特別研究報告

題目

版管理システムにおけるスプリットコミットの調査とその自動検出手法の提案

指導教員

楠本 真二 教授

報告者

有馬 諒

平成 29 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 28 年度 特別研究報告

版管理システムにおけるスプリットコミットの調査と その自動検出手法の提案

有馬 諒

内容梗概

版管理システムとは変更履歴を管理するシステムであり，主にソフトウェア開発におけるソースコードなどの管理に利用されている．版管理システムにおいて各コミットは 1 つのタスクからなるべきだと言われている．しかし，1 つのコミットに複数のタスクが含まれているものや，1 つのタスクが複数のコミットに分割されているものも存在する．本研究では後者のコミットをスプリットコミットと呼ぶ．スプリットコミットはコミット履歴の可読性やリポジトリマイニングの性能に悪影響を与えると考えられる．本研究ではまずどのようなスプリットコミットがリポジトリに含まれているかを目視によって調査し，見つかったスプリットコミットを特徴により 3 つのレベルに分類した．次に目視による調査から得られた特徴から，スプリットコミットの自動検出手法を提案した．この手法では，入力としてコミットの組を受け取ると，それらのコミットの時点でのソースコードからメソッドを頂点としたグラフを構築する．このグラフ上での変更されたメソッド間の距離を用いて入力のコミットの組がスプリットコミットであるかどうかを判定し，その結果を出力する．また，提案手法の評価のために 2 つのオープンソースソフトウェアのリポジトリを対象とした実験を行った．目視による調査を行ったコミットに対して提案手法を適用し調査結果と比較した実験では F 値 0.7，より多くのコミットに対して提案手法を適用し結果を目視によって確認した実験では適合率 0.8 の精度を示し，提案手法がスプリットコミットの検出に有効であることを示した．

主な用語

班管理システム, スプリットコミット，リポジトリマイニング，ソフトウェア開発

目次

1 まえがき 1

2 準備 3

2.1 版管理システム . . . 3

2.2 もつれたコミット . . . 4

2.3 変更履歴を用いたコミットの作成支援 . . . 4

2.4 スプリットコミットの定義 . . . 4

3 調査 6

3.1 調査対象 . . . 6

3.2 調査手順 . . . 6

3.3 結果 . . . 6

4 提案手法 10

4.1 提案手法の手順 . . . 10

4.2 パラメータの決定 . . . 12

4.3 スプリットコミット検出の例 . . . 13

5 実装 15

6 実験 17

6.1 実験方法 . . . 17

6.2 評価指標 . . . 17

6.3 実験結果 . . . 18

7 実験や調査の妥当性 22

7.1 対象ソフトウェアおよび対象コミット . . . 22

7.2 対象言語 . . . 22

7.3 目視での確認 . . . 22

8 あとがき 23

謝辞 24

参考文献 25

図目次

1 ブランチを用いたコミットの状態の例 3
2 スプリットコミットの対象とするコミットの組 5
3 レベル 1 スプリットコミットの例 7
4 レベル 2 スプリットコミットの例 8
5 レベル 3 スプリットコミットの例 9
6 提案手法の概要 10
7 例で用いるソースコード 13
8 ソースコードから生成されたグラフ 14
9 距離を求めるグラフ 16
10 操作後のグラフ 16
11 2 つのコミットでの `TestAll` クラスの変更内容 19
12 提案手法によって検出されたスプリットコミット 20

表目次

1 調査対象 6
2 スプリットコミットの個数と組数に対する割合 7
3 実験 2 での対象区間の詳細 17
4 実験 1 の結果 18
5 #3068126 と#a784206 での変更ファイル 19
6 実験 2 の結果 21
7 検出されたコミットの変更内容 21

1 まえがき

版管理システムとはファイルの変更履歴を管理するシステムである．版管理システムにおいて，変更履歴はコミットと呼ばれる単位ごとに管理されるが，1 つのコミットにどれだけの変更内容を含めるべきかという問題がある．Git 公式ドキュメント [1] ではコミットの単位が論理的に独立した変更になるべきであると述べられており，1 つの機能追加やバグ修正の変更ごとに 1 つのコミットとすべきであると考えられる．そのような変更内容のまとまりを本研究ではタスクと呼ぶ．文献 [2] では単一のタスクごとに 1 つのコミットとするタスクレベルコミットという考えが紹介されている．

複数のタスクを含む大規模なコミットはもつれたコミットと呼ばれ，リポジトリマイニングの性能に悪影響を及ぼすことが明らかになっている [3]．また，このような大規模なコミットを複数のコミットに分割する手法について研究がなされている [3, 4]．しかしこれらの研究では，1 つのタスクが複数のコミットに分割されているものについては扱われていない．本研究ではこのようなコミットをスプリットコミットと呼ぶ．

スプリットコミットの問題点として以下のものが考えられる．

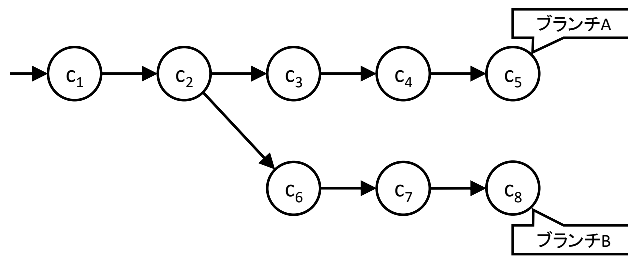
リポジトリマイニングの性能低下：版管理システムのリポジトリに蓄積された変更履歴を解析し，ソフトウェア開発に有用な情報を得る研究が行われている．このような解析のことをリポジトリマイニングという．リポジトリマイニングの例としては，ロジカルカップリング [5, 6, 7, 8]，バグ検知 [9, 10, 11]，関連する変更のクラスタリング [12, 13, 14] などがある．ロジカルカップリングとはソフトウェア中のあるモジュールを変更した際に，そのモジュールと同時に変更されるべき別のモジュールが存在するというモジュール間の論理的な依存関係であり，修正漏れの検知への利用が研究されている [7]．コミットの変更内容からロジカルカップリングを求める際にスプリットコミットが存在すると，本来検出されるべき依存関係が検出されなくなる可能性がある．このようにリポジトリマイニング手法の中にはコミットに含まれる変更内容に依存しているものがあり，そのような手法をスプリットコミットを含むリポジトリに適用した場合，期待する性能が得られない可能性がある．

