特別研究報告

題目

修正の分類に基づくリポジトリ再構築手法

指導教員
楠本 真二 教授

報告者
楠 野明

平成 25 年 2 月 12 日

大阪大学 基礎工学部 情報科学科

平成 24 年度 特別研究報告

修正の分類に基づくリポジトリ再構築手法

楠 野明

内容梗概

近年，ソフトウェアリポジトリマイニングが注目されている．ソフトウェアリポジトリにはソフトウェアの開発に関する様々な情報が格納されている．リポジトリに格納されている情報を抽出（マイニング）することによって，将来の開発および保守作業に有益な知見を得ることができる．ソフトウェアリポジトリマイニングにおいては，コミットの情報を利用することが多い．コミットとは，開発者が手元で行ったソースコードなどに対する変更をリポジトリに対して反映させることである．コミットを分析することによって，開発者が過去に行った変更に関する情報を得ることができる．

しかし，コミットの中には他に比べて非常に規模の大きいものが存在する．それらは例えばフォーマット変更などであり，IDE のフォーマット一括変換機能等を用いて自動的にフォーマットが整えられるという修正である．このような規模の大きいコミットの存在は，その他のコミットから抽出した情報の影響を過度に低くしてしまう場合がある．たとえば，各モジュールにおいて過去に修正された総行数を計算することにより，それらに必要であった修正コストを見積もる場合，IDE によるフォーマット変換は考慮に入れるべきではない．もしそれらを考慮に入れてしまうと，正しい見積もりが行えなくなる．このような理由により，規模の大きなコミットはしばしば分析対象から除外される．しかし，フォーマット変換と同時に数行に対してバグ修正を行っているコミットが存在した場合，このコミットの除外によりフォーマット変換による影響を取り除くことができるが，バグ修正の影響も図らず取り除いてしまう．

このような，1 つのコミット中に複数の修正が存在することによって発生する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離する手法を提案する．提案手法を用いることによって，各コミットには一種類のみの修正が含まれるようになる．よって，規模の大きいコミットを除外することによって，数行のバグ修正に関する情報までも取り除いてしまうことがなくなる．

提案手法を実装し，評価実験を行った．ソフトウェア開発者による手動のコミット分離との比較を行った結果，提案手法によるコミット分離の適合率は 67%，再現率は 72%であっ

た．また，大規模なオープンソースソフトウェアに対して適用し，多数のコミットに複数種類の分類が含まれていたことを確認した．

主な用語

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

目 次

1 まえがき **2**

2 準備 **4**

2.1 版管理システム 4

2.2 ビッグコミット 4

2.2.1 定義 4

2.2.2 問題点 5

2.3 フォーマット変換 5

2.4 字句解析 6

2.5 最長共通部分列 6

3 研究動機 **8**

4 修正の分類 **10**

4.1 概要 10

4.2 ステップ 1 : トークンリストの取得 10

4.3 ステップ 2 : LCS の特定 12

4.4 ステップ 3 : トークンの種類における対応関係の特定 12

4.5 ステップ 4 : 空白，タブ，改行文字の変更を分類 13

5 リポジトリの再構築 **15**

6 実装 **17**

6.1 概要 17

6.2 修正の分類の精度を改善するための経験則 17

6.2.1 問題点 1 17

6.2.2 問題点 2 19

7 実験 **22**

7.1 実験 A 23

7.1.1 実験方法 23

7.1.2 実験結果 23

7.2 実験 B 24

7.2.1 実験方法 24

7.2.2 実験結果 24

8 議論 **28**
8.1 実験 A 28
8.1.1 分類がうまくいかなかった例 28
8.1.2 正解集合数とツール出力数間の差について 30
8.2 実験 B 31

9 関連研究 **32**

10 あとがき **33**

謝辞 **34**

参考文献 **35**

図 目 次

1 ビッグコミットの例 5
2 字句解析の流れ 6
3 LCS の例 7
4 複数種類の修正が行われる例 9
5 修正の分類処理の流れ 11
6 フォーマット修正と判断される/されない例 15
7 提案手法の全体図 16
8 問題点 1 のソースコード例 18
9 問題点 1 における手法改善前後の検出結果 18
10 問題点 2 のソースコード例 20
11 問題点 2 における手法改善前後の検出結果 20
12 被験者実験における修正箇所の色分け例 22
13 分割内訳 (トークン) 25
14 分割内訳 (行) 25
15 平均分離数 (トークン) 27
16 平均分離数 (行) 27
17 修正の分類が誤りとされた例 28
18 分類誤りの改善方法 29
19 正解集合数とツール出力数間の差の原因 30

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, 2, 3, 4, 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貯蔵庫のようなものである．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データマイニング手法の総称である．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]．その中で広く分析の対象とされているものとして，ソースコードに関する開発履歴情報がある．開発履歴情報は開発の際に，主に Subversion などの版管理システムを使用することで提供される．開発履歴情報には行われたすべての修正内容が蓄えられており，最新のソースコードからは得られない情報を取得することもできる [8]．特にソースコードの修正履歴は，不具合予測などの分野でその有用性が示されており，その情報を分析対象とした研究が多くの研究者によって行われている [9]．開発者が版管理システムを用いて開発を行う際に，機能追加や削除のようなソフトウェアの振る舞いに影響を与える修正と，フォーマット変換のようなソフトウェアの振る舞いに影響を与えない修正が同時に行われることがある．このような修正の混在はリポジトリマイニングによって得られる知見に影響を与えている可能性がある．例えば，ソフトウェアにおいて今後修正が加わる可能性の高い箇所を予測する際に，頻繁にフォーマット変換が行われているが，ソフトウェアの振る舞いに影響を与える修正が一切行われていない箇所が，今後修正が加わる可能性の高い箇所として特定される可能性がある．このような場合は，対象をソフトウェアの振る舞いに影響を与える修正が行われたものに絞ったほうがより有益な結果が得られると考えられる．

また，一部のソフトウェアリポジトリマイニングでは，前処理として一度に行われた大規模な修正を対象から除外することがある．その理由として，大規模な修正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 [10]．また，一度に行われる修正は規模が小さいものがほとんどである [11]．そのため，このような大規模な修正を分析対象とすると，ソフトウェアの振る舞いに影響を与えない修正によって，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るおそれがある．そのため，大規模な修正を分析対象から除外するという処理が行われている．しかし，大規模な修正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，分析対象からの除外によって本来は分析すべき修正も除外している可能性がある．

本論文では上述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，ソフトウェアの振る舞いに影響を与える修正とソフトウェアの振る舞いに影響を与えない修正に分類し，その分類に基づいて開発履歴情報を再構築する手法を提案する．提案手法では，ソース

コードに加えられた修正を以下の 3 種類に分類している．

- プログラムコードに対する変更
- フォーマットの変更
- コメントに対する変更

開発履歴情報を分析した研究は多数存在するが，開発履歴を再構築する手法はこれまでに提案されていない．開発履歴情報を用いた分析を行う際に，前処理として提案手法を開発履歴に適用することで，ソフトウェアの振る舞いに影響を与える修正のみを分析の対象にすることができ，また大規模な修正に含まれるソフトウェアの振る舞いに影響を与える修正を抽出することができる．

提案手法をツールとして実装し，評価実験を行った．その結果，開発者自身による修正の分離を正解としたとき，提案手法による修正の分離の適合率は約 67%となり，再現率は約 72%という結果になった．また，オープンソースのリポジトリに対して提案手法を適用し，プログラム要素に対する修正とそれ以外の修正が同時に行われているという事例が多数存在することを確認した．

2 章では，論文中で登場する用語や定義の説明を行う．3 章では，研究の動機を説明する．4 章では，提案手法の内，修正を分類する方法について述べる．5 章では，提案手法において，リポジトリを再構築する方法について説明する．6 章では，提案手法を実装したツールについて説明する．7 章では，実施した評価実験の方法と結果を説明する．8 章では，実験の結果をうけての手法の問題点を考察し，改善方法を説明する．9 章では，既存の研究について触れる．10 章では，この研究のまとめを行い，今後の課題を示す．

2　準備

2.1　版管理システム

版管理システムとは，コンピュータ上で作成・編集されたプロジェクト内の様々なファイルの変更履歴を管理するシステムで，バージョン管理システムとも呼ばれる．版管理システムを用いて開発することで，何度も修正を加えられたファイルに関しても，過去の修正内容を確認したり，過去の状態を復元したりすることが容易になる．また，版管理システムには複数人による開発を想定した機能が備えられている．複数人によって開発が行われる際には，同じファイルに対して別々の開発者が修正を行い競合が起こる問題があるが，版管理システムはそれらの問題を解決する機能を提供する．実際に広く用いられる版管理システムとしては，Subversion，CVS，Git などが存在する．

以降に，版管理システムに関するいくつかの用語について述べる．

リポジトリ: 本論文では，版管理システムによって保管される修正履歴の保管庫のことをリポジトリと呼ぶ．

リビジョン: 版管理システムで管理されたソフトウェアに含まれるソースコードの特定の時点での状態を表す．開発者がソフトウェアに対して修正を加え，その修正をリポジトリに反映することで新たなリビジョンが作成される．

コミット: ソフトウェアに対して行われた修正をリポジトリに反映すること．コミットを行うことによってリビジョンが作成される．

2.2　ビッグコミット

2.2.1　定義

ビッグコミットとは，規模の大きなコミットである．規模が大きいとは，開発者が一度にリポジトリに対して反映させる修正内容が多いという意味で，コミットがビッグコミットとなるかどうかはコミットに含まれる修正の量によって定まる．修正の量は，ファイル数，もしくは行数をもとに決定され，一度に多数のファイルに変更を加えた修正情報を持つコミットや，一度に多くの行に変更を加えた修正情報を持つコミットはビッグコミットとされる [11, 12]．

ビッグコミットの例を図 1 に示す．図 1 では，インデントの追加，削除やメソッド名の修正が行われている．約 4000 行に渡ってインデントの変更が行われているので，修正された行数は多いといえる．

