正前コード片が空である場合は追加，修正後コード片が空である場合は削除となる．
- **修正規模** その修正パターンが，字句単位のみの修正なのか，それよりも大きた単位の修正なのかを表す．字句単位の修正の場合は**字句**，より大きな修正の場合は**文**と表現する．

本研究では，修正パターンに関する上記メトリクスを用いて，リファクタリング候補をユーザに提示する．これらのメトリクスは，リファクタリングする価値がある重複コードを抽出するために用いるものであり，重複コードがどのようにリファクタリング可能かを推測するためのものではない．なお，本研究におけるリファクタリングは，メソッドの抽出やメソッドの引き上げ等を行うことによって，類似したコード片を一つにまとめることを指す．本研究では，重複コードのリファクタリング方法までは支援していない．ユーザは，提案手法により提示された重複コードがリファクタリングする価値があると判断できた場合，そのリファクタリング方法は自分自身で検討しなければならない．

3.2　修正パターンの導出

ここでは，修正パターンの導出方法について説明する．提案手法の入力は，リファクタリング候補を検出したいソフトウェアのリポジトリである．出力は，導出した修正パターンである．修正パターンの導出は下記の手順にて行われる．

- **STEP1** 各コミットにおいて修正されたソースファイルに対して，そのコミットの前後のリビジョンを取得．
- **STEP2** STEP1 で取得した各ソースファイル対を比較することにより，修正内容を抽出．
- **STEP3** STEP2 で抽出した修正内容を利用して，修正のパターンを導出．

以降，本節では，上記の手順を詳細に説明する．ここでは，入力として与えられたリポジトリを R とする．

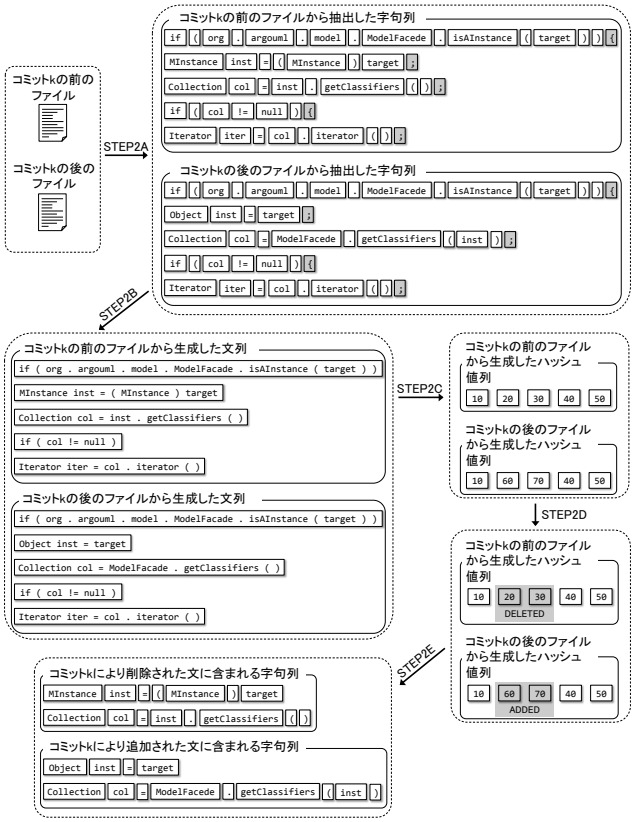

図 3　STEP2 の処理の例

STEP1: 各コミット前後のリビジョンの取得

R に含まれている全てのコミットの集合を $C(R)$ とする．R に n 個のコミットが存在する場合，$C(R)$ は下記の式で表される．

$$C(R) = \{c_1, c_2, \cdots, c_n\} \tag{1}$$

次に，各コミットにおいて修正されているソースファイル群を特定する．コミット c_k において，修正されたソースファイル群を $S(c_k)$ とする．コミット c_k において，m 個のソースファイルが修正されている場合，$S(c_k)$ は下記のように表される．

$$S(c_k) = \{f_1, f_2, \cdots, f_m\} \tag{2}$$

なお，$S(c_k)$ には，コミット c_k において削除されたソースファイルや追加されたソースファイルは含まれない．

STEP1 の出力は $S(c_k)(\forall c_k \in C(R))$ である．

STEP2: 修正内容を抽出

各コミットにおいて修正されたソースファイル群に対して，そのコミットの前後の内容を比較することにより，修正内容を抽出する．具体的には，STEP2 は下記の手順からなる．図 3 は STEP2 の処理の様子を表している．