コミット履歴の可読性の低下：ソースコードの変更内容や変更者，コメントなどがコミットには含まれており，コミット履歴をたどると誰がどのような意図をもって変更を加えたのかを知るのに役に立つ．しかし 1 つのタスクが複数のコミットに分割されている場合，コミット履歴の可読性が低下しコミット履歴を用いた変更内容の理解が難しくなる．

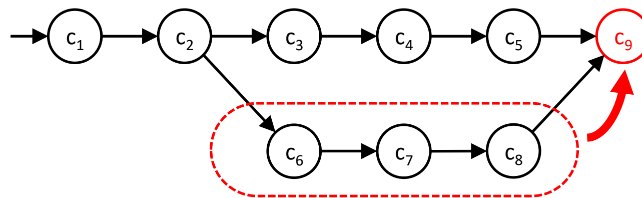
そこで本研究ではまず，実際のリポジトリにどのようなスプリットコミットがどの程度含まれているか目視による調査を行った．調査の結果，多数のスプリットコミットを発見しそれらを特徴によって 3 つに分類した．次に，調査で得られた特徴をもとにした自動検出手法を提案した．提案手法の評価のために行った実験において，目視による調査を行ったコミットに対して提案手法を適用し調査結果と比較

した実験では F 値 0.7，より多くのコミットに対して提案手法を適用し結果を目視によって確認した実験では適合率 0.8 の精度を示し，提案手法がスプリットコミットの検出に有効であることを示した．

以降，2 章では本研究に関連した用語や技術について述べる．3 章では実際にソフトウェアの開発が行われているリポジトリに対して行ったスプリットコミットの調査について述べる．4 章ではメソッドを頂点としたグラフからスプリットコミットを検出する方法を提案する．5 章では提案手法の実装について述べる．6 章では提案手法の評価実験について述べる．7 章では調査や実験の妥当性について述べ，最後に 8 章で本研究のまとめと今後の課題について述べる．



(a) ブランチによって分岐した変更履歴



(b) マージ後のコミットの状態

図 1　ブランチを用いたコミットの状態の例

2　準備

2.1　版管理システム

版管理システムとはファイルの変更履歴を管理するシステムであり，主にソフトウェア開発におけるソースコードなどの管理に利用されている．代表的な版管理システムとしては CVS，Subversion，Git などがある．版管理システムを用いることにより，開発しているソフトウェアを過去の時点に戻すことや，過去の修正を閲覧することが容易になる．

2.1.1　コミット

版管理システムにおいて，変更履歴はコミットと呼ばれる単位ごとに管理されている．各コミットは元となるコミットからの変更内容，日時，変更者，コメントなどから構成される．複数のコミットから構成される蓄積された変更履歴全体はリポジトリと呼ばれる．

2.1.2　ブランチ

多くの版管理システムは，ブランチと呼ばれる複数人で並行して作業を行うための機能を持つ．ブランチを用いると変更履歴を分岐して保存することができるため，主となるブランチに影響を与えずにファイルに変更を加えることができる．図 1(a) にブランチを用いたときのコミットの状態の例を示す．

あるブランチで行った変更を別のブランチに反映することをマージといい，マージを行うとマージコミットと呼ばれるコミットが作成される．マージコミットには，2 つのブランチが分岐した時点からの変更内容がすべて含まれる．図 1(a) の状態からブランチ B をブランチ A にマージした後の状態を図 1(b) に示す．この図においてコミット c_9 はマージコミットであり，このコミットにはコミット c_6〜c_8 の変更内容がすべて含まれる．

2.2　もつれたコミット

本研究では，1 つの独立したコミットとすべき変更内容のまとまりをタスクと呼ぶ．タスクの例として 1 つの機能追加のための変更や，バグ修正のための変更などがある．

複数のタスクを含む大規模なコミットはもつれたコミットと呼ばれている．文献 [15] ではもつれたコミットを変更の目的によって分類し，どのような目的の際にもつれたコミットが生じるかについて調査されている．また，文献 [3] ではもつれたコミットがリポジトリマイニングの性能に悪影響を与えることが示され，もつれたコミットの分割手法としてメソッドの呼び出し関係を用いた方法が提案がされている．文献 [4] では，もつれたコミットの別の分割手法として変数間のデータ依存を用いた手法が提案されている．しかしこれらの研究では，本研究で扱ったスプリットコミットについては述べられていない．

2.3　変更履歴を用いたコミットの作成支援

IDE の機能を用いて開発者の操作をすべて記録することで，ソースコードの変更履歴を版管理システムにおけるコミットよりもさらに細粒度に収集し，このデータをソフトウェアの理解支援に用いる研究が行われている [16, 17, 18, 19]．この細粒度の変更履歴に対して，分割や統合，並び替えなどの操作を行うことで，1 つのタスクからなるコミットの作成支援を行う研究が行われている [20, 21]．これらの研究では IDE の機能を用いて記録された変更履歴を対象としており，版管理システムのコミットを用いた本研究とは異なる．