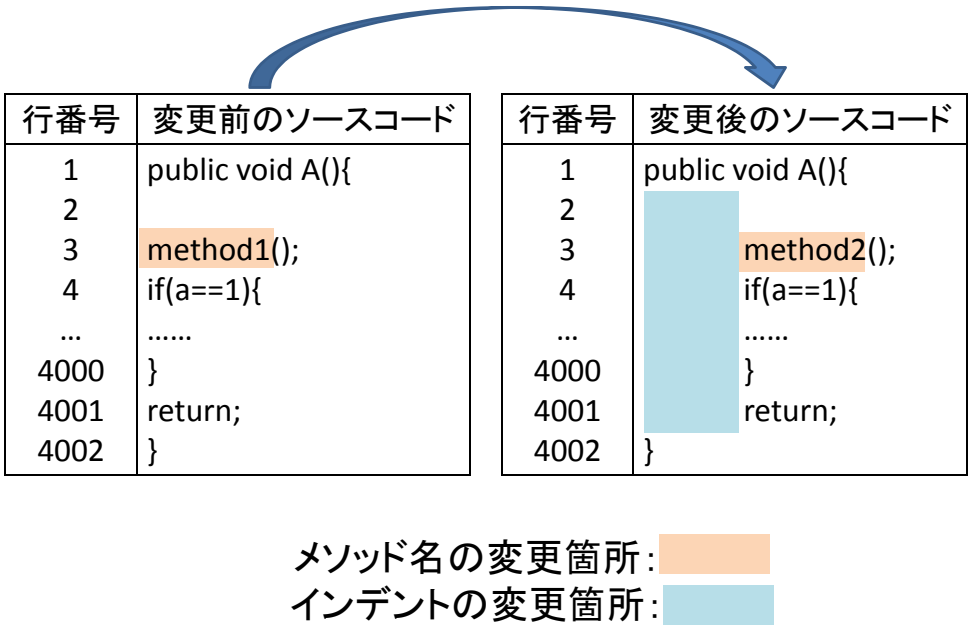

図 1: ビッグコミットの例

2.2.2 問題点

1 章において述べた通り，ビックコミットの問題点は，修正の規模が大きいことである．規模が大きいために，分析結果に与える影響が大きく，規模の小さなコミットが分析結果に与える影響が現れにくくなってしまう．そのためビッグコミットを分析の対象から除外するリポジトリマイニング手法も存在する．

しかし，ビッグコミットを分析対象から除外することにも問題がある．それは，ビッグコミットにも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るためである [11]．このため，ビッグコミットを分析対象から除外することによって，分析結果に影響を与える情報が失われていると考えられる．

2.3 フォーマット変換

フォーマット変換とは，ソースコードの読みやすくするためにインデントを行うことである．開発者がフォーマット変換を行う際にソースコードに挿入されるのは空白や，タブである．また，同じくソースコードを読みやすくする，という目的でソースコード中に改行を挿入することもある．本論文では，これらの空白，タブ，改行の挿入，削除をあわせてフォー

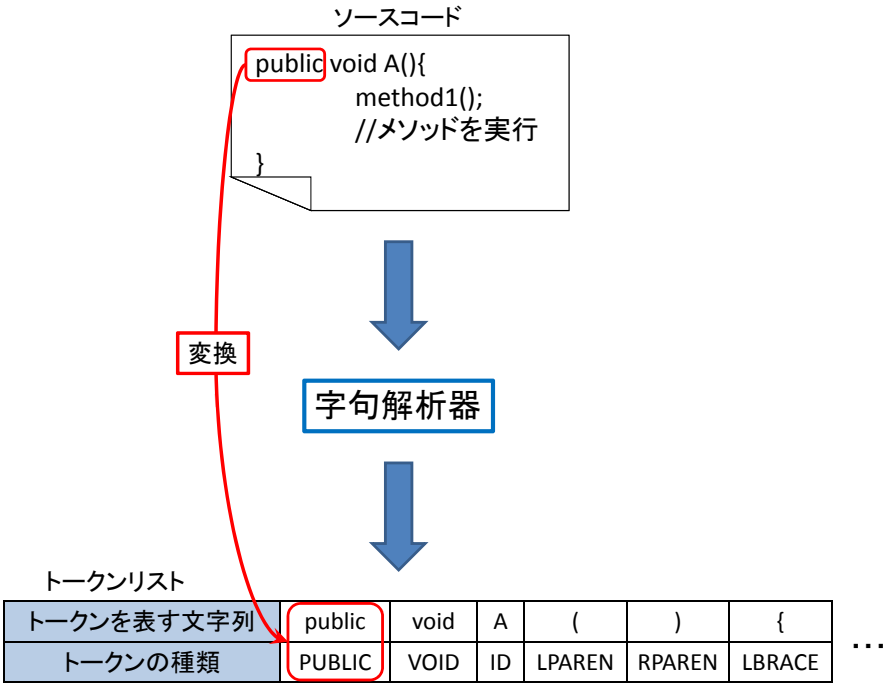

図 2: 字句解析の流れ

マット変換と呼ぶ.

2.4 字句解析

字句解析とは，ソースコードの文字の並びを，トークンの並びに変換することである．字句解析を行うツールは字句解析器と呼ばれる．字句解析は図 2 のように行われる．例では，ソースコードに存在した「public」という文字の並びが，字句解析によって「public」というトークンを表す文字列と,「PUBLIC」というトークンの種類の情報をもったトークンに変換されている．このような変換を繰り返すことで，ソースコードがトークンの並びに変換される．

2.5 最長共通部分列

最長共通部分列 (Longest Common Subsequence. 以下,LCS) は 2 つの要素の列において，存在する共通部分列のうち，もっとも長い系列である．LCS の例を図 3 に示す．図 3 は要素列として，X[BDCABA] と Y[ABCBDA] を与えている．この 2 つの要素列の共通部分列は，[BCA]，[ABA]，[BCBA] と複数存在する．これらの部分列で最も長い [BCBA] が LCS

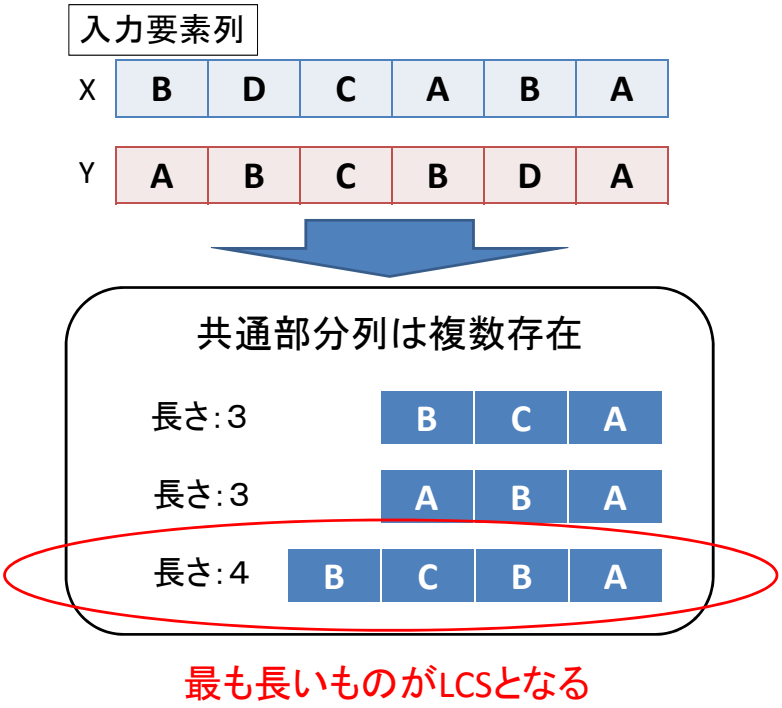

図 3: LCS の例

として特定される.

3　研究動機

図 4 は，オープンソースソフトウェアの「SQurreL SQL Client」のリポジトリに対して行われたあるコミットにおいて，ファイルがどのように修正されたかを示している．図の上部は修正前のリビジョンにおけるソースコードで，図の下部は修正後のリビジョンにおけるソースコードである．このソースコード間の修正には 100 行程度にインデントの変更が行われ，数行に文の追加が行われており，複数種類の修正が混在している．このコミットを分析する際には，文の追加を行っている修正のみを分析する方が，より有益な結果が出ると考えられる．

本論文の提案手法は，リポジトリを入力として，リポジトリを出力する．入力リポジトリの 1 つのコミットに含まれるプログラムコードに対する変更とフォーマットの変更，およびコメントに対する変更を分離し，出力リポジトリに別々にコミットすることで，ソフトウェアの振る舞いに影響を与える修正と，そうでない修正を分離する．これを入力リポジトリに含まれるソースコードの修正履歴全体に対して行うことで，ソフトウェアの振る舞いに影響を与える修正のみを抽出して解析を行うことが可能になる．また，それにより提案手法適用前のリポジトリでは解析の対象から除外されていたソフトウェアの振る舞いに影響を与える修正が，適用後のリポジトリでは解析できるようになることも期待できる．

行番号	変更前のソースコード
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	変更後のソースコード
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 4: 複数種類の修正が行われる例

4　修正の分類

4.1　概要

本章では，修正を分類する手法について述べる．提案手法では，入力リポジトリにおいて連続する 2 つのリビジョン間で行われたファイル群に対する変更を，以下の 4 種類に分類する．

プログラムコードに対する変更: プログラムコードの変更．およびソースファイルの追加・削除．

フォーマットの変更: 空白，タブ，改行文字の追加・削除

コメントに対する変更: コメントの追加・削除，およびコメント内容の変更

ソースファイル以外のファイルに対する修正: ソースファイル以外のファイルに関する追加・削除・変更

以降では，変更されたソースファイルについて，加えられた修正をプログラムコードに対する変更とフォーマットの変更，もしくはコメントに対する変更に分類する方法を説明する．

修正を分類する処理は，図 5 のような流れで，以下の 4 つのステップで行われる．

ステップ 1：**トークンリストの取得**

ステップ 2：**LCS の特定**

ステップ 3：**トークンの種類による対応関係の特定**

ステップ 4：**空白，タブ，改行文字に対する変更の分類**

以降 4.2 章から 4.5 章にて処理をステップに分けて説明する．また，説明は修正前のリビジョンを r_n，修正後のリビジョンを r_{n+1} として r_n，r_{n+1} 間の修正を分類する，という想定で行う．