- **STEP2A** コミット前後のリビジョンのソースファイルを字句解析し，字句列を得る．
- **STEP2B** コミット前後の字句列から，プログラム文のテキストを生成する．ここでプログラム文とは，セミコロン (“;”)，開き中括弧 (“{”)，閉じ中括弧 (“}”) で囲まれた字句列を結合した文字列である．なお，セミコロン，開き中括弧，閉じ中括弧自体は文には含めない．

- **STEP2C** STEP2B で生成した各文字列からハッシュ値を算出する．この処理により，修正前ファイルと修正後ファイルそれぞれからハッシュ値の列が得られる．
- **STEP2D** STEP2C で得られた二つのハッシュ値列に対して最長共通部分列検出アルゴリズム [1] を適用することにより，一致しない部分を特定する．
- **STEP2E** STEP2D で特定した一致しない部分を STEP2A の出力である字句列に対応づける．これにより，プログラム文の単位で一致しない修正前後の字句列を得る．

STEP2 で出力される各修正内容は，下記の情報を持つ．

- 修正前コード片において削除された字句列および文列
- 修正後コード片において追加された字句列および文列
- その修正が行われた日時
- その修正が行われたファイル名 (ファイルパス)

$S(c_k)$ から抽出された修正内容の集合を $M(c_k)$ と表す．STEP2 の出力は，$M(c_k)(\forall c_k \in C(R))$ である．

STEP3: 修正パターンを導出

STEP2 の出力である $M(c_k)(\forall c_k \in C(R))$ から，修正パターンを導出する．

$M(c_k)$ に含まれる或る修正内容 m に着目する．m の修正前字句列および修正後字句列をそれぞれ m_{before} および m_{after} とする．STEP2 で得られた全修正内容の中で，修正前字句列が m_{before} である修正内容の集合を $M(m_{before})$ とする．また，修正前字句列が m_{before} でありかつ修正後字句列が m_{after} である修正内容の集合を $M(m_{before},m_{after})$ とする．修正パターンの各メトリクスは下記のように求めることができる．

- 支持度：$M(m_{before},m_{after})$ に含まれる要素数．
- 修正後文数：m_{after} に含まれる文の数．
- 出現期間：$M(m_{before},m_{after})$ に含まれる修正内容において，最も新しい修正内容の日時と最も古い修正の日時の差．
- ファイル数：$M(m_{before},m_{after})$ に含まれる修正内容のユニークなファイル名の数．
- 修正種別：m_{before} および m_{after} がどちらも空でなければ変更．m_{before} が空の場合は追加，m_{after} が空の場合は削除．
- 修正規模：m_{before} と m_{after} の字句列を最長共通部分列検出アルゴリズム [1] を用いて比較する．このときに全ての識別子名およびリテラルは特殊字句に置換されている．比較の結果，差がなかった場合は字句単位，差があった場合は文単位となる．

STEP2 で得られた全ての修正内容に対して，上記の処理を行い，修正パターンを導出する．

表 1　ArgoUML の概要

開始リビジョン (日時)	1 (1998-01-27 07:19:17)
最終リビジョン (日時)	19,910 (2013-02-08 12:31:06)
対象リビジョン数	19,910
最終リビジョンの行数	370,152

4. 評　　価

提案手法をツールとして実装し(注1)，実験を行った．この実験の目的は，提案手法を利用することでリファクタリングの対象として適切な重複コードが提示されるかを調査することである．

実験対象はオープンソースソフトウェアの ArgoUML である．表 1 に示すように，ArgoUML は約 15 年間開発されているソフトウェアであり，約 20,000 のリビジョンが存在している．最新リビジョンのサイズは約 37 万行である．

提案手法を用いて ArgoUML のリポジトリから修正パターンを導出した．その結果，64,124 の修正パターンを得た．次に，下記のしきい値を用いて修正パターンの絞り込みを行った．

- **条件 1:** 支持度の下限は 3
- **条件 2:** 修正後文数の下限は 5
- **条件 3:** 修正種別は変更のみ
- **条件 4:** 修正規模は文単位のみ

上記の条件を用いた理由は，重複コードを除去する動機として下記の状況が成り立つ場合が多いと著者らが考えたためである．

- 同様の修正が多い回数行われた重複コードは，少ない回数行われた重複コードに比べて，除去の優先度が高い (条件 1).
- 大きい重複コードは，小さい重複コードに比べて，除去の優先度が高い (条件 2).
- 変更が行われた重複コードは，追加された重複コードに比べて，除去の優先度が高い (条件 3).
- 大きな変更が加わった重複コードは，小さな変更が加わった重複コードに比べて，除去の優先度が高い (条件 4).