2.4　スプリットコミットの定義

本研究ではリポジトリ中のコミット c_1，c_2 の組 (c_1, c_2) がスプリットコミットであることを以下のように定義する

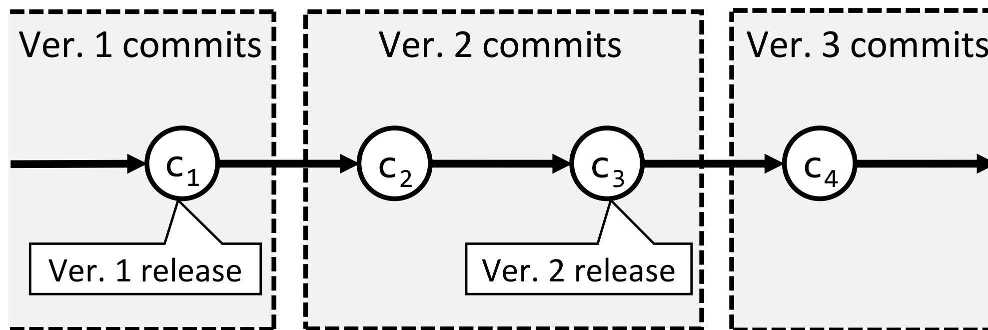


図 2　スプリットコミットの対象とするコミットの組

1. c_1，c_2 が同じバージョンの開発にかかわるコミットである．
2. c_1，c_2 はどちらともマージコミットではない．
3. c_1 の方が c_2 よりも古いコミットである．
4. c_1，c_2 の両方に同じタスクに関する変更が含まれている．

異なるバージョンの開発に関わる 2 つのコミットをまとめて 1 つのコミットとした場合，それらのバージョンのリリースが行われたコミットでのソースコードの内容が変わる場合がある．本研究ではこのような影響を及ぼすコミットの統合は好ましくないと考えたため，(1) の条件を設定した．図 2 において，(c_2, c_3) はスプリットコミットの可能性があるが，(c_1, c_2) はスプリットコミットの対象から除外する．

(2) は，マージコミットに含まれる変更内容は他のいくつかのコミットに含まれる変更内容を統合したものであるため対象から除外した．

(3) は，(c_1, c_2) と (c_2, c_1) のような重複した組が含まれないようにするために設定した．

(4) がスプリットコミットの本質的な定義であり，どのような変更を同じタスクとしてみなすかによってスプリットコミットとなるコミットの組は大きく異なる．そこで，実際にソフトウェアの開発が行われているリポジトリに対して調査を行うことで，どのようなコミットの組をスプリットコミットとするかを検討する．

3　調査

本研究ではまず，実際にソフトウェアの開発が行われているリポジトリにはどのようなスプリットコミットが含まれているかを調べるために調査を行った．

3.1　調査対象

Apache Commons Collections[22]（以降 Collections と呼ぶ）と，Retrofit[23] の 2 つのオープンソースソフトウェアのリポジトリを調査対象とした．調査対象の詳細を表 1 に示す．ここで区間数とはバージョンのリリースコミットによって分割される区間の数を表す．

3.2　調査手順

本研究では，以下手順で調査を行った．

1. 調査対象の期間にあるすべてのコミットについて各コミットに含まれる含まれるタスクを，変更内容やコミットのコメントなどを目視によって確認することで特定した．
2. 特定したタスクをもとに，同じタスクや関連するタスクを含むコミットの組を目視によって見つけ出し，これをスプリットコミットとした．
3. 見つかったスプリットコミットについて以下の項目を確認し，スプリットコミットの特徴を調査した．
    - 変更内容の種類（機能追加，バグ修正，コメント追加など）は何か．
    - 2 つのコミットの変更内容にどのような関連があるか．
    - 2 つのコミットの変更場所はどの程度離れているか．

3.3　結果

調査の結果，Collections リポジトリからは 49 組，Retrofit リポジトリからは 32 個のスプリットコミットを抽出した．また，抽出されたスプリットコミットは，調査によって得られた特徴から以下の 3

表 1　調査対象

	対象期間	コミット数	区間数	組数	追加行数	削除行数
Collections	2001/04/15〜2001/07/15	55	1	1,485	15,401	2,055
Retrofit	2010/09/07〜2011/06/07	45	5	229	7,370	2,303

```
  /**
   * Returns the values for the BeanMap.
   *
   * @return values for the BeanMap. Modifications to this collection
   *         do not alter the underlying BeanMap.
   */
  public Collection values() {
    ArrayList answer = new ArrayList( readMethods.size() );
    for ( Iterator iter = valueIterator(); iter.hasNext(); ) {
      answer.add( iter.next() );
    }
-   return answer;
+   return Collections.unmodifiableList(answer);
  }
```

(a) 2002/03/25 06:53 のコミット

```
  /**
   * Returns the values for the BeanMap.
   *
-  * @return values for the BeanMap. Modifications to this collection
-  *         do not alter the underlying BeanMap.
+  * @return values for the BeanMap.  The returned collection is not
+  *         modifiable.
   */
  public Collection values() {
     ArrayList answer = new ArrayList( readMethods.size() );
     for ( Iterator iter = valueIterator(); iter.hasNext(); ) {
         answer.add( iter.next() );
     }
     return Collections.unmodifiableList(answer);
  }
```

(b) 2002/03/25 07:00 のコミット

図 3　レベル 1 スプリットコミットの例

つのレベルに分類することができた．

レベル 1：スニペットレベル

レベル 2：メソッドレベル

レベル 3：機能レベル

見つかったスプリットコミットのレベル別の個数とその調査した組数に対する割合を表 2 に示す．

以降，3 つのレベルでのスプリットコミットの特徴について述べる．例として，図 3，4，5 に Collections リポジトリで見つかった 3 つのレベルのスプリットコミットを示す．図において + で始まる行はそのコミットで追加された行，-で始まる行はそのコミットで削除された行を表す．