4.2　ステップ 1：トークンリストの取得

このステップでは，修正されたソースファイルについて，r_n，r_{n+1} におけるそれぞれのソースファイルからトークンのリストを取得する．トークンリストの取得は対象のソースファイルに対して，字句解析を適用することで実現する．一般的な字句解析では，コメントや空白，タブ，改行などは無視されるが，提案手法ではこれらの要素もすべて 1 つのトークンとして取り扱う．

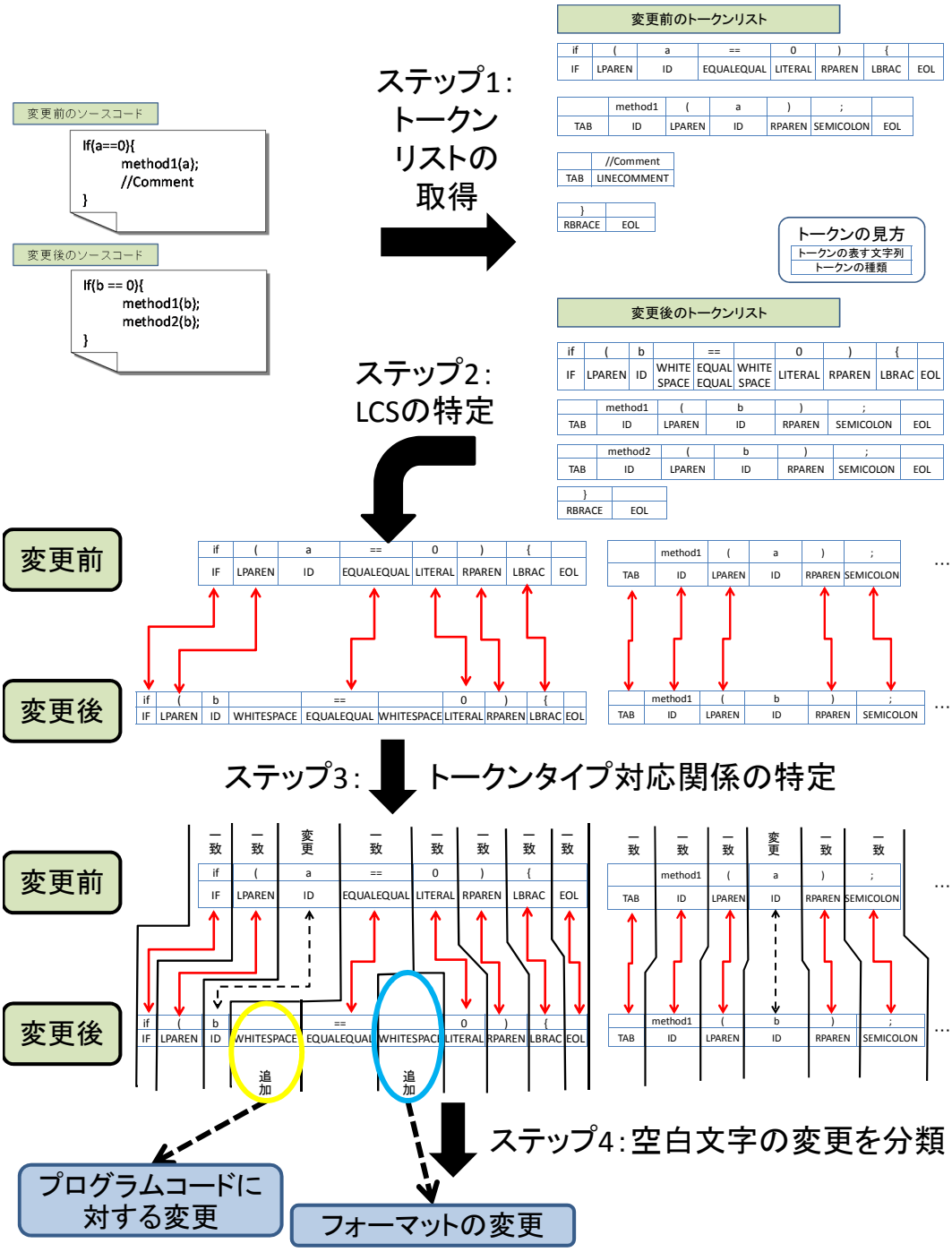

図 5: 修正の分類処理の流れ

4.3 ステップ 2 : LCS の特定

LCS とは最長共通部分列 (Longest Common Subsequence) のことで，このステップではステップ 1 で取得した 2 つのトークンリストから LCS を求める．LCS の特定には最長共通部分列問題のアルゴリズムを使用する．提案手法では，2 つのトークンが表す文字列が一致する場合，2 つのトークンが一致するものとみなす.

4.4 ステップ 3 : トークンの種類における対応関係の特定

このステップでは，トークンリスト間でトークンの種類に基づいたトークンの対応関係を特定する．トークンの対応関係を特定するアルゴリズムを Algorithm1 に示す．このアルゴリズムは，入力としてトークンの対応関係を必要とする．この対応関係はステップ 2 を終えた状態では，LCS におけるトークンの対応関係 (*tokenPairs*) のみが含まれた状態であり，LCS に含まれるトークン以外は，対応関係を持たない状態である．Algorithm1 は LCS に含まれないトークンに，トークンの種類のみを考慮した対応関係を付加し，出力する．トークンの種類のみを考慮した対応関係はトークンの種類が一致するものに対して，LCS に含まれるトークンの順序関係を無視しない範囲で結ばれる．なお，アルゴリズム中の *getTokenPairsIn*(*token1*,*token2*,*token3*, *token4*) は，トークン対応関係のリストから，LCS に含まれる対応関係である *token1*，*token3* の対応と *token2*，*token4* の対応に挟まれた部分リストを取得する．また，HEAD と TAIL は特殊なトークンを表しており，HEAD はトークンリストの先頭を，TAIL はトークンリストの末尾を表している．

以上の処理によって，トークン単位の修正情報が特定される．特定された対応関係は以下の 4 つに分けられる．

- 一致 : 求めた LCS に含まれる，種類も文字列も同一の 2 つのトークンの対応関係
- 修正 : LCS には含まれないが，トークン種類の比較では対応するトークンが存在し，そのトークンらのトークンリスト内での順序が入れ替わることで，LCS の対応関係に修正が起こらない，トークン同士の関係
- 追加 : 後のリビジョンのトークンリストにのみ存在し,「修正」の関係ももたないトークンの関係
- 削除 : 前のリビジョンのトークンリストにのみ存在し,「修正」の関係ももたないトークンの関係

以上の 4 種類の内，修正，追加，削除がトークン単位の修正情報である.

Algorithm 1 トークンの対応関係の特定

Require: *tokenPairs* (a list of token pairs)

Ensure: *tokenPairs* (a list of token pairs)

```
previousBeforeToken ← HEAD
previousAfterToken ← HEAD
for all pair ∈ tokenPairs do
  if isLCSPair(pair) then
    currentPairs ← getTokenPairsIn(previousBeforeToken, pair.getBeforeToken(),
                                   previousAfterToken, pair.getAfterToken())
    currentPairs.setTokenTypePairs()
    previousBeforeToken ← pair.getBeforeToken()
    previousAfterToken ← pair.getAfterToken()
  end if
end for
currentPairs ← getTokenPairsIn(previousBeforeToken, TAIL, previousAfterToken, TAIL)
currentPairs.setTokenTypePairs()
```

4.5 ステップ 4：空白，タブ，改行文字の変更を分類

このステップでは，前ステップで得られたトークン単位の差分の内，空白文字の追加・削除に注目し,「フォーマットの変更」をそれ以外の「プログラムコードに対する変更] と「コメントに対する変更」から分類する．分類は，その空白文字の追加・削除がプログラムコードに対する変更に伴う修正であるかどうかで判断される．例を図 6 に示す．図 6 では，フォーマットの変更と判断される例と，判断されない例が示されている．2 つの例はどちらも一致の関係の間で挟まれている．この際に空白文字の修正をどちらに分類するかは，挟まれた同じブロックにプログラムコードに対する修正と分類される追加，もしくは削除の関係が存在するかどうかで判定する．図 6 の例では右は同じブロックにプログラムコードに対する変更が存在しないので,「フォーマットの変更」と判定され，左の例だと同じブロックにプログラムコードに対する変更が存在するため「プログラムコードに対する変更」と判定される．

Algorithm 2 *setTokenTypePairs*

Require: *tokenPairs* (a list of token pairs)

Ensure: *tokenPairs* (a list of token pairs)

```
for all pair ∈ tokenPairs do
  if ¬pair.havePair() then
    if pair.getBeforeToken() ≠ NULL then
      beforeToken ⇐ pair.getBeforeToken()
      for all anotherpair ∈ ittokenPairs do
        if beforeToken.isSameType(anotherpair.getAfterToken()) then
          pair.setPair(anotherpair)
        end if
      end for
    else
      afterToken ⇐ pair.getAfterToken()
      for all anotherpair ∈ ittokenPairs do
        if afterToken.isSameType(anotherpair.getBeforeToken()) then
          pair.setPair(anotherpair)
        end if
      end for
    end if
  end if
end for
```

