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.

修正の分類に基づいたコミット分割手法の提案

楠　　野明†　堀田　圭佑†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科，吹田市
E-mail: †{k-noa,k-hotta,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェアリポジトリマイニングが広く注目されている．これはソフトウェアリポジトリから将来の開発及び保守作業に有用な知見を得ようとする研究である．ソフトウェアリポジトリマイニングにおいて，ビッグコミットを解析対象から除外する処理が行われる．ビッグコミットとは大規模な修正をリポジトリに反映するコミットである．ビッグコミットはフォーマットの修正を多く含むことが指摘されており，そのため分析対象から除外される．しかし，ビッグコミットはフォーマットの修正と同時に分析すべき情報も含む場合があるため，ビッグコミットの除外によってそれらが失われている．よってリポジトリマイニングの結果が正確でなくなっている可能性がある．この問題を解決するため，本研究ではコミットに含まれる修正を分類し，それに基づいてコミットを分割する手法を提案する．これによって，コミットに含まれる分析すべき修正のみを対象とした解析が可能となる．

キーワード　ソフトウェアリポジトリマイニング，版管理システム，コミット，最長共通部分列

Dividing commits based on classification of modifications

Noa KUSUNOKI†, Keisuke HOTTA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

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

Abstract　Many researchers have mined software repositories to gain knowledge or principles that can encourage efficient software development. Historical code repositories are one of the well-mined repositories. In mining historical code repositories, researchers often omit big commits from their mining targets. A big commit indicates a commit that modifies many source files or many lines of code. The reason of the preprocessing is that it is said that most of modifications included in big commits were trivial ones such as re-formatting of source code. However, nobody can say that all of such modifications are trivial without any exception. In other words, big commits can include non-trivial modifications on code. Hence, omitting big commits should reduce the accuracy of the mining. This paper proposes a method that divides a commit into multiple commits based on types of modifications. The proposed method enables us to retrieve valuable information that was discarded by ommiting big commits.

Key words　software repository mining, version control system, commit, longest common subsequence

1. まえがき

近年，ソフトウェアリポジトリマイニングに関する研究が注目されている [1]〜[5]．ソフトウェアリポジトリとはソフトウェア開発にかかわる様々な情報を蓄積したものである．ソフトウェアリポジトリマイニングとは，このソフトウェアリポジトリを分析し，ソフトウェアを開発するうえで有益な知見を得ることを目的とした研究分野である．ソフトウェアリポジトリには，過去に行われた修正内容や，不具合の記録，開発中にやり取りされた電子メールなど様々な情報が蓄積されている [6], [7]．ソフトウェアリポジトリマイニングでは，前処理としてビッグコミットを対象から除外することが多い．ビッグコミットとはコミット(注1)のうち修正が大規模(注2)なものを指す．その理由として，ビッグコミットにはソフトウェアの振る舞いに影響を与える修正が少ない，という点がある [8]．また，一度のコミットで加えられる修正は規模が小さいものがほとんどである [9]．そのため，ビッグコミットを分析対象とすると，本来分析したいソフトウェアの振る舞いに影響する修正が，分析結果に及ぼす影響が現れにくくなってしまう．よって，ビッグコミットを分析対象から除外するという前処理がしばしば行われている．し

(注1)：コミットとは修正をリポジトリに反映する操作
(注2)：ここでいう大規模とは一度に修正が加わったファイル数や，行数が多いことを意味する．

行番号	修正前のソースコード
1	package net.sourceforge.squirrel_sql.fw.util;
2	/*
…	…
35	private interface ActionProperties
36	{
37	String DISABLED_IMAGE = "disabledimage";
38	String IMAGE = "image";
39	String NAME = "name";
40	String ROLLOVER_IMAGE = "rolloverimage";
41	String TOOLTIP = "tooltip";
42	}
…	…
302	}
303	else
304	{
305	s_log.debug("No resource found for " + keyName + " : "
306	+ propName);
307	}
…	…
431	}
432	

行番号	修正後のソースコード
1	package net.sourceforge.squirrel_sql.fw.util;
2	
3	/*
…	…
36	private interface ActionProperties
37	{
38	String DISABLED_IMAGE = "disabledimage";
39	
40	String IMAGE = "image";
41	
42	String NAME = "name";
43	
44	String ROLLOVER_IMAGE = "rolloverimage";
45	
46	String TOOLTIP = "tooltip";
47	}
…	…
301	}else
302	{
303	if (s_log.isDebugEnabled())
304	{
305	s_log.debug("No resource found for " + keyName + " : " + propName);
306	}
307	}
…	…
425	}
426	