表 2　スプリットコミットの個数と組数に対する割合

	調査組数	レベル 1		レベル 2		レベル 3	
Collections	1,485	13	(0.9%)	21	(1.4%)	49	(3.3%)
Retrofit	229	4	(1.7%)	9	(3.9%)	32	(14.0%)

```
     * Insert an element into queue.
     *
     * @param element the element to be inserted
+    *
+    * @exception ClassCastException if the specified <code>element</code>'s
+    * type prevents it from being compared to other items in the queue to
+    * determine its relative priority.
     */
-    void insert( Comparable element );
+    void insert( Object element );
```

(a) 2002/03/19 13:34 のコミット

```
    /**
     * Insert an element into queue.
     *
     * @param element the element to be inserted
     */
-   public synchronized void insert( final Comparable element )
+   public synchronized void insert( final Object element )
    {
       m_priorityQueue.insert( element );
    }
```

(b) 2002/03/19 22:19 のコミット

図 4　レベル 2 スプリットコミットの例

レベル 1：スニペットレベル

レベル 1 のスプリットコミットは，c_2 での変更内容が c_1 での変更漏れの追加や変更の取り消し，コメントの修正やソースコードの整形のような処理に影響しない変更など c_2 単独のコミットである必要がないものを指す．レベル 1 のスプリットコミットの特徴は，それぞれのコミットでの変更箇所が同じメソッド内など，同一，もしくは非常に近い場所を編集していることである．このレベルのスプリットコミットをまとめて 1 つのコミットとすることでコミット履歴の可読性が向上すると考える．

Collections でのレベル 1 スプリットコミットの例を図 3 に示す．(a) のコミットではメソッドの戻り値を Read-only にする変更が行われているが，それに合わせたコメントの修正は (b) のコミットで行われている．

レベル 2：メソッドレベル

レベル 2 のスプリットコミットは，レベル 1 のスプリットコミットに加えて，c_2 での変更が c_1 での変更に依存しているものを指す．それぞれのコミットで変更されたメソッド間に呼び出し関係や継承関係があることがが特徴である．このレベルのスプリットコミットをまとめることで，ロジカルカップリング [5, 6, 7, 8] などのリポジトリ分析性能の向上が期待できる．

Collections でのレベル 2 スプリットコミットの例を図 4 に示す．(a) のコミットではインターフェー

```
  public abstract class TestList extends TestCollection {
  …
+    public void testListAdd() {
+      List list = makeList();
+      if(tryToAdd(list,"1")) {
+        assert(list.contains("1"));
+        if(tryToAdd(list,"2")) {
+          assert(list.contains("1"));
+          assert(list.contains("2"));
+          if(tryToAdd(list,"3")) {
```

(a) 2001/04/26 09:06 のコミット

```
  public abstract class TestList extends TestCollection {
  …
+   public void testListSetByIndexBoundsChecking2() {
+     List list = makeList();
+     tryToAdd(list,"element");
+     tryToAdd(list,"element2");
+
+     try {
+       list.set(Integer.MIN_VALUE,"a");
+       fail("List.set should throw IndexOutOfBoundsException [Integer.MIN_VALUE]");
```

(b) 2001/05/05 01:34 のコミット

図 5　レベル 3 スプリットコミットの例

ス中のメソッドの引数が `Comparable` から `Object` に変更され，より広い型の値を受け取るように変更されている．このコミットの変更を受けて (b) のコミットではそのインターフェースを実装するクラスのメソッドの引数も `Comparable` から `Object` に変更されている．

レベル 3：機能レベル

レベル 3 のスプリットコミットは，1 つの大きな機能を実現するためのコミットの組を指す．そのため，レベル 3 のスプリットコミットはレベル 1，レベル 2 のスプリットコミットを包含している．レベル 3 のスプリットコミットを用いることで，あるコミットに関連したコミットを探し出すことができる．

Collections でのレベル 3 スプリットコミットの例を図 5 に示す．(a) のコミットでは，`TestList` クラスにリストに対するテストケースを新たに追加している．(b) のコミットでは，それとは別のテストケースを `TestList` クラスに追加している．