その結果，13 個の修正パターンを抽出した(注2)．各修正パターンについて以下の調査を行った．

- 修正前後のリビジョンのコードの閲覧
- 最新リビジョンにおいて対応するコードの閲覧

この調査を行った結果，特定された 13 個の修正パターンを表 2 に示すように分類できた．カテゴリ **A** は，最新リビジョンにおいてすでにリファクタリングされていた候補を表す．六つの修正パターンがカテゴリ **A** に分類された．そのうちの四つが

表 2　調 査 結 果

	カテゴリ	修正パターン数
A:	既にリファクタリング済み	6
	A1: メソッドの引き上げ	4
	A2: メソッドの抽出	2
B:	リファクタリングの必要なし	4
	B1: 自動生成コード	2
	B2: 典型的な処理	1
	B3: 最終リビジョンで存在しない	1
C:	リファクタリングする価値がある	3

(注1)：https://github.com/YoshikiHigo/MPAnalyzer

(注2)：今回の実験では，抽出した修正パターン全てを手作業により確認するために，比較的厳しめの条件を用いた．より緩い条件を用いることにより，抽出される修正パターンの数は増える．

```
 41 public class ActionStateDiagram extends ActionAddDiagram {
      ...
 82   public boolean isValidNamespace(MNamespace ns) {
 83     if (ns instanceof MClassifier) return true;          DELETED
 84       return false;
 85   }
      ...
```

(a) 修正前のソースコード (リビジョン 3,760)

```
 54 public class ActionStateDiagram extends ActionAddDiagram {
      ...
108   public boolean isValidNamespace(Object handle) {
109     if (!ModelFacade.isANamespace(handle)) {
110       cat.error("No namespace as argument");
111       cat.error(handle);
112       throw new IllegalArgumentException(
113       "The argument " + handle + "is not a namespace.");
114     }
115     MNamespace ns = (MNamespace)handle;
116     if (ns instanceof MClassifier)
117       return true;                                       ADDED
118     return false;
119   }
      ...
```

(b) 修正後のソースコード (リビジョン 3,761)

図 4　提案手法により特定されたリファクタリングの候補．この候補は最新リビジョンにおいて，共通の親クラスに重複コードを引き上げることにより実際にリファクタリングされていた．

修正パターンのコードを共通の親クラスに引き上げることにより，重複コードを除去していた．

図 4 はその一例を示す．図 4(a) は修正前のコードを表しており，リビジョン 3,761 への更新の際に DELETED の部分が削除され，図 4(b) の ADDED の部分のコードが追加されている．この修正は全部で四つのクラス，ActionStateDiagram，ActionSequenceDiagram，ActionCollaborationDiagram，ActionDeploymentDiagram の中の isValidNamespace(Object) という共通のシグネチャを持つメソッドの中で行われていた．そして，最新リビジョンでは，これら四つのクラスは ActionNewDiagram という共通のクラスを継承しており，そこにこのメソッドが引き上げられていた．

また，二つの修正パターンは，重複コードを新しいメソッドやコンストラクタとして抽出することにより除去していた．図 5 はその例を示す．この例では，変数 persister が指すオブジェクトを生成する処理が追加されている．どの型のオブジェクトが生成されるかは変数 projectMember が指すオブジェクトの型に依存している．最新リビジョンでは，この追加された if-else 文が新しく作成されたコンストラクタ呼び出しに変更されていた．この修正パターンは全部で三箇所に現れており，最新リビジョンではすべて同様のリファクタリングが行われていた．