図 1　複数種類の修正が行われる例

かし，ビッグコミットの中にソフトウェアの振る舞いに影響を与える修正が含まれていた場合，このような前処理によって本来は分析すべき修正も除外してしまう．

本研究ではこの問題を解決するために，ソースコードに加えられた修正を分類し，その分類に基づいてコミットを分割する手法を提案する．コミット自体を分割する方法はこれまでに提案されていない．提案手法では，ソースコードに加えられた修正を以下の三種類に分類する．

- プログラムコードの修正
- フォーマットの修正
- コメントの修正

提案手法をツールとして実装し，評価実験を行った．複数のソフトウェアに対して実験を行った結果，その開発者自身による修正の分割を正解としたとき，提案手法による修正の分割の適合率は約 67%となり，再現率は約 72%という結果になった．

2. 研 究 動 機

図 1 は，オープンソースソフトウェアの "SQuirreL SQL Client" のリポジトリに対して行われたあるコミットにおける修正を示している．このコミットでは 100 行程度にフォーマットの修正が行われ，303 行目，304 行目および 306 行目では文の追加が行われている．図 1 のような修正が，ビッグコミットを除外する前処理によって除外された場合，303 行目，304 行目，306 行目の文の追加が見落とされるという問題が起こる．上記の問題は，一つのコミットに存在するフォーマットの修正と数行の文の追加を分離し，二つに分割された修正を別々にコミットすることによって解決できる．そこで本研究では一つのコミットに含まれる複数種類の修正を別々のコミットに分割する手法を提案する．

3. 提 案 手 法

3.1 概　　要

提案手法はリポジトリを入力とし，入力リポジトリに含まれるコミットを分割した後のコミットを出力する．コミットは 4. にて説明する手順で分割される．

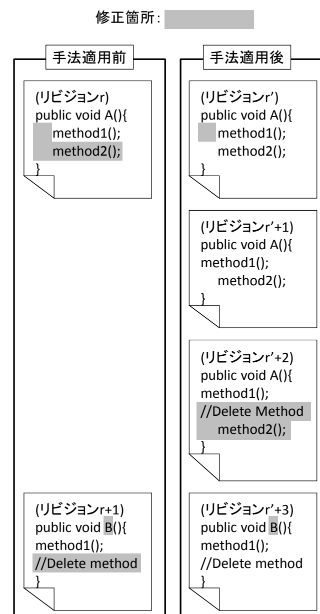


図 2　提案手法適用による修正分類の例

実際に提案手法を適用した際に，修正の加わったソースファイルにおいて，行われていた修正がどのように分割されるかという例を図 2 に示す．図 2 では提案手法適用前のリビジョン r のソースファイルとリビジョン $r+1$ のソースファイル間で，以下の四つの修正が行われている．

- インデントの削除
- コメントの追加
- 文の追加
- 字句単位の変更

提案手法を適用した結果，ソースファイルにおいてはリビジョン r' から $r'+3$ の間で分割された修正が別々のコミットで行われた修正として保管される．

3.2 分類の定義

コミットを分割する際に行われる修正の分類では，コミットに含まれる修正を以下に定義する四種類に分類する．

修正 A フォーマットの修正

修正 B コメントの修正

修正 C プログラムコードの修正，およびソースファイルの追加・削除

修正 D ソースファイル以外のファイルの修正，およびソースファイル以外のファイルの追加・削除

ただし，修正 A におけるフォーマットの修正とは，空白，タブ，および改行文字の修正のことである．

3.3 用語の定義

本節では以降の説明で必要となる諸用語の定義を行う．

［**Definition** 3.1］（列） n 以下のすべての自然数からなる集合を I とする．このとき，ある集合 X に対して I を定義域とする写像 $\phi : I \to X$ を列と呼び，$\phi = \langle x_1, x_2, \ldots, x_n \rangle (\forall i \in 1..n[i \in I \land x_i \in X])$ と表記する．また ϕ の逆像を f_ϕ とする．例えば $\phi(i) = x_i$ であるとき，$f_\phi(x_i) = i$ となる．以降，ϕ の定義域を I_ϕ，値域である集合 X を，V_ϕ とする．