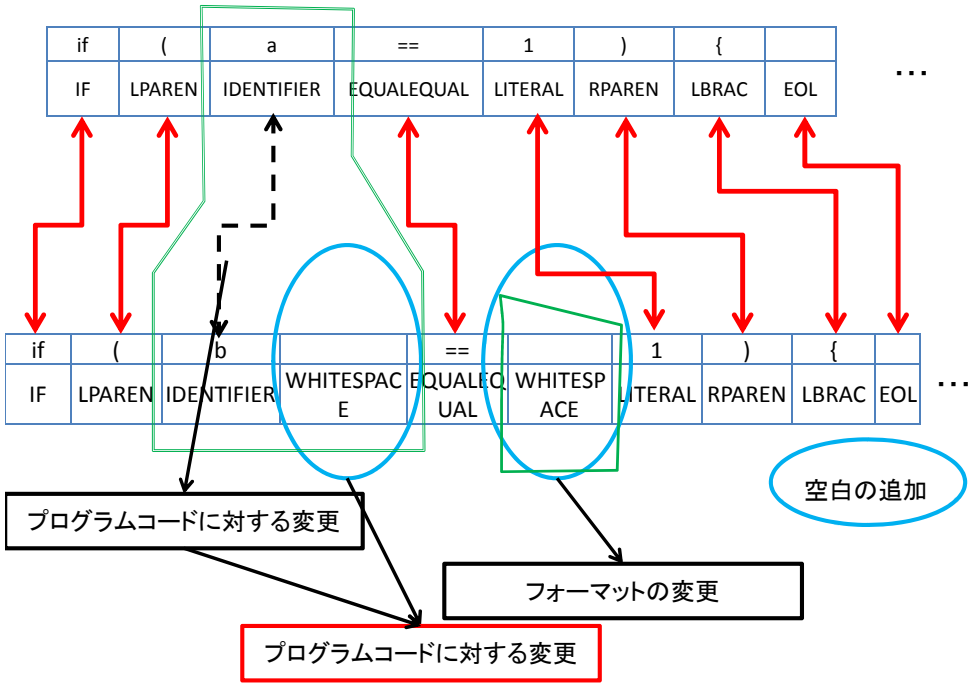

図 6: フォーマット修正と判断される/されない例

5　リポジトリの再構築

リポジトリの再構築は次のように行われる．入力リポジトリについて，2 つの連続するリビジョンを比較し，修正情報を特定，分類する．この修正情報の分類は 4 章で述べた流れで行われる．次に分類した修正を 1 種類ずつ出力リポジトリにコミットする．これにより入力リポジトリでは 1 度のコミットで行われていた修正を最大 4 つのコミットに分割する．これをすべてのリビジョン間について繰り返し行うことでリポジトリを再構築する．例を図 7 に示す．図 7 ではリビジョン r とリビジョン r+1 間で行われた修正が 4 つの修正に分類されている．また，リビジョン r' はリビジョン r と同じソースコードの状態を記録しているとする．分類された修正の内，フォーマットの変更をリビジョン r' に適用しコミットを行い，リビジョン r'+1 が作成される．次にフォーマットの変更が適用されたリビジョン r'+1 に対してコメントに対する変更を適用しコミットを行い，リビジョン r'+2 が作成される．同様にプログラムコードに対する変更を適用しコミットを行うことで，リビジョン r'+3 を，ソースファイル以外のファイルに対する修正を適用しコミットを行うことでリビジョン r'+4 を作成する．また，リビジョン r とリビジョン r+1 の間で行われていた修正は修正の分類によって,「プログラムコードに対する変更」,「フォーマットの変更」,「コメントに対する変更」

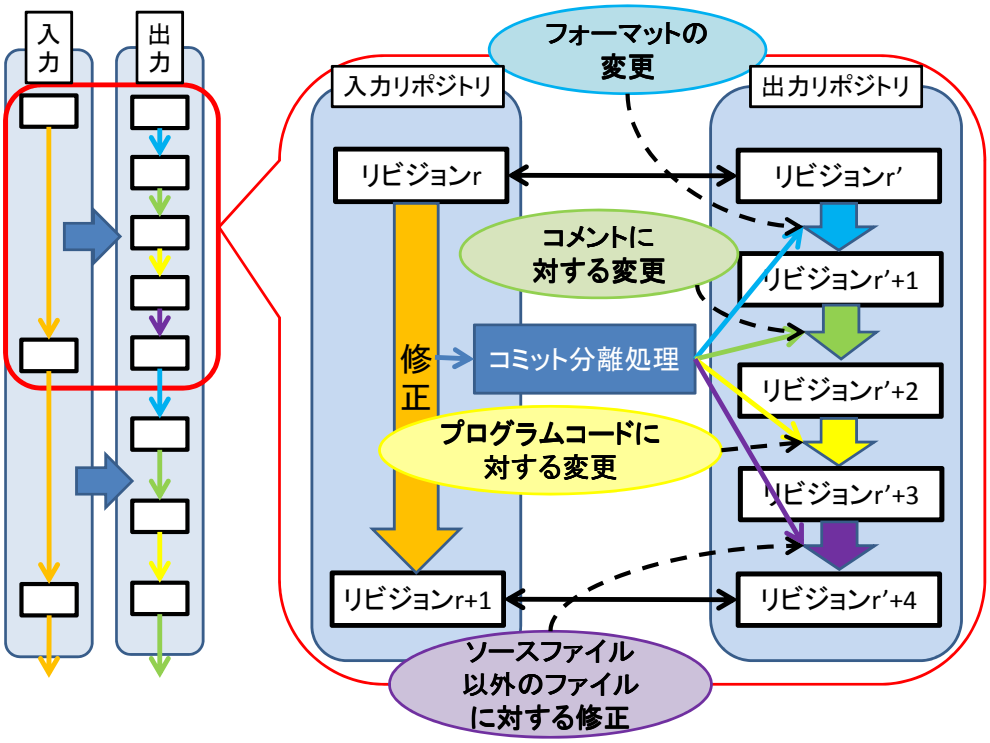

図 7: 提案手法の全体図

「ソースファイル以外の修正」のいずれか 1 つに必ず属し，かつ高々1 つに属するので，4 種類の変更をすべて適用したリビジョン r'+4 に含まれるソースコードはリビジョン r+1 に含まれるソースコードと完全に一致する．

6　実装

6.1　概要

ツールは Java を用いて実装した．現在のところツールが対象としているのは，Subversion を用いて版管理され，Java で記述されたソースコードの修正履歴をもったリポジトリである．提案手法を組み込んだツールは，リポジトリを入力として，全てのコミットの修正内容を分離し再構築したリポジトリを出力する．

6.2　修正の分類の精度を改善するための経験則

提案手法が特定するトークン単位の修正分類を，よりソフトウェア開発者の認識に近づけるために，修正箇所特定と修正の分類それぞれに関して改善策を実装した．改善を行った問題点は以下の 2 点である．

問題点 1： 行末の頻出トークンに関する LCS の特定誤り

問題点 2： コメント追加 (削除) 前後に出現する空白文字追加 (削除) の修正分類誤り

以降で各問題点の改善方法を説明する．

6.2.1　問題点 1

この問題点は，提案手法において LCS を用いて修正箇所を特定するために起こるものである．LCS の特定がトークンリストの後方から行われるために，変更前後にソースコードにおいて対応する箇所の特定に誤りが生じるのである．この問題の例を図 8 に示す．図 8 では文の追加が行われている．そのため正しい修正箇所は図 8 下部のように 1 行全体であるのに対し，図 8 上部のツールが出力した修正箇所は 2 行にまたがるものになっている．

以降では，図 8 のソースコードを字句解析したトークンリストを用いて，図 9 で説明する．この問題の発生原因はステップ 2 において LCS を特定するために用いるアルゴリズムにある．LCS の特定は，修正前後のソースファイルから取得したトークンリストの後方から比較を行い，対応関係を特定していく形で行われるため，図 9 の場合は修正後のソースコードにおいて追加されたトークンの一部である [“)”，“;”，“(改行)”] と，修正前のソースコードにも修正後のソースコードにも存在する [“)”，“;”，“(改行)”] の間に対応関係を結んでしまっている．

この問題を改善するために，手法のステップ 3 終了後に対応関係を修正する機能を実装した．以下において追加した処理を 3 つの手順に分けて述べる

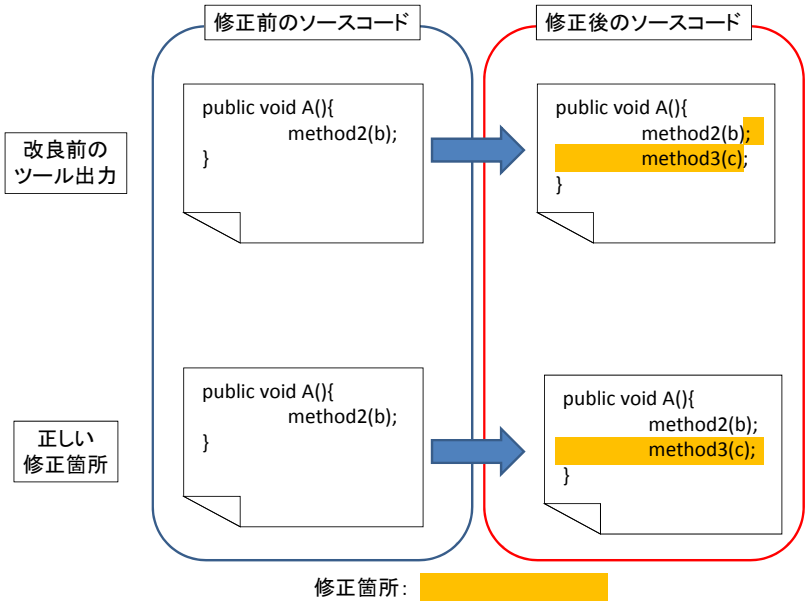

図 8: 問題点 1 のソースコード例

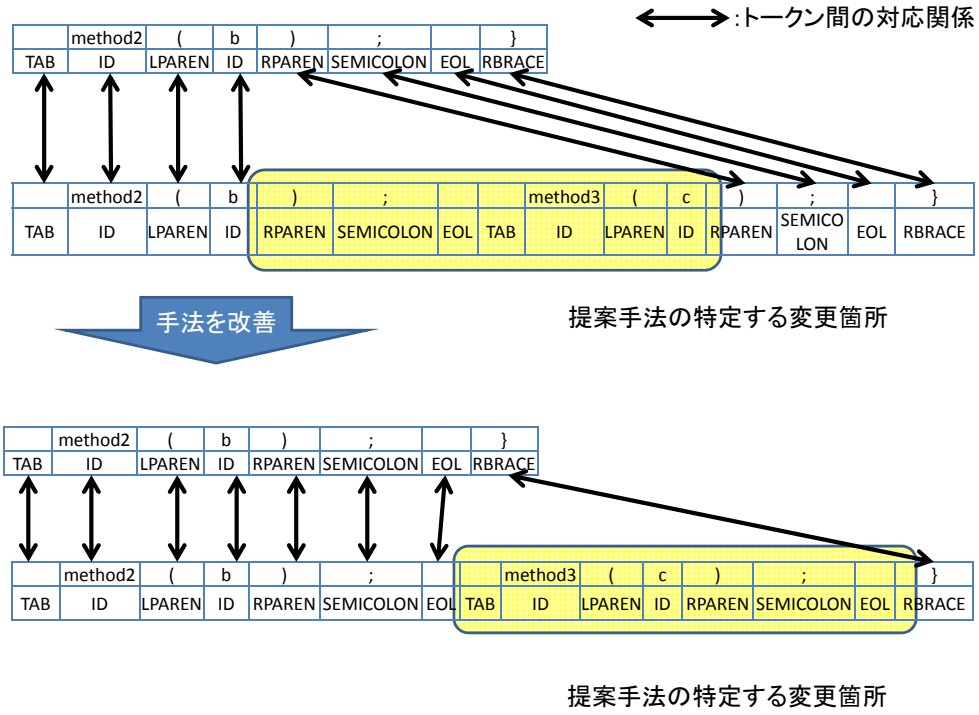

図 9: 問題点 1 における手法改善前後の検出結果

手順 1： トークンが追加，もしくは削除された箇所を対象に，トークン同士の対応関係を保存したリストを探索する．探索は，トークンリスト先頭のトークンに関するものから順に行う．また，対応関係のリストには，トークンの追加，および削除は対応するトークンが存在しない対応関係として保存されている．