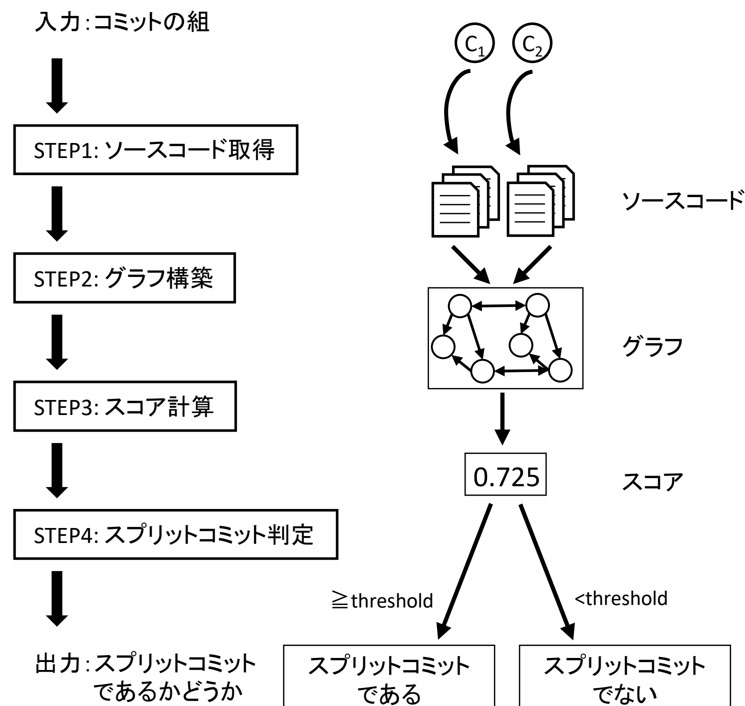


図 6　提案手法の概要

4　提案手法

本章では提案するスプリットコミットの自動検出手法について述べる．

4.1　提案手法の手順

提案手法の入力はコミットの組であり，出力はそのコミットの組がスプリットコミットであるかどうかである．提案手法は以下の STEP からなる．

STEP1: ソースコード取得

STEP2: グラフ構築

STEP3: スコア計算

STEP4: スプリットコミットの判定

提案手法の概要を図 6 に示す．以降，各 STEP の詳細について述べる．

STEP1: ソースコード取得

入力のコミットの組について，それぞれのコミットの時点でのソースコードをリポジトリから取得する．

STEP2: グラフ構築

取得したソースコードからグラフを構築する．

まず，取得したソースコードを解析し，ソースコードに含まれているメソッド，各メソッドの定義されているクラス，メソッド同士の呼び出し関係を取得する．この結果から以下の重み付き有効グラフ $G = (V, E)$ を構築する．

- 頂点は，コミット $c_n \in \{c_1, c_2\}$ とメソッド m の順序対 (c_n, m) とする．これはどちらのコミットのメソッドであるかを区別するためである．本論文では (c_n, m) を m^n と表す．
- 辺を頂点の順序対 (a, b) とする．辺には E_{same}，$E_{calling}$，E_{called}，E_{def} の 4 種類がある．
  - E_{same} は 2 つのコミットで同じメソッド表す頂点間に張る辺である．辺の重みは 1 とする．
  - $E_{calling}$ はメソッド m_1 がメソッド m_2 を呼び出しているとき， 頂点 m_1^c から頂点 m_2^c へ張る辺である．辺の重みは $w_{calling}$ とする．
  - E_{called} はメソッド m_1 がメソッド m_2 に呼び出されているとき，頂点 m_1^c から頂点 m_2^c へ張る辺である．辺の重みは w_{called} とする．メソッド呼び出しの向きによって重さを変化させることができるように 2 種類の辺を定義した．
  - E_{def} はメソッド m_1 とメソッド m_2 が同じクラスで定義されているときに張る辺である．辺の重みは w_{def} とする．

入力として与えられるコミットを (c_1, c_2) としたとき，形式的な定義は以下のとおりである．

$$V = \{m^n | m^n \text{はコミット } c_n \in \{c_1, c_2\} \text{ の時点でのソースコードが持つメソッド } m\} \tag{1}$$

$$E = E_{same} \cup E_{calling} \cup E_{called} \cup E_{def} \tag{2}$$

$$\begin{aligned} E_{same} = \{(a, b) | a = m_1^1 \in V, b = m_2^2 \in V, m_1 = m_2\} \\ \cup \{(b, a) | a = m_1^1 \in V, b = m_2^2 \in V, m_1 = m_2\} \end{aligned} \tag{3}$$

$$E_{calling} = \{(a, b) | a = m_1^n \in V, b = n_1^n \in V, m_1 \text{が } m_2 \text{を呼び出す }\} \tag{4}$$

$$E_{called} = \{(a, b) | a = m_1^n \in V, b = m_2^n \in V, m_1 \text{が } m_2 \text{に呼び出される }\} \tag{5}$$

$$E_{def} = \{(a, b) | a = m_1^n \in V, b = m_2^n) \in V, m_1 \text{と } m_2 \text{は同じクラスで定義 }\} \tag{6}$$

STEP3: スコア計算

STEP2 で構成したグラフ上でのメソッド間の距離をもとにスコアを計算する．

まず，コミット c_1 で編集された各メソッドについて，コミット c_2 で編集されたメソッドのうちグラフ上での距離が一番近いものを探索し，そのメソッドまでの距離を求める．コミット c_1 のメソッドから，コミット c_2 のメソッドへの最短経路には少なくとも 1 つの E_{same} の辺が含まれるため，その距離は 1 以上となる．次に，各メソッドで求めた距離の逆数をとり，その平均を s_1 とする．s_1 は 0 以上 1 以下となる．同様にコミット c_2 で編集された各メソッドについて，コミット c_1 で編集されたメソッドで一番近いものまでの距離を求め，求めた距離の逆数の平均を s_2 とする．最後に s_1 と s_2 の平均と，s_2 のうち高いほうをスコアとする．これは予備調査において，古いほうのコミット c_1 で編集されたメソッドの一部を新しいコミット c_2 においても編集しているスプリットコミットが多く見られたためである．

コミット c_1 で編集されたメソッドの集合を V_1，コミット c_2 で編集されたメソッドの集合を V_2，頂点 a から頂点 b までの距離を $\mathrm{distance}(a, b)$ としたとき，形式的な定義は以下のとおりである．

$$S_1 = \frac{1}{|V_1|} \sum_{m_1 \in V_1} \frac{1}{\operatorname*{argmin}_{m_2 \in V_2} \mathrm{distance}(m_1, m_2)} \tag{7}$$

$$S_2 = \frac{1}{|V_2|} \sum_{m_2 \in V_2} \frac{1}{\operatorname*{argmin}_{m_1 \in V_1} \mathrm{distance}(m_2, m_1)} \tag{8}$$

$$score = \max((S_1 + S_2)/2, S_2) \tag{9}$$

STEP4: スプリットコミットの判定

STEP3 で求めたスコアが閾値（$threshold$）よりも高ければスプリットコミットであると判定し出力する．

4.2 パラメータの決定

提案手法では $threshold$，$w_{calling}$，w_{called}，w_{def} の 4 つのパラメータを決めなければならない．本研究では調査によって得られた特徴から，呼び出し関係 2 回分のメソッド間の距離，もしくは同じクラスのメソッド間の距離をもとに計算したスコアが閾値となるようにパラメータを定める．このことを式で表すと以下のようになる．

$$threshold = \left(2w_{calling} + 1\right)^{-1} \tag{10}$$

$$= \left(2w_{called} + 1\right)^{-1} \tag{11}$$

$$= \left(w_{def} + 1\right)^{-1} \tag{12}$$

```
class A{
  void a(){
+
+    d();
  }
  void b(){
    a();
  }
  void c(){
  }
}
```

```
class B{
  void d(){
+
  }
}
```