［**Definition** 3.2］（部分列） $\phi : I \to X$ と表記される列 ϕ に対して，I の部分集合 $I'(I' \subset I)$ を定義域とする写像 ϕ' を部分列と呼ぶ．

［**Definition** 3.3］（共通部分列） 二つの列 A, B を考える．このとき以下の条件を満たす部分列 A'，B' を共通部分列と呼ぶ．

- A' は A の部分列である．
- B' は B の部分列である．
- $|I_{A'}| = |I_{B'}|$ となる．
- $|I_{A'}| = |I_{B'}| = m$ として $\forall i \in 1..m[a_i = b_i]$ となる．

［**Definition** 3.4］（最長共通部分列） 二つの列 A, B に対して複数の共通部分列が存在するとき，その中で要素数が最大のものを最長共通部分列と呼ぶ．

［**Definition** 3.5］（字句） 字句は，字句を表す文字列と字句の種類の情報を持つ．K を字句の種類の集合としたとき，字句 t は $t = (s, k)$ と表記される．ここで，s は文字列を表し，k は字句の種類を表す $(k \in K)$．また，写像 g_s, g_k をそれぞれ $g_s(t) = s$, $g_k(t) = k$ と定義する．さらに，字句 t_1, t_2 において，$g_s(t1) = g_s(t2)$ となるとき $t_1 = t_2$ とみなす．

［**Definition** 3.6］（字句の種類） K には，一般的な字句解析器で定義される字句の種類に加えて，コメント，空白，タブ，および改行文字も字句の種類として含まれる．また，K の部分集合として以下を定義する．

- $W = \{$ 空白，タブ，改行 $\}$
- $C = \{$ コメント $\}$
- $P = K - (W \cup C)$

［**Definition** 3.7］（字句列） V_ϕ が字句の集合であるような列 ϕ を，字句列とする．

4. 修正の分割手順

提案手法における，コミットに含まれる修正の分割手順は以下の四つになる．

ステップ 1：字句列の取得

ステップ 2：修正前後における字句の対応の特定

ステップ 3：字句に行われた修正の分類

ステップ 4：コミットの分割

以降では各ステップで行われる処理を説明する．

ステップ 1：字句列の取得

ファイル F があるコミットで修正されていた場合，その修正前のリビジョンを F_{before}，修正後のリビジョンを F_{after} とする．これらを入力として字句解析を行い，字句列 $T_{before} = \langle x_1, x_2, \ldots, x_l \rangle$，および $T_{after} = \langle y_1, y_2, \ldots, y_m \rangle$ を取得する．

ステップ 2：修正前後における字句の対応関係の特定

ステップ 1 で取得した字句列 T_{before}, T_{after} について字句列に含まれる字句の対応関係を特定する．対応関係は以下の二種類存在する．

文字列対応関係 字句を表す文字列の一致に基づく字句同士の対応関係

種類対応関係 字句の種類の一致に基づく字句同士の対応関係

それぞれ以下のように求める．

文字列対応関係の取得方法

まず T_{before}，T_{after} の最長共通部分列を求める．その際，最長共通部分列が複数存在した場合はいずれか一つを用いる．その最長共通部分列を T'_{before}, T'_{after} としたとき，T_{before}の要素 x_iと T_{after}の要素 y_j が以下のすべての条件を満たす時，x_i と y_j は文字列対応関係を持つといい，$x_i \Leftrightarrow y_j$ と表す．

- $x_i \in V_{T'_{before}}$
- $y_j \in V_{T'_{after}}$
- $f_{T'_{before}}(x_i) = f_{T'_{after}}(y_j)$