手順 2： 特定された追加，もしくは削除されたトークンの種類が「“)”，“}”，“;”，“(改行)”」の何れかであれば，一致，もしくは修正の対応関係が現れるか，リストの末端まで次の対応関係を確認する．発見された追加，もしくは削除されたトークンの種類が上記の何れかでない場合や，リストの末端に到達した場合はこの処理は終了する

手順 3： 現れた一致，もしくは修正の対応関係に含まれるトークンを確認し，それが手順 1 で発見されたトークンの種類と一致していたならば，手順 1 で発見したトークンをその対応関係に含め，対応関係に含まれるトークンの内，手順 1 で発見したトークンと同じトークンリストに含まれるものを対応関係から取り除き，追加，もしくは削除されたトークンとする．

手順 3 を終えたのちは，手順 1 で発見した対応関係の次の対応関係から，手順 1 と同じ探索を再開し，この処理をリストの末端まで繰り返す．

この処理を図 8 のソースコードに対して行うと，図 9 のように修正箇所の特定を改善できた．図 9 では，修正箇所として特定された部分列の先頭と部分列の直後の対応関係をもったトークンが，どちらも “}” であるので，対応関係を修正している．これを複数回繰り返すことで図 9 上部のトークン間の対応関係は，下部のように変化した．

6.2.2 問題点 2

この問題点は，提案手法が行う分類方法が原因で起こる．コメントの追加・削除が行われた際に，その前後で行われた空白，タブ，改行文字の追加・削除をコメントに対する変更とするべきところを，フォーマットの変更と分類してしまっているのである．図 10 に例を示す．図 10 ではコメントの追加とそれと同時に，タブと改行の追加が行われている．しかし，図 10 下部のように全てコメントに対する変更とすべきであるのに対し，ツール改善前はコメントの追加のみをコメントに対する変更と分類し，タブと改行の追加はフォーマット変更としてしまっている．

この問題を改善するための処理を，問題点 1 を解決するための処理の後に追加した．追加した処理の手順を 5 つに分けて以下で説明する．

手順 1： コメントの追加，もしくは削除された箇所を対象に，トークン同士の対応関係を保存したリストを探索する．探索は，トークンリスト先頭のトークンに関するものか

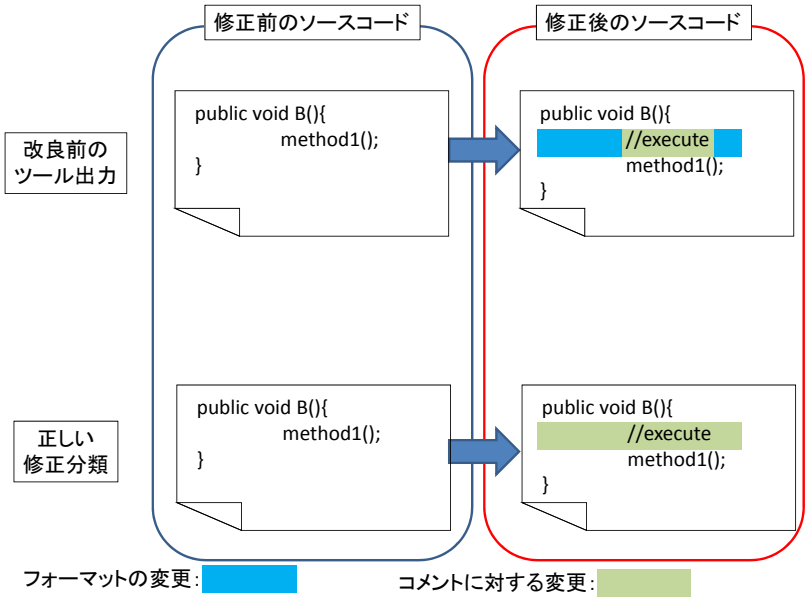

図 10: 問題点 2 のソースコード例

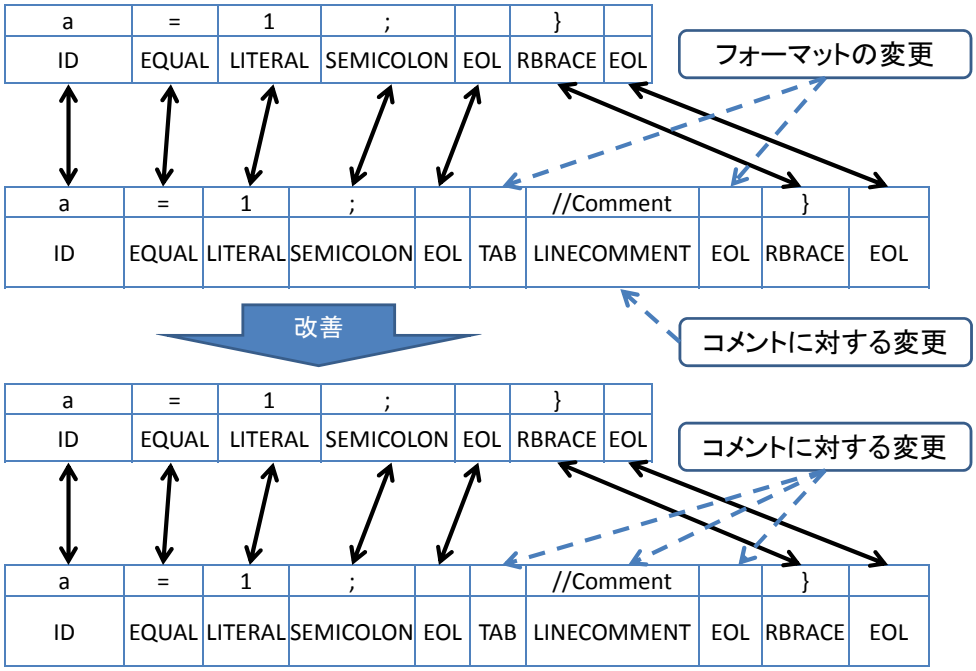

図 11: 問題点 2 における手法改善前後の検出結果

ら順に行う

手順 2： コメントの追加 (削除) された箇所を発見したら，空白またはタブの追加 (削除) を対象に，発見した対応関係の手前の対応関係から順次トークンリストの先頭へ向かって探索を行う．

手順 3： 空白，またはタブの追加 (削除) を発見したら，コメントの修正と分類し，探索を再開する．以下の何れかの条件を満たす対応関係を発見するまでは手順 3 を繰り返す．

- 一致，または修正の関係をもつトークンの対応関係
- 改行の追加 (削除)
- プログラム要素の修正

以上の条件を満たしたら次の手順に移行する．

手順 4： 手順 1 で発見したコメントの追加 (削除) された箇所から，手順 2 と同じように探索を開始する．ただし，探索はトークンリストの末尾に向かって行う．

手順 5： 手順 4 で対象の箇所を発見したら，手順 3 と同様の処理を行う．ただし，手順 3 の条件を満たした際に発見した対応関係が「改行の追加」であった場合には，発見した改行の追加もコメントの修正に含めて，処理を終了する．

以上の処理を行い，修正の分類は図 10 下部のように改善した．図 11 上部では，コメントに対する変更の前後で行われているタブ，改行の追加がフォーマットの変更と分類されている．処理を追加したことにより，図 11 下部のようにコメントに対する変更の前後で行われているタブ，改行の追加をコメントに対する変更に含めることができている．

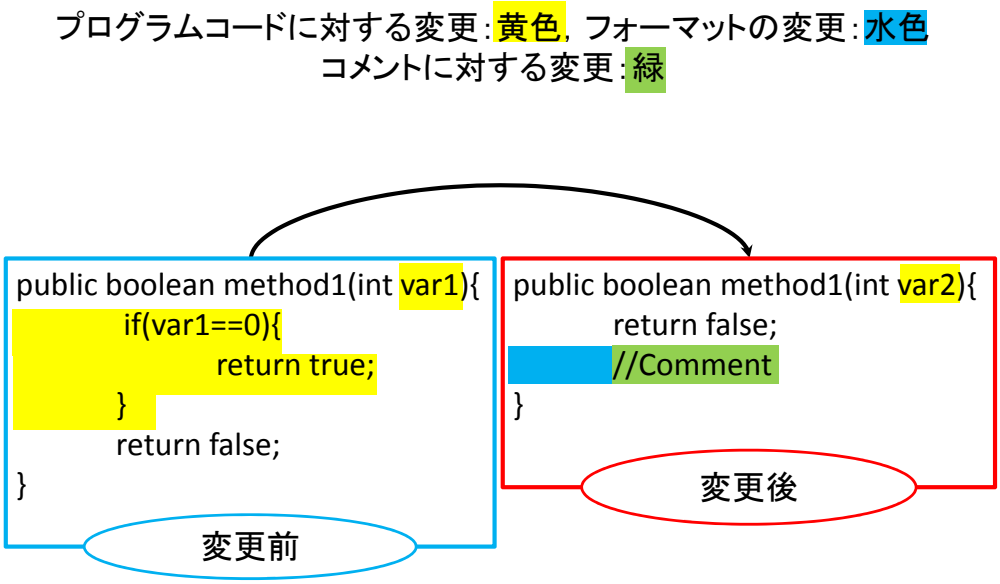

図 12: 被験者実験における修正箇所の色分け例

7 実験

本研究では以下の 2 つの評価実験を行った.