カテゴリ **B** は，リファクタリング候補としては適切ではないと判断された修正パターンである．自動生成部分から検出された修正パターンが二つあった．これは，自動生成されたコードを人間が手で修正したのではなく，コンパイラコンパイラに入力として与える構文ファイルが変更されたために，自動生成コードが変化していた．自動生成コードそのものがリポジトリで管理されていたため，このような例を見つけてしまった．ソフトウェアの開発者であればどのファイルが自動生成かは認識しているはずなので，このようなファイルを除外して修正パターンを特定することは難しくない．

また，Java 言語で記述されたソフトウェアにおいて頻出する

```
 45 public class XmiFilePersister extends AbstractFilePersister {
      ...
 77   public void doSave(Project project, File file)
 78       throws SaveException {
        ...
107     if (projectMember.getType().equalsIgnoreCase("xmi")) {
108       if (LOG.isInfoEnabled()) {
109         LOG.info("Saving member of type: "
110               + ((ProjectMember) project.getMembers()
111                     .get(i)).getType());
112       }
113       projectMember.save(writer, null);                  DELETED
144     }
        ...
```

(a) 修正前のソースコード (リビジョン 7,436)

```
 47 public class XmiFilePersister extends AbstractFilePersister {
      ...
 79   public void doSave(Project project, File file)
 80       throws SaveException {
      ...
109     if (projectMember.getType().equalsIgnoreCase("xmi")) {
110       if (LOG.isInfoEnabled()) {
111         LOG.info("Saving member of type: "
112               + ((ProjectMember) project.getMembers()
113                     .get(i)).getType());
114       }
115       MemberFilePersister persister = null;
116       if (projectMember instanceof ProjectMemberDiagram) {
117         persister = new DiagramMemberFilePersister();
118       } else if (projectMember instanceof ProjectMemberTodoList) {
119         persister = new TodoListMemberFilePersister();
120       } else if (projectMember instanceof ProjectMemberModel) {
121         persister = new ModelMemberFilePersister();
122       }
123       persister.save(projectMember, writer, null);       ADDED
124     }
        ...
```

(b) 修正後のソースコード (リビジョン 7,437)

```
208     if (projectMember.getType().equalsIgnoreCase(getExtension())) {
209       if (LOG.isLoggable(Level.INFO)) {
210         LOG.log(Level.INFO, "Saving member of type: {0}",
211                 projectMember.getType());
212       }
213       MemberFilePersister persister = new ModelMemberFilePersister();
214       persister.save(projectMember, stream);
215     }                                                    DELEGATED
```

(c) 最新リビジョンのソースコード (リビジョン 19,910)

図 5　提案手法により特定されたリファクタリングの候補．この候補は最新リビジョンにおいて，重複コードを新しいコンストラクタとしてくくり出すことにより実際にリファクタリングされていた．

と思われる修正パターンが一つ見つかった．そのコードを図 6 に表す．この修正は Enumeration を使って行なっていた繰り返し処理を for 文に変更している．Java には他にも繰り返し処理を行うためのメカニズムとして Iterator や for 文が用意されており，それらの間における処理の変更はしばしば行われる．この修正パターンは Java 言語特有のものであり，ソフトウェア特有のものではないため，一つにまとめることが必ずしも良いとはいえないと著者らは考えてカテゴリ **B** に分類した．

また，最新リビジョンではファイルそのものがなくなっていた修正パターンが一つ存在した．

最新リビジョンで修正後のコードが残っており，著者らがリファクタリング候補として適切であると判断した修正パターンは三つあった．これらは，カテゴリ **A** と同様に，重複コードを共通の親クラスに引き上げたり新しいメソッドとしてくくり出したりすることで簡単にリファクタリング可能である．

5. 関 連 研 究

5.1 コード変更の抽出

Kim らはコード変更内容を要約する手法を提案し，提案を元に LSDiff というツールを実装している [6]．彼らの手法を用い

```
222 /** Reply the bounding box for this FigEdge. */
223 public Rectangle getBounds() {
224   Rectangle res = _fig.getBounds();
225   Enumeration enum = _pathItems.elements();
226   while (enum.hasMoreElements()) {
227     Fig f = ((PathItem) enum.nextElement()).getFig();
228     res = res.union(f.getBounds());
229   }
230   return res;
231 }
```