種類対応関係の取得方法

T_{before}の要素 x_i, $x_u(x_i,\ x_u \in V_{T'_{before}} \land f_{T'_{before}}(x_i) = w \quad \land f_{T'_{before}}(x_u) = w + 1)$ と T_{after}の要素 y_j, $y_v(y_j, y_v \in V_{T'_{after}} \land f_{T'_{after}}(y_j) = w \quad \land f_{T'_{after}}(y_v) = w + 1)$ の間で，$x_i \Leftrightarrow y_j, x_u \Leftrightarrow y_v$ であると仮定する．また，T_{before} に関して，$\{i+1, \ldots, u-1\}$ を定義域とする T_{before}の部分列を X，T_{after} に関して，$\{j+1, \ldots, v-1\}$ を定義域とする T_{after}の部分列を Y とする．このとき，(X, Y) の対を一致間部分列対と呼ぶ．この一致間部分列対を S として，一致間部分列対 S はそれを構成要素である部分列 X, Y のうちいずれかの要素数は 0 になりえるが，双方の要素数が 0 にはなりえない．T_{before}, T_{after} の間で取得できるすべての一致間部分列対に対して，Algorithm1 を実行することで，種類対応関係を取る．Algorithm1 は，部分列 X, Y を入力として，種類対応関係を持つ字句対の列を出力する．上記の処理によって，T_{before} の要素 x_sと T_{after}の要素 y_t の間に種類対応関係があるとき，$x_s \leftrightarrow y_t$ と表す．

ステップ 3：字句に行われた修正の分類

ステップ 2 の二つの対応関係を利用することで，T_{before}, T_{after} において行われた修正を字句単位で特定できる．各字句の修正が追加，削除，変更のいずれであるかは以下の条件で決定する．

Algorithm 1 種類対応関係の特定

Input: X, Y
Output: TP

```
pairposition ← 1
for all beforeToken ∈ V_X do
  i ← pairposition
  for i to |I_Y| do
    afterToken ← Y(i)
    if g_v(beforeToken) == g_v(afterToken) then
      TP ← TP + ⟨beforeToken, afterToken⟩
      pairposition ← i + 1
      break
    end if
  end for
end for
```

- y_jが追加 ($y_j \in V_{T_{after}}$)：

$$\forall x_i \in V_{T_{before}}[x_i \not\Leftrightarrow y_j \wedge x_i \not\leftrightarrow y_j] \tag{1}$$

- x_iが削除 ($x_i \in V_{T_{before}}$)：

$$\forall y_j \in V_{T_{after}}[x_i \not\Leftrightarrow y_j \wedge x_i \not\leftrightarrow y_j] \tag{2}$$

- x_iが変更 ($x_i \in V_{T_{before}}$)：

$$\exists y_j \in V_{T_{after}}[x_i \leftrightarrow y_j] \tag{3}$$

上記の判定の結果，追加，削除，および修正のいずれかが字句 t に対して行われている時，字句 t が修正された，という．このステップでは字句に対して行われた修正を修正 A，修正 B，修正 C の三つに分類する．以下にある字句 t に対する修正を分類する条件を示す．

修正 A　以下の二つの条件をすべて満たす
- $g_v(t) \in W$
- t が属する一致間部分列対 $S(S = (X, Y))$ について，$V_S = V_X \cup V_Y$ としたとき，$\forall c \in V_S[g_v(c) \notin P]$ が満たされる

修正 B　$g_v(t) \in C$

修正 C　以下二つの条件のいずれかを満たす
- $g_v(t) \in P$
- t が属する一致間部分列対 S($S = (X, Y)$) について，$V_S = V_X \cup V_Y$ としたとき，$\exists c \in V_S[g_v(c) \in P] \wedge g_v(t) \in W$

ステップ 4：コミットの分割

ステップ 3 で行った修正の分類に基づいてコミットの分割を行う．ソースファイルに対する修正は修正 A，B，C の三種類にまとめられ，分割によって一つのコミットが最大三つのコミットに分割される．

5. 実　　験

本研究では提案手法を実装したツールを用いて，修正の分類の正しさを確認する評価実験を行った．実験の対象は著者らの研究室で開発されたいくつかのソフトウェアであり，実験は対象ソフトウェアの開発者の協力のもと行った．