実験 A： 提案手法が行う修正の分類について，分類の正しさを確認する．被験者を用いて行った.

実験 B： 提案手法を用いることで, 修正をどのように分離できるかを確認する．オープンソースのリポジトリを用いて行った.

上記の評価実験を，Java を用いて提案手法を実装したツールを使って行った．現在のところ，Java で記述されており，かつバージョン管理システム Subversion を用いて管理されているソフトウェアのみをツールは対象としている.

7.1 実験 A

7.1.1 実験方法

この実験は，被験者自身が作成したソフトウェアのリポジトリを対象とする．実験の準備を以下の手順で行った．

手順 1： 対象にツールを適用する．

手順 2： 提案手法を適用する前のリポジトリから，1 度のコミットで 2 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見つける．

手順 3： 実験データとするファイルに含まれた修正箇所を修正の種類に応じて色分けする．修正箇所の色分けの例を図 12 に示す．

以上の手順で準備した実験データを被験者に見てもらい，各修正箇所に対して，以下の 2 つのことを確認してもらい，提案手法の行う分類の正しさを確認した．

- 修正の範囲は正しいか
- 修正の分類は正しいか

7.1.2 実験結果

被験者自身が作成したリポジトリから実験データを作成し，各被験者，自身が作成した 5 つのファイルに対して実験を行った．結果は表 1 のようになった．表の横軸は，被験者を表し，縦軸はそれぞれ以下のことを表している．

正解集合: ツールの出力と被験者の意見をもとに構成された各種類の修正箇所の数

表 1: 実験 A 結果

被験者	正解集合				ツール出力				出力正解数				適合率	再現率
	P	F	C	合計	P	F	C	合計	P	F	C	合計	(%)	(%)
1	126	20	31	177	286	34	34	354	102	18	21	141	39.83	79.66
2	17	10	9	36	19	9	9	37	16	8	8	32	86.49	88.89
3	61	21	33	115	61	23	33	117	60	21	31	112	95.73	97.39
4	284	166	40	490	247	88	40	375	192	77	38	307	81.87	62.65
総合	488	217	113	818	613	154	116	883	370	124	98	592	67.04	72.37

ツール出力: ツールが特定した各種類の修正箇所の数

出力正解数： ツール出力に含まれる修正箇所の内，正解集合と，修正の範囲と修正の種類の両方が一致している修正箇所の数

適合率： ツール出力に含まれる修正の内，被験者に正解とされたものの割合

再現率： 正解集合に含まれる修正の内，ツールが出力できたものの割合

また，表の P，F，C はそれぞれ変更の種類を表しており，P はプログラムコードに対する変更を，F はフォーマットの変更を，C はコメントに対する変更を示す．また，適合率，再現率は以下のように計算した．

$$(\text{適合率}) = \frac{\text{出力正解数}}{\text{ツール出力の修正箇所数}} * 100$$

$$(\text{再現率}) = \frac{\text{出力正解数}}{\text{正解集合の個数}} * 100$$

計算の結果，適合率は 67.04%となり，再現率は 72.37%となった．

また，プログラムコードに対する変更をソフトウェアの振る舞いに影響を与える修正，フォーマットの変更，およびコメントに対する変更をソフトウェアの振る舞いに影響を与えない修正とみなして，それぞれに対して実験結果をまとめたところ，表 2，3 のようになった．表 3 では，コメントに対する変更をフォーマットの変更と判定した場合と，フォーマットの変更をコメントに対する変更と判定した場合は誤った分類としていない．

その結果，ソフトウェアの振る舞いに影響を与える修正については適合率が 60.36%，再現率が 75.82%となり，ソフトウェアの振る舞いに影響を与えない修正に関しては，適合率が 80.74%，再現率が 66.06%となった．

7.2 実験 B

7.2.1 実験方法

オープンソースのソフトウェアである “SQuirreL SQL Client” の Subversion リポジトリに対して，提案手法を実装したツールを適用し，リポジトリの再構築を行った．その後，入力リポジトリと出力リポジトリ内の各リビジョン間で行われた修正を，行数とトークン数でそれぞれ取得し比較した．

7.2.2 実験結果

実験の結果，“SQuirreL SQL Client” は入力リポジトリのリビジョン数 6,737 に対して，出力リポジトリのリビジョン数は 12,217 となった．

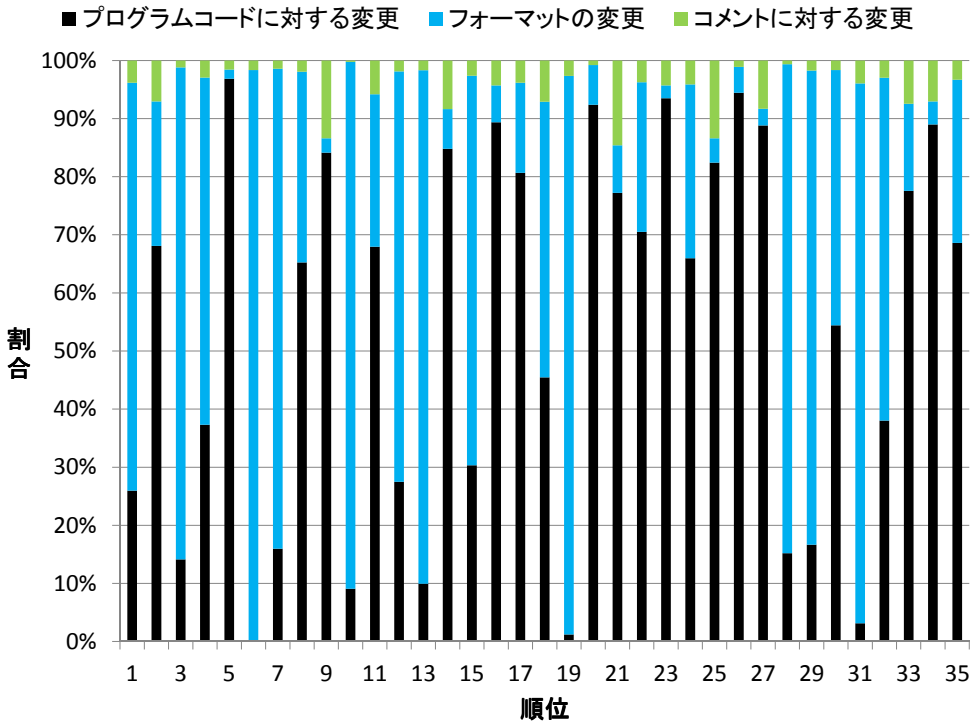

図 13: 分割内訳 (トークン)

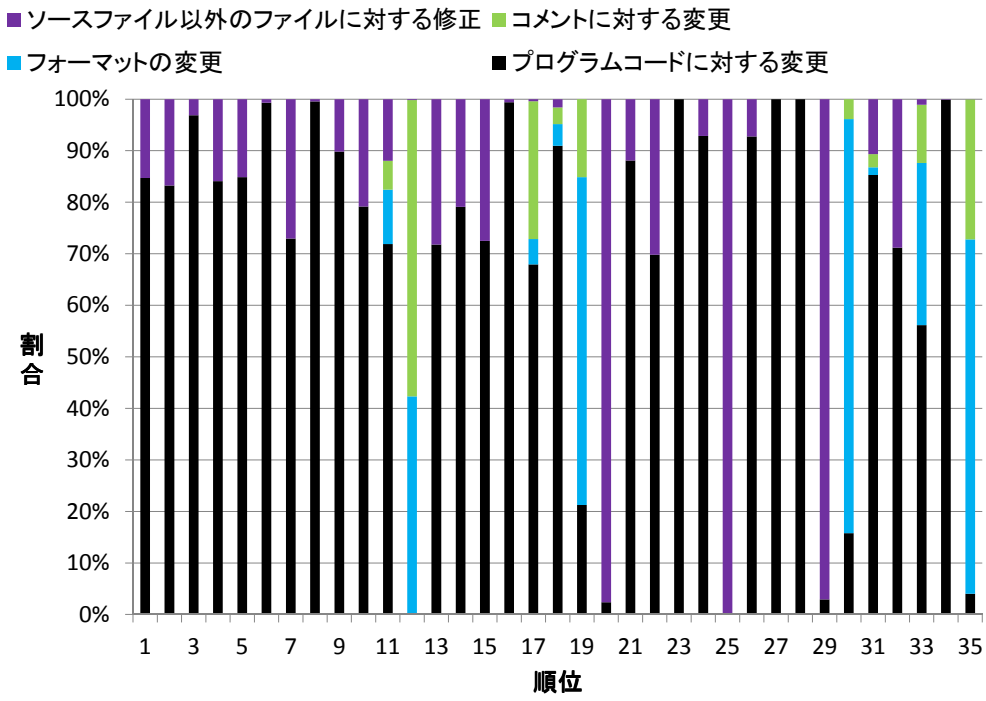

図 14: 分割内訳 (行)

また，入力リポジトリにおいてコミットを，修正量の降順に並べ，上位 1%のコミットについてどのように分離が行われたかを確認したところ，図 13，14 のようになった，横軸は 1 つのコミットを表し，縦軸は 1 つのコミットにおける，各種類の変更の割合を示している．たとえば，図 13 の 1 番左のコミットでは合計 79,871 個のトークンに変更が行われており，その内，プログラムコードに対する変更が 20,751，フォーマットの変更が 56,050，コメントに対する変更が 3,070 で各変更が行われたトークン数が合計に占める割合がグラフに示されている．

また，図 15，16 は各コミットが何種類の変更を含んでいたかを表している．グラフの横軸はコミットを表し，縦軸は含まれている変更の種類数の平均値を表している．グラフでは，コミットを修正量で降順に並べた後に上位から 50 個ずつ含まれている変更の種類数を取得し，その平均をとった値を折れ線の各点としている．たとえば，図 13 の場合は，1 番左の値は修正量が多い順の 1-50 位のコミットに含まれる変更の種類数の平均を，その 1 つ右の値は 51-100 位のコミットに含まれる変更の種類数の平均を，の様にして折れ線グラフの各点がプロットされている．