(a) コミット c_1 でのソースコード

```
class A{
  void a(){
-
    d();
  }
  void b(){
    a();
  }
  void c(){
  }
}
```

```
class B{
  void d(){

  }
+ void e(){
+ }
}
```

(b) コミット c_2 でのソースコード

図 7　例で用いるソースコード

ここで $threshold = 0.7$ とすると，辺の重みは以下のようになる．

$$(w_{calling}, w_{called}, w_{def}) = \left(\frac{3}{14}, \frac{3}{14}, \frac{3}{7} \right) \tag{13}$$

これらの値を本研究ではパラメータとする．

4.3　スプリットコミット検出の例

スプリットコミット検出の例を示す．

はじめに，入力として与えられたコミットの組 (c_1, c_2) から，それぞれのコミットの時点でのソースコードを取得する．コミット c_1 の時点でのソースコードを図 7(a)，コミット c_2 の時点でのソースコードを図 7(b) に示す．ここで，$+$ で始まる行はそのコミットで追加された行，$-$ で始まる行はそのコミットで削除された行を表す．

次に得られたソースコードからグラフを作成する．

作成したグラフを図 8 に示す．

次に，編集されたメソッド間のグラフ上での距離を求める．

- コミット c_1 で編集されたメソッド a^1 からコミット c_2 で編集されたメソッドの中で一番近いメソッドは a^2 であり，その距離は 1 である
- d^1 からコミット c_2 で編集されたメソッドの中で一番近いメソッドは a^2 であり，その距離は $17/14 = 1.21$ である．
- a^2 からコミット c_1 で編集されたメソッドの中で一番近いメソッドは a^1 であり，その距離は 1 である．
- e^2 からコミット c_1 で編集されたメソッドの中で一番近いメソッドは d^1 であり，その距離は $10/7 = 1.43$ である．

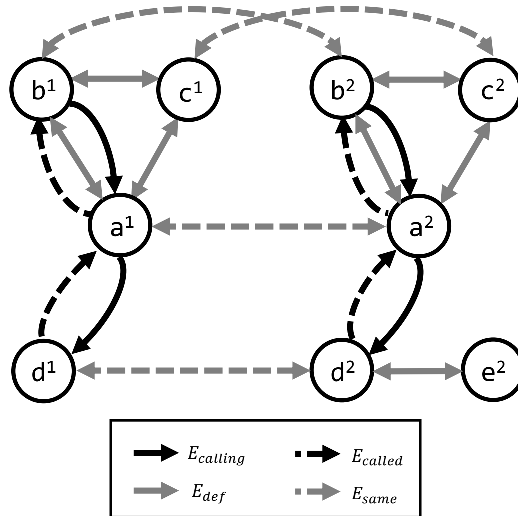


図 8　ソースコードから生成されたグラフ

以上より，$s_1 = 0.91$，$s_2 = 0.85$ である．

最後にスコアを求めると，$score = max((s_1 + s_2)/2, s_1) = 0.88$ となり，このコミットの組のスコアは 0.88 である．よって，この 2 つのコミットの組はスプリットコミットであると提案手法は判定する．

5 実装

本章では，提案手法を実装したツールついて述べる．このツールは Java プロジェクトの Git リポジトリを対象としており，入力として解析対象のリポジトリと，バージョンごとのコミットの区間を与えると，区間ごとのスプリットコミットを出力する．以降，提案手法の各 STEP の実装ついて説明する．

STEP1: ソースコード取得

本ツールではリポジトリから各コミット時点でのソースコードを取得する際に JGit[24] を用いた．JGit とは Git の Java による実装であり，コマンドラインからの利用のほか Java プログラムから直接 Git リポジトリを操作する API も用意されている．本ツールでは，この API を利用し各コミットの時点でのソースコードを取得した．

STEP2: グラフ構築

本ツールではグラフ構築に必要なソースコード解析に，Eclipse Java development tools (JDT) [25] を用いた．JDT とは統合開発環境である Eclipse のプラグイン開発のためのライブラリであり，その機能の 1 つとしてソースコードを解析し抽象構文木の構築やメソッド呼び出し関係の解析などを行う機能がある．このソースコードの解析機能を用いて，ソースコードに含まれているメソッド，各メソッドの定義されているクラス，メソッド同士の呼び出し関係を取得し，これらの情報からグラフを構築した．

STEP3: スコア計算

提案手法では一方のコミットで変更された各メソッドから，他方のコミットで編集されたメソッドで一番近いものを探索し，そのメソッドまでの距離を求める必要がある．本ツールではこの処理に Dijkstra 法を用いた．

例として，図 9 のグラフ上で a^1，b^1 の各頂点から c^2，d^2 の頂点のうち一番近いものを探索することを考える．単純な Dijkstra 法では a^1，b^1 の各頂点を始点とした 2 回の探索が必要になり効率が悪い．そこでグラフに対して以下の操作を行う．