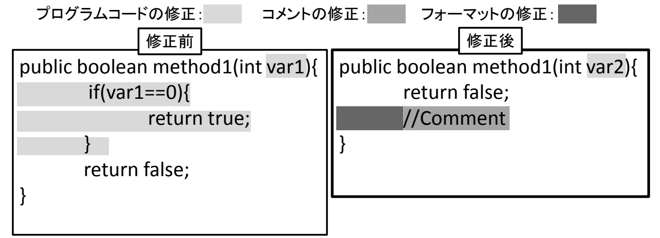


図 3　実験における修正箇所の色分け例

5.1 実験方法

著者らは実験の準備を以下の手順で行った．

手順 1　実験対象リポジトリにツールを適用する．

手順 2　入力として与えられたリポジトリから，一度のコミットで二種類以上の修正が同時に行われたソースファイルを実験データとして抽出する．

手順 3　実験データとするファイルに加えられた修正を，提案手法が特定した分類に従い色分けする．修正箇所の色分けの例を図 3 に示す．

手順 1-3 で準備した実験データの各修正箇所に対して，被験者に以下の二つのことを確認してもらった．
- 修正の範囲は正しいか
- 修正の分類は正しいか

この二つがすべて正しいとき，提案手法による分割が正しいと判断した．また，正しくないと判断されたものに関しては，被験者に正しい範囲と分類を指摘してもらった．

5.2 評価項目

実験のデータ，および結果を以下の項目で評価した．

出力数　提案手法が特定した各種類の修正箇所の集合を H として，その要素数

出力正解数　H の内，分割が正しいとされた修正箇所の集合を I として，その要素数

正解数　被験者に修正された修正箇所の修正後の集合を J として，$I \cup J$ の要素数

適合率，再現率

$$\frac{|I|}{|H|} * 100, \qquad \frac{|I|}{|I \cup J|} * 100$$

5.3 実験結果

各被験者は 5 つの実験データに対して実験を行った．結果は表 1 のようになった．表の行は各被験者を表し，列は評価項目を表している．表 1 における総合の項目の適合率は約 67%となり，再現率は約 72%となった．

表 1　実験結果

被験者	正解数	出力数	出力正解数	適合率	再現率
1	177	354	141	39.83	79.66
2	36	37	32	86.49	88.89
3	115	117	112	95.73	97.39
4	490	375	307	81.87	62.65
総合	818	883	592	67.04	72.37

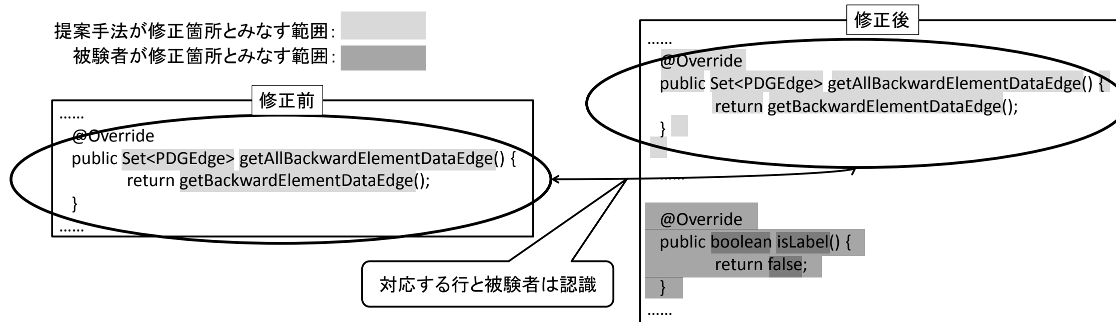


図 4　分類が誤りとされた例

6. 議　　論

6.1　分類がうまくいかなかった例

実験において行った分類の内，分類がうまく行えなかった例について述べる．

図 5.3 のような分類が，被験者によって誤りとされることが多かった．図における強調箇所は修正箇所であり，提案手法が修正箇所とみなした箇所と，被験者が認識した修正箇所を重ねて示している．最長共通部分列を用いた方法では修正されたとみなされる字句数が最小となるように修正箇所を求めるが，提案手法が特定した修正箇所における修正字句数は 23 であるのに対して，被験者の認識した修正箇所における修正字句数は 24 であり，提案手法が特定した修正箇所における修正字句数の方が少ない．これにより提案手法が特定した修正箇所と，被験者が認識した修正箇所にずれが生じたと考えられる．