表 2: 実験 A 結果 (ソフトウェアの振る舞いに影響を与える修正)

被験者	正解集合	ツール出力	出力正解数	適合率 (%)	再現率 (%)
1	126	286	102	35.66	80.95
2	17	19	16	84.21	94.12
3	61	61	60	98.36	98.36
4	284	247	192	77.73	67.61
総合	488	613	370	60.36	75.82

表 3: 実験 A 結果 (ソフトウェアの振る舞いに影響を与えない修正)

被験者	正解集合	ツール出力	出力正解数	適合率 (%)	再現率 (%)
1	51	68	39	57.35	76.47
2	19	18	14	77.78	73.68
3	54	56	50	89.29	92.59
4	206	128	115	89.84	55.83
総合	330	270	218	80.74	66.06

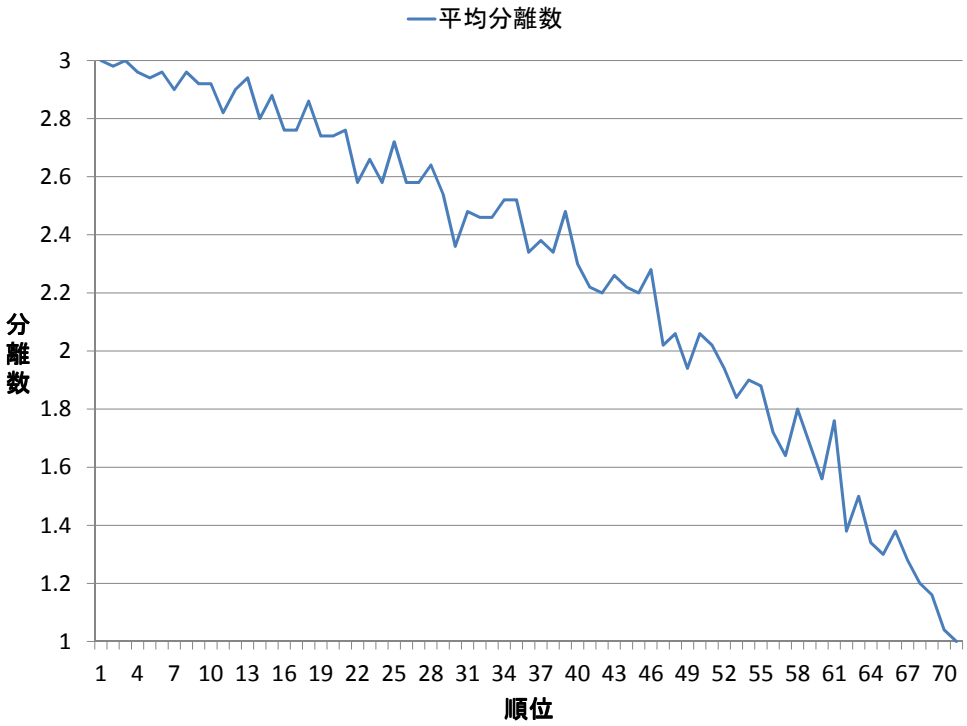

図 15: 平均分離数 (トークン)

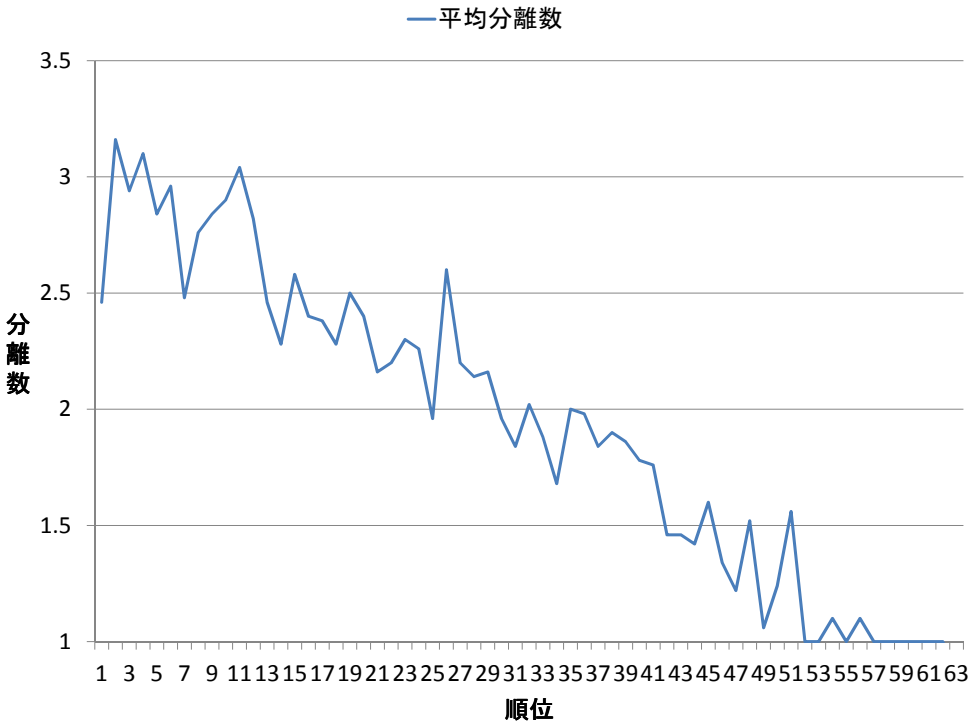

図 16: 平均分離数 (行)

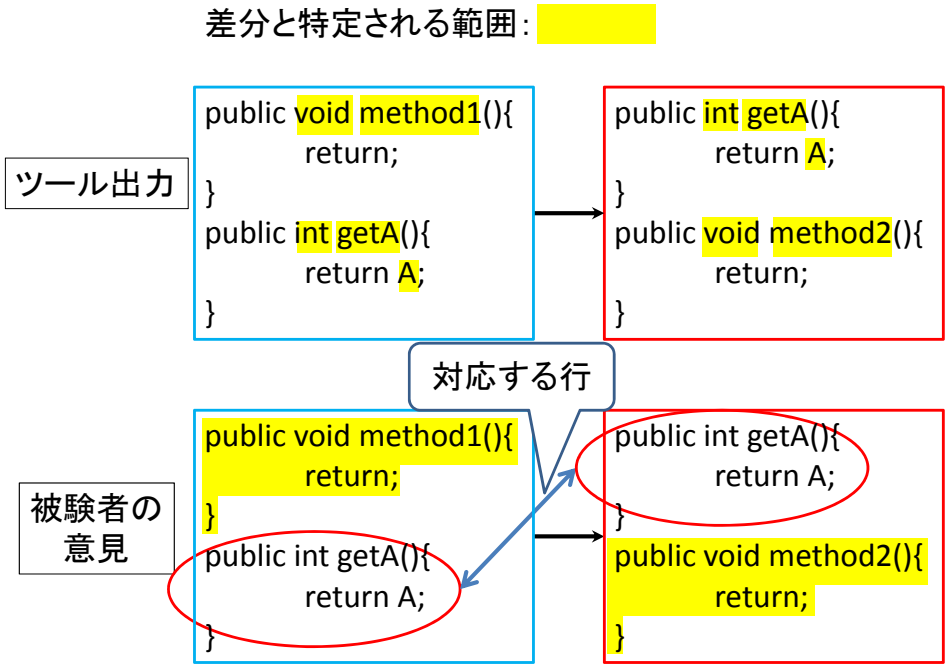

図 17: 修正の分類が誤りとされた例

8 議論

8.1 実験 A

8.1.1 分類がうまくいかなかった例

実験 A において行われた分類の内，被験者に誤った分類であるとされた，すなわち分類がうまく行えなかった例について述べる．また，誤った分類を行わないようにツールを改善する方法について説明する．

図 17 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．図において，黄色のハイライトで強調されているコードは差分とする位置を示している．上半分はツールの出力であり，下半分は被験者の意見を反映させた差分の位置である．図の左半分に書かれたソースコードは変更前の状態であり，左側は変更後を表す．表 1 において，ツール出力の内，正解出力数に含まれていないものは，被験者によって誤りと判定されたものである．この誤りの原因は，LCS の特定アルゴリズムにある．LCS は 2 つの字句列に対して，最長の共通部分列をとるものであり，かつ字句は空白であれプログラムコードに関するものであれ，価値は変わらず 1 つの字句として扱う．これにより，被験者が対応している，と考える字句を

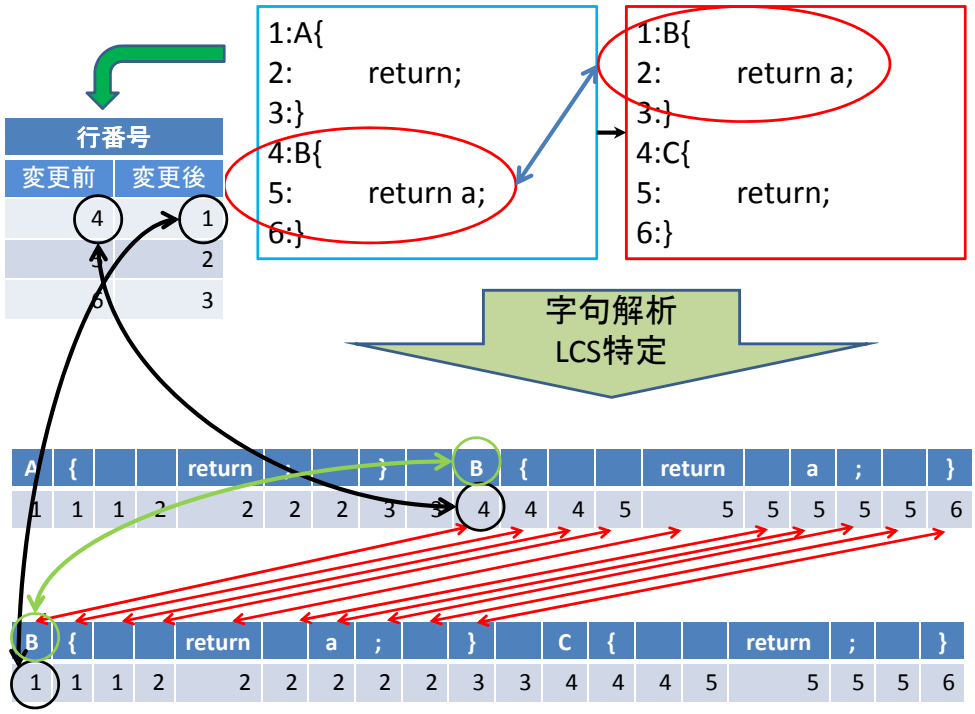

図 18: 分類誤りの改善方法

LCS に含めることができなかったと考えられる．

この誤りの改善方法としては，LCS 特定の際に行単位の diff 情報を用いることが考えられる．改善方法は図 18 のように行う．まず，変更が加えられたファイルについて，修正前後のソースコードを入力として行単位の diff を取得する．これにより行同士の対応関係を取得する．そして変更前後のソースコードから取得した 2 つの字句列と行同士の対応関係を入力として LCS を特定する．特定の際は，以下の 2 つの条件を同時に満たすもののみを，共通の要素とみなす．