1. すべての辺の向きを逆にする．
2. 頂点 S を追加する．
3. 頂点 S から c^2，d^2 へ重み 0 の辺を張る．

図 10 に操作後のグラフを示す．このグラフ上で S から a^1，b^1 への距離を求めることで，a^1，b^1 の各頂点から c^2，d^2 の頂点のうち一番近い頂点までの距離を求めることができる．この手法により，変更

図 9　距離を求めるグラフ

図 10　操作後のグラフ

されたメソッドが多い場合でも 1 回の探索で各頂点からの距離を求めることができる．

STEP4: スプリットコミットの判定

STEP3 で求めたスコアと閾値を比較し，スコアのほうが高ければスプリットコミットであると出力する．

6 実験

提案手法の精度の評価のために 2 種類の実験を行った．

6.1 実験方法

実験 1

提案手法の精度の評価のために，3 章で行った 2 つのリポジトリに対する調査によって得られた 1,714 組のデータセットに対して提案手法を適用し，スプリットコミットの判定を行った．

実験 2

実験 1 よりも大きいデータセットに対する提案手法の精度の評価のために，調査区間よりも広い区間に対して提案手法を適用し，スプリットコミットの判定を行った．実験 2 での実験データの詳細を表 3 に示す．ここで区間数とはバージョンのリリースコミットによって分割される区間の数を表す．

6.2 評価指標

本研究では評価指標として適合率，再現率，F 値の 3 つを用いた．

適合率 提案手法がスプリットコミットであると判定したもののうち実際にスプリットコミットであったものの割合

再現率 実際のスプリットコミットのうち提案手法がスプリットコミットであると判定したものの割合

F 値 適合率と再現率の調和平均

提案手法がスプリットコミットであると判定したもののうち実際にスプリットコミットであったものの個数を TP，提案手法がスプリットコミットであると判定したもののうちスプリットコミットでなかったものの個数を FP，実際のスプリットコミットのうち，提案手法がスプリットコミットでないと判定したもの個数を FN としたとき，適合率，再現率，F 値は以下のようになる．

表 3 実験 2 での対象区間の詳細

	対象期間	コミット数	区間数	組数	追加行数	削除行数
Collections	2001/04/15～2002/04/03	247	4	17,575	38,269	9,624
Retrofit	2010/09/07～2012/06/14	85	6	1,044	14,578	8,953

$$適合率 = \frac{TP}{TP + FP} \tag{14}$$

$$再現率 = \frac{TP}{TP + TN} \tag{15}$$

$$\mathrm{F}\ 値 = \frac{2\ 適合率 \cdot 再現率}{適合率 + 再現率} \tag{16}$$

;

実験 1 では適合率，再現率，F 値の 3 つを評価指標として用いる．実験 2 では正解データが無いため適合率のみを評価指標として用いる．

6.3 実験結果

実験 1

実験 1 の結果を表 4 に示す．F 値は，Collections では 0.714，Retrofit では 0.745 であり，提案手法はスプリットコミットの検出に有効であるといえる．また，Retrofit では適合率が 1.000 と誤検出が存在しなかったものの，再現率は 0.594 と Collections に比べ低くなっている．このことから，提案手法の最適なパラメータについては検討が必要である．

スプリットコミットでないものをスプリットコミットとして判定してしまったものには，プログラムのエントリーポイントなど，多くのコミットで変更されるメソッドをそれぞれのコミットで変更したものがあった．この例として，Collections リポジトリの 2001/04/15 04:32 のコミット`#004d5e6` と 2001/05/05 11:23 のコミット`#32080b3` がある．この 2 つはスプリットコミットとして検出されたが，`#004d5e6` での変更内容は `ArrayStack` クラスとそのテストの追加，`#32080b3` での変更内容は `ExtendedProperties` クラスのテストの追加であり同じタスクではないため，この組はスプリットコミットではないと判断した．誤検出の原因として，図 11 に 2 つのコミットでの `TestAll` クラスの変更内容を示す．図からわかるように，両方のコミットで `suite` メソッドというすべてのテストのエントリーポイントを修正している．`suite` メソッドは 2 つのコミットでの他の変更メソッドを呼び出しており，提案手法で構築するグラフ上で `suite` メソッドを表す頂点から他の変更メソッドを表す頂点への距

表 4　実験 1 の結果

	適合率	再現率	F 値
Collections	0.714	0.714	0.714
Retrofit	1.000	0.594	0.745

```
    public static Test suite() {
      TestSuite suite = new TestSuite();
+     suite.addTest(TestArrayStack.suite());
      suite.addTest(TestCursorableLinkedList.suite());
      return suite;
    }
```

(a) 2001/04/15 04:32 のコミット#004d5e6

```
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(TestArrayList.suite());
    suite.addTest(TestArrayStack.suite());
    suite.addTest(TestCursorableLinkedList.suite());
    suite.addTest(TestFastArrayList.suite());
    suite.addTest(TestFastArrayList1.suite());
    suite.addTest(TestFastHashMap.suite());
    suite.addTest(TestFastHashMap1.suite());
    suite.addTest(TestFastTreeMap.suite());
    suite.addTest(TestFastTreeMap1.suite());
    suite.addTest(TestHashMap.suite());
    suite.addTest(TestTreeMap.suite());
    suite.addTest(TestCollectionUtils.suite());
+   suite.addTest(TestExtendedProperties.suite());
    return suite;
  }
```

(b) 2001/05/05 11:23 のコミット#32080b3

図 11　2 つのコミットでの TestAll クラスの変更内容

離は短いためスコアが高くなる．この影響で誤検出が発生したと考えられる．解決策として，このように多くのコミットで変更されるメソッドの経路の重みを大きくすることが考えられる．