6.2　被験者による判断基準の違い

5. における表 1 の値に関して考察する．被験者 4 に対する実験の結果，正解数が 490 であるのに対して，出力数は 375 と大きな差が生じた．これは図 5(c) のような例が原因である．複数の字句がまとまって追加されたという修正に対して，提案手法はそれらを一つのまとまったプログラムコードの修正と判定している．それに対して被験者は追加された字句の内，空白，タブ，改行文字の追加をフォーマットの修正と判定している．これにより正解数と出力数に大きな差が生じた．また，被験者 1 に対する実験結果においても，正解数，出力数に大きな差が生じた．これは図 5(a) のよう例が原因で，提案手法が複数の細かい修正があると判定した箇所を，被験者は一つのまとまった修正であると判定したためである．このような提案手法による判定の基準と被験者の判定基準の違いが，被験者 1 および被験者 4 の適合率及び再現率の低下に大きな影響を与えている．しかしながら，図 5(b) のように，被験者 3 は提案手法が特定する修正箇所を正しいと判定している．すなわち被験者 3 は提案手法に近い判定基準で修正箇所の判定を行っていたといえる．したがって，表 1 における被験者 3 の結果をみると，適合率，再現率の双方ともに他の被験者と比較して高い値となっている．それに対して，図 5(c) のように修正箇所を判定した被験者 4 は，表 1 においても適合率と再現率の値は被験者 3 に比べて低くなっている．このように被験者ごとに判定基準が異なるため，より精度の高い判定を行うには被験者ごとの判定基準を加味する必要がある．

7. 関 連 研 究

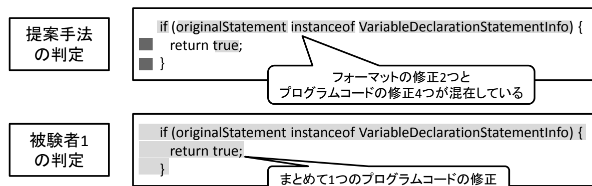
Fluri らは，ソースコードに加えられた修正をその意味に応じて自動的に分類する手法を提案している [10]．この手法はソースコードから抽象構文木を構築し，それらを比較することで，修正が行われた箇所を特定する．また，それぞれの修正内容に応じて，特定した修正を文献 [11] で述べられている定義に従って分類している．

Kawrykow と Robillard はソースコードに加えられた修正の内，リポジトリマイニングを行う上で有用ではないものを特定する手法を提案している [12]．彼らの手法では，ソースコードに加えられたプログラムコードの修正の内，変数名の変更などに伴う修正を “表面的な” 変更とし，これらをリポジトリマイニングの対象に加えることは有用ではないとしている．

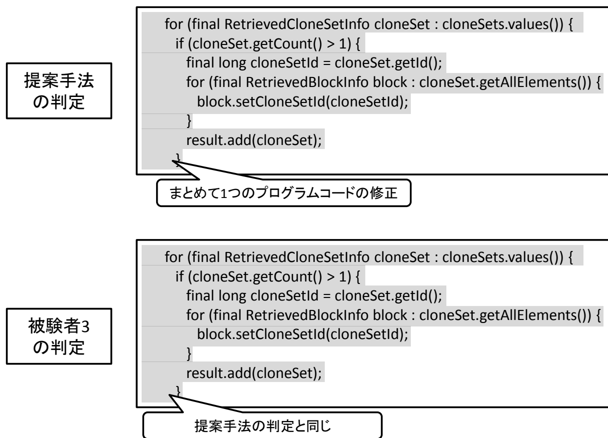
これらの手法は提案手法と同様に，ソースコードに加えられた修正を分類している．しかし，提案手法はコミットを分割することでフォーマットの修正やコメントの修正を取り除き，プログラムコードの修正のみを抽出できる点で，既存の修正を分類する手法とは異なる．

Hayashi らはソースコードの編集履歴のリファクタリング手法を提案し，その自動化ツールを統合開発環境 Eclipse のプラグインとして開発している [13]．Hayashi らの手法は開発者が行った編集操作を自動的に記録し，その操作に対して順序の入れ替えや，統合，取り消しを可能にしている．異なる目的を持つ編集操作が混在した履歴についても，編集操作の入れ替えや統合によってそれぞれの目的ごとに編集操作をまとめることで，同じ目的で行われた編集操作を一つの修正としてまとめてリポジトリにコミットすることを可能にしている．Hayashi らの手法がソースコードに対する編集履歴の再構築を行うのに対し，提案手法は版管理システムに変更を反映させて得られた開発履歴の再構築を行っているという点で異なる．