DELETED

(a) 修正前のソースコード (リビジョン 145)

```
240 /** Reply the bounding box for this FigEdge. */
241 public Rectangle getBounds() {
242   Rectangle res = _fig.getBounds();
243   int size = _pathItems.size();
244   for (int i = 0; i < size; i++) {
245     Fig f = ((PathItem) _pathItems.elementAt(i)).getFig();
246     res.add(f.getBounds());
247   }
248   return res;
249 }
```

ADDED

(b) 修正後のソースコード (リビジョン 146)

図 6　提案手法により特定されたリファクタリングの候補．この修正は Java においてよく行われる変更であると著者らは考えたため，リファクタリングの候補として適切ではないと判断した．

ることにより，API に対してどのような変更があったのか，メソッド呼び出し等のプログラム要素にどのような変更があったのかを容易に把握できる．彼らの手法の目的は理解支援であり，人間が理解しやすいレベルにまでコード変更の内容を抽象化する．それに対して，本研究の手法は，コード変更の抽出はリファクタリング候補を発見を目的としており，その変更内容を抽象化はしていない．提案手法は，リファクタリング候補となり得ない変更を取り除くために，ソースコードのテキストではなく，字句列を比較するという方法をとっている．

Fluri らは細粒度のコード変更内容抽出手法を提案している [2]．彼らの手法は，二つのバージョンのソースコードから生成した抽象構文木を比較する．この比較により，一方の木を他方の木に変化させるために必要な操作の列を得る．抽象構文木を比較することで行単位の比較では特定できない変更 (例えば引数の順番の入れ替え) を抽出できる．本研究では，非常に多くのリビジョンを解析してリファクタリングの候補を特定する必要があることから，抽象構文木を必要としない方法を用いたが，抽象構文木を用いた修正パターンの導出もリファクタリング候補の検出には有用であると著者らは考えている．

5.2　リファクタリング候補の発見

著者らはこれまでにクラス，メソッド，ブロック単位で重複コードを検出し，それらをリファクタリング候補として提示する方法を提案している [4]．この手法では，各候補はメトリクスを用いて特徴付けがされているため，リファクタリング方法も含めて支援している．たとえば，重複コードを所有するクラスは共通の親クラスを持つか，重複コードは新しいメソッドとしてくくり出すことが容易か，等が簡単に判別できる．

また，堀田らはプログラム依存グラフを用いて重複コードを検出し，それらを Form Template Method を用いて除去する手法を提案している [11]．重複コードの検出にプログラム依存グラフを用いているため，著者らの過去の手法とは違い，必ずしもメソッド全体やブロック全体が重複していなくても除去可能な重複コードを検出できる．

著者らの過去の手法や堀田らの手法は，一つのバージョンのソースコードを解析することによってリファクタリング可能な重複コードを特定する手法である．それに対して，本研究の手法は，過去に実際に同様の修正が行われた重複コードをリファクタリング候補として提示する．

6. おわりに

本稿では，過去に同様の修正が行われた重複コードをリファクタリング候補として提示する手法を提案した．既存手法とは異なり，過去に同様の修正を必要した重複コードをメソッドの抽出やメソッドの引き上げ等の候補として提示するという点で既存研究とは大きく異なる．提案手法をオープンソースソフトウェアに対して適用し，特定した 13 の候補のうち，リファクタリング候補として適切でなかったものは四つのみであった．今後は，実際に修正が行われたコードだけではなく，その周辺のコードの情報も用いてより適切なリファクタリング支援を行えるように手法を拡張していく予定である．

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)，挑戦的萌芽研究 (課題番号 : 24650011)，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) の助成を得た．

文　献

[1] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Processing Information Retrieval*, pages 39–48, 2000.

[2] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[3] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[4] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evlution: Research and Practice*, 20(6):435–461, 2008.

[5] C. J. Kapser and M. W. Godfrey. ”cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, 2008.

[6] M. Kim, D. Notkin, D. Grossman, and G. W. Jr. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39:45–62, 2013.

[7] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering*, pages 187–196, 2005.

[8] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools : A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[9] 肥後芳樹, 吉田則裕. コードクローンを対象としたリファクタリング. **コンピュータソフトウェア**, 28(4):43–56, 2011.

[10] 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**, 54(5):1686–1696, 2013.

[11] 堀田圭祐, 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローンに対するテンプレートメソッドパターン適用支援手法. **電子情報通信学会論文誌** *D*, 95(7):1439–1453, 2012.