- 字句の文字列が一致する
- 比較している 2 つの字句それぞれの行番号が，行同士の対応関係と一致している

この，現在の提案手法における LCS 特定法よりも条件の厳しい LCS 特定のステップを，現在の LCS 特定のステップの前に行うことで，行同士の対応関係を考慮して LCS を特定し，この問題を解決することができると考えられる．

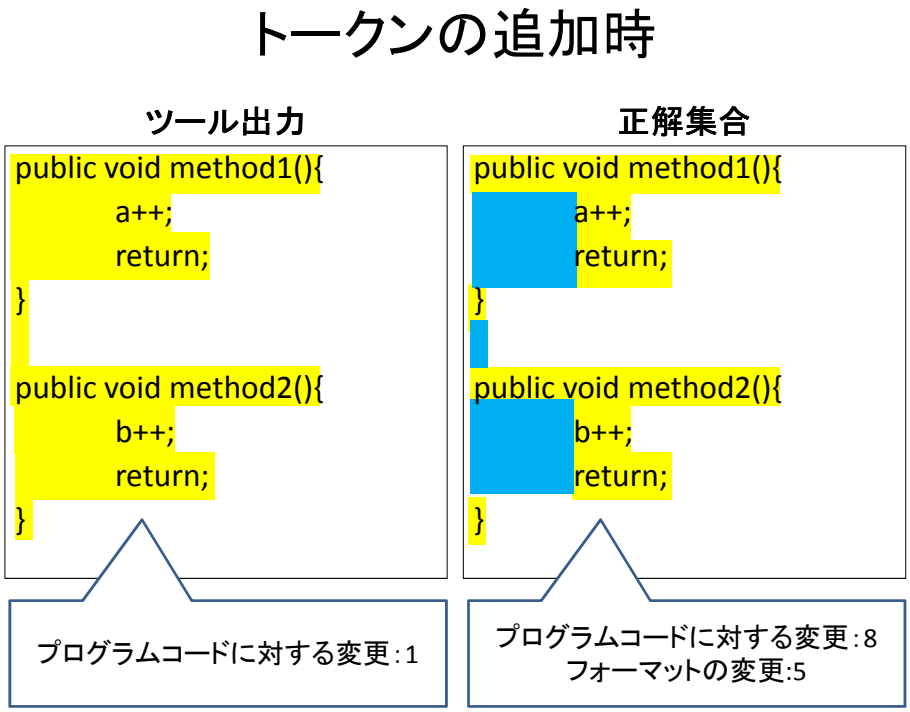

図 19: 正解集合数とツール出力数間の差の原因

8.1.2 正解集合数とツール出力数間の差について

7 章における表 1 の値に関して考察する．被験者 4 において，正解集合の数が 490 であるのに対して，ツール出力は 375 と大きな差がある．これは図 19 のような例が原因である．複数のトークンがまとまって追加されたという修正に関して，ツールはそれらを 1 つのまとまったプログラムコードに対する修正と判定している．それに対して正解集合では追加されたトークンの内，空白，タブ，改行文字の追加をフォーマットの変更と判定している．そのため，1 つのまとまりであった変更が，複数の変更に分離されている．このようにしてツール出力と正解集合の数値の差が発生している．また，被験者 1 の正解集合，ツール出力の数値においても大きな差があるが，これは図 19 の反対で，図の左側が正解集合で，図の右側がツール出力のような状態になっている．また，図 19 のような例は，実験結果における適合率，および再現率の低下にも関係しており，図 19 の場合は，合計 14 個誤り数を増やす原因となっている．しかしながら，図 19 の例に関しては，ツール出力を正しいと判定する被験者と，誤りと判定する被験者の両方が存在するため，この誤りは被験者に合わせてツール出力を変更することでしか解決できないと考えられる．

8.2 実験 B

7 章で実験 B の結果として述べた図 14 を見ると，プログラムコードに対する変更が含まれる割合の多いコミットが多い．これは 1 章にて述べた,「大規模な修正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 [10]」という大規模な修正の特徴と異なるように考えられる．本章ではこの結果について議論する．

図 14 において，もっとも規模の大きいコミットの分離内訳を表 4 に示す．表 4 はリビジョン 5,833 とリビジョン 5,834 の間で行われた修正の内訳を示しており，表の縦の項目はそれぞれ，P はプログラムコードに対する変更，F はフォーマットの変更，C はコメントに対する変更，その他はソースファイル以外のファイルに対する修正を表している．修正量を見ると，修正が加わった行数が 290,442 行であり，そのうちプログラムコードに対する変更は 245,982 行と非常に多くの割合を占めている．しかし，このリビジョン間の変更の内容はフォルダ移動であった．このフォルダ移動により，大量のファイルが移動し，結果として大量の行の修正が行われたと判定されている．しかし，このファイル移動はファイルの位置を変更したのみで，内容には修正が行われておらず，ソフトウェアの振る舞いに影響を与えているとは言えない．提案手法では，ソースファイルの追加，削除をすべてプログラムコードに対する変更に含めるため今回の実験ではこのようになったと考えられるが，将来的にはこのフォルダ移動のような修正もソフトウェアの振る舞いに影響を与えない変更として分離すべきであると考えられる．

表 4: リビジョン 5,833-リビジョン 5,834 における修正内訳

	P	F	C	その他	合計
修正量 (行)	245,982	0	0	44,460	290,442
割合 (%)	84.69	0	0	15.31	100

9 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 [13]．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 [14] 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 [15]．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に対する修正の内，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する手法を提案している．しかし，これらの手法はソースコードを比較する際に空白や改行文字，コメントを無視しているという点で提案手法と異なる．提案手法は，フォーマットの修正，コメントに対する修正として，これらを分離しているという点で，既存の修正を分類する手法とは異なる．また，修正の分類に基づいて，リポジトリを再構築している点も，これらの手法にはない大きな特徴である．

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 上に開発している [16]．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，その操作を自由に順序入れ替えや，統合，取り消すことを可能にしている．複数の変更が混在した履歴についても，変更を分離し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 1-5 つ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修正が存在することを指摘している [11]．Hindle らの報告によれば，大規模な修正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 [12]．

10　あとがき

本研究では，コミットに含まれる修正情報を分類し，その分類に基づいて，リポジトリを再構築する手法を提案した。

また 2 つの評価実験を行い，被験者を用いた実験では，手法による分類の正しさを確認した．オープンソースのリポジトリを用いた実験では，1 つの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と，ソフトウェアの振る舞いに影響しない修正の両方が混在することを確認し，それらの分離を行った．

今後の課題は以下の 2 点である．

- 修正の分類方法を改善することによる適合率，再現率の向上
- 手法適用前後のリポジトリに対して，それぞれリポジトリマイニングを行い，その分析結果の変化の確認

謝辞

本研究を行うにあたり，日頃より理解あるご指導，ご助言を賜り，多大な励ましを頂きました楠本 真二 教授に深く感謝を申し上げます.

本研究に関して，的確かつ有益なご指摘，およびご助言を頂きました岡野 浩三 准教授に心より感謝を申し上げます.

本研究に多大なるご助言，およびご協力を頂きました井垣 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て，細部にわたる貴重な御指摘と熱心かつ丁寧なご指導を頂きました肥後 芳樹 助教に心より感謝の意を表します.

本研究におけるツールの開発，および全過程におきまして多大なるご助言，ご協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 圭佑氏に深く感謝を申し上げます.

本研究において，実験にご協力頂き，大変重要なデータを提供してくださっ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楠本研究室の 2 名の学生の皆様に厚くお礼申し上げます.

本研究において，様々な形でご助言やご協力，励ましを頂きました楠本研究室の皆様に深く感謝致します.

また，本研究に至るまでの様々な演習や講義，実習等におきましてご指導を頂き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，心より御礼を申し上げます.

参考文献

[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. SS, ソフトウェアサイエンス, Vol. 106, No. 16, pp. 1–6, 2006.

[3] Mina Askari and Ric Holt. Information theoretic evaluation of change prediction models for large-scale software. In *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 126–132. ACM, 2006.

[4] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I Maletic. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 47–53. ACM, 2006.

[5] Huzefa Kagdi, Jonathan I Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pp. 145–154. IEEE, 2007.

[6] A.E. Hassan. The road ahead for mining software repositories. pp. 48–57, 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. 3–3, 2010.

[8] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pp. 190–198. IEEE, 1998.

[9] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 19, No. 2, pp. 77–131, 2007.

[10] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pp. 2–6. sn, 2004.

[11] L.P. Hattori and M. Lanza. On the nature of commits. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pp. 63–71. IEEE, 2008.

[12] A. Hindle, D.M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 99–108. ACM, 2008.

[13] Beat Fluri, Michael Wursch, Martin PInzger, and Harald C Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, Vol. 33, No. 11, pp. 725–743, 2007.

[14] Beat Fluri and Harald C Gall. Classifying change types for qualifying change couplings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pp. 35–45. IEEE, 2006.

[15] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pp. 351–360, 2011.

[16] Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. Refactoring edit history of source code. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 617–620. IEEE, 2012.