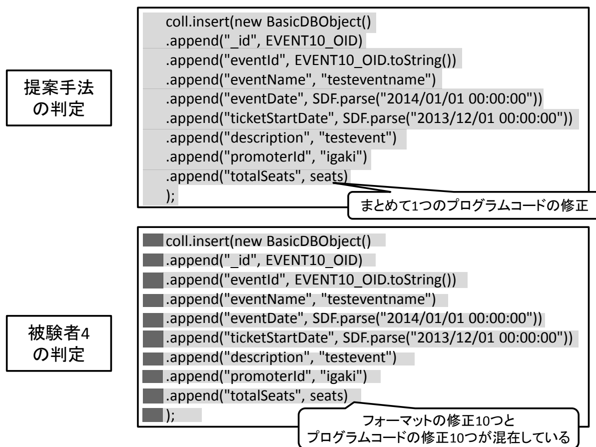
開発履歴に出現する大規模な修正に関する調査もいくつか行われている．Hattori らが行った調査によると，修正は五個以下のファイルに対して変更を加えるものがほとんどであるが，一部 100 を超えるファイルに対して変更が加えられるコミットが存在することを指摘している [9]．Hindle らの報告によれば，



(a) 被験者 1



(b) 被験者 3



(c) 被験者 4

図 5　被験者による判定基準例

ビッグコミットが起こる要因として，本流から枝分かれして開発されていたソースコードを本流へ合流させること (merge) や，フォーマット変更や不要になったコードの削除などを挙げている [14].

8. あとがき

本研究では，コミットに含まれる修正を分類し，その分類に基づいて，コミットを分割する手法を提案した．

また二つの評価実験を行い，分類の正しさを評価する実験では，適合率は約 67%，再現率は約 72%となった．

今後の課題は以下の二点である．

- 修正の分類方法を改善することで，より開発者の認識に近い分類を行うようにする．
- 実際のリポジトリマイニング手法を，提案手法の適用前後のリポジトリに対して適用し，その結果の変化を確認する．

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] 松下　誠，“ソフトウェア工学の新潮流 (1) リポジトリマイニング,” ソフトウェアエンジニアリング最前線 2009，pp.21–24, Sep. 2009.

[2] 林晋平，佐伯元司，“リファクタリング支援に用いる知識抽出のためのソフトウェアリポジトリの解析,” 電子情報通信学会技術研究報告，vol.106，no.16，pp.1–6，2006.

[3] M. Askari and R. Holt, “Information Theoretic Evaluation of Change Prediction Models for Large-scale Software,” Proc. of the 3rd International Workshop on Mining Software Repositories, pp.126–132, May 2006.

[4] H. Kagdi, S. Yusuf, and J.I. Maletic, “Mining Sequences of Changed-Files from Version Histories,” Proc. of the 3rd International Workshop on Mining Software Repositories, pp.47–53, May 2006.

[5] H. Kagdi, J.I. Maletic, and B. Sharif, “Mining Software Repositories for Traceability Links,” Proc. of the 15th International Conference on Program Comprehension, pp.145–154, June 2007.

[6] A.E. Hassan, “The Road ahead for Mining Software Repositories,” Frontiers of Software Maintenance, pp.48–57, Sep. 2008.

[7] 小林隆志，林晋平，“データマイニング技術を応用したソフトウェア構築・保守支援の研究動向,” コンピュータ ソフトウェア，vol.27，no.3，pp.13–23，Aug. 2010.

[8] T. Zimmermann and P. Weißgerber, “Preprocessing CVS Data for Fine-grained Analysis,” Proc. of the 1st International Workshop on Mining Software Repositories, pp.2–6, May 2004.

[9] L.P. Hattori and M. Lanza, “On the Nature of Commits,” Proc. of the 23rd International Conference on Automated Software Engineering Workshops, pp.63–71, Sep. 2008.

[10] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction,” IEEE Trans. on Software Engineering, vol.33, no.11, pp.725–743, Nov. 2007.

[11] B. Fluri and H.C. Gall, “Classifying Change Types for Qualifying Change Couplings,” Proc. of the 14th International Conference on Program Comprehension, pp.35–45, June 2006.

[12] D. Kawrykow and M.P. Robillard, “Non-Essential Changes in Version Histories,” Proc. of the 33rd International Conference on Software Engineering, pp.351–360, May 2011.

[13] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, “Refactoring Edit History of Source Code,” Proc. of the 28th International Conference on Software Maintenance, pp.617–620, Sep. 2012.

[14] A. Hindle, D.M. German, and R. Holt, “What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits,” Proc. of the 5th International Working Conference on Mining Software Repositories, pp.99–108, May 2008.