スプリットコミットであるものをスプリットコミットでないと判定してしまったものには，コミットに複数のタスクが含まれており，そのうち一部のタスクのみが両方のコミットに含まれているコミットがあった．この例として Collections リポジトリの 2001/04/17 07:42 のコミット#3068126 と 2001/04/21 01:54 のコミット#a784206 がある．#3068126 ではいくつかのクラスとそのテストを追加

表 5　#3068126 と#a784206 での変更ファイル

両方のコミットでの変更ファイル	#3068126 での変更ファイル	#a784206 での変更ファイル
TestAll.java	FastArrayList.java	TestArrayStack.java
TestFastArrayList.java	FastHashMap.java	TestCollection.java
TestFastHashMap.java	FastTreeMap.java	TestCursorableLinkedList.java
TestFastTreeMap.java		TestList.java
		TestMap.java
		TestObject.java

```
     raf.setLength(newLength);

     // Calculate the position of the tail end of the data in the ring buffer
     int endOfLastElement = wrapPosition(
         last.position + Element.HEADER_LENGTH + last.length);

     // If the buffer is split, we need to make it contiguous
+    FileChannel channel = raf.getChannel();
     if (endOfLastElement < first.position) {
-      FileChannel channel = raf.getChannel();
       channel.position(fileLength); // destination position
       int count = endOfLastElement - Element.HEADER_LENGTH;
       if (channel.transferTo(HEADER_LENGTH, count, channel) != count) {
         throw new AssertionError("Copied insufficient number of bytes!");
       }
     }

+    // Sync new file length (considered metadata) to storage.
+    channel.force(true);
```

(a) 2001/04/15 04:32 のコミット#004d5e6

```
+    // Set new file length (considered metadata) and sync it to storage.
     raf.setLength(newLength);
+    FileChannel channel = raf.getChannel();
+    channel.force(true);

     // Calculate the position of the tail end of the data in the ring buffer
     int endOfLastElement = wrapPosition(
         last.position + Element.HEADER_LENGTH + last.length);

     // If the buffer is split, we need to make it contiguous
-    FileChannel channel = raf.getChannel();
     if (endOfLastElement < first.position) {
       channel.position(fileLength); // destination position
       int count = endOfLastElement - Element.HEADER_LENGTH;
       if (channel.transferTo(HEADER_LENGTH, count, channel) != count) {
         throw new AssertionError("Copied insufficient number of bytes!");
       }
     }

-    // Sync new file length (considered metadata) to storage.
-    channel.force(true);
```

(b) 2001/05/05 11:23 のコミット#32080b3

図 12　提案手法によって検出されたスプリットコミット

している．#a784206 では#3068126 で追加したテストを変更しているためスプリットコミットである．しかし，表 5 に示すようにそれぞれのコミットでは他にもいくつかのファイルを変更している．提案手法では，2 つのコミットに共通しないタスクが含まれる場合スコアが下がってしまうため，スプリットコミットとして検出できなかったと考えられる．

実験 2

実験 2 の結果を表 6 に示す．適合率は，Collections では 0.822，Retrofit では 0.884 であった．検出されたスプリットコミットの例を以下に示す．

- Retrofit リポジトリの 2010/11/24 08:09 のコミット`#57a57d2` と 2010/11/24 08:23 のコミット`#1eab56a` がスプリットコミットとして検出された．検出されたスプリットコミットの編集箇所を図 12 に示す．このコミットの組では`#57a57d2` で編集したメソッドのソースコードを`#1eab56a` コミットで整形していた．これらのコミットを 1 つにまとめることで，コミット履歴を簡潔にすることができる．
- Collections リポジトリにある 2002/03/02 から 2002/03/20 にかけての 6 つのコミットの中から，13 組がスプリットコミットと判定された．各コミットの変更内容を表 7 に示す．表の示す通り，これらの 6 つのコミットはすべて `ComparatorChain` クラスに関する変更であった．このようにレベル 3 スプリットコミットは，ある機能に関するコミットの検索に有効である．

実験結果から，提案手法はより大きなデータセットに対しても有効であるといえる．

表 6　実験 2 の結果

	検出数	正解数	適合率
Collections	416	342	0.822
Retrofit	95	84	0.884

表 7　検出されたコミットの変更内容

日時	コミット ID	変更内容
2002/03/02 08:29	`#5ae8487`	ComparatorChain クラスを追加
2002/03/02 08:40	`#c55fe26`	メソッドを 2 つ追加
2002/03/02 08:48	`#f247986`	リストを防御的コピーに変更
2002/03/05 04:18	`#5155b93`	コメントとメソッドを追加
2002/03/20 07:25	`#54c23b2`	メソッドとテストケースを追加
2002/03/20 09:25	`#d0d74a4`	比較関数を修正

7 実験や調査の妥当性

本章では，本研究で行った実験や調査の妥当性について述べる．

7.1 対象ソフトウェアおよび対象コミット

本研究では 2 つのオープンソースソフトウェアのリポジトリのみを扱っており，またそのコミットの中でも開発序盤のコミットのみが対象であった．他のソフトウェアのリポジトリや，開発後半のコミットでは本研究と異なる結果となる可能性がある．

7.2 対象言語

本研究で扱ったリポジトリはどちらも Java ソフトウェアである．異なるプログラミング言語のソフトウェアの場合は本研究と異なる結果となる可能性がある．また，提案手法はメソッドの呼び出し関係とメソッドが定義されているクラスの情報を必要とするため，C 言語のようにクラスが無い言語には適用できない．

7.3 目視での確認

調査や実験での正解データは目視により作成したため，分類に誤りがある可能性がある．

8 あとがき

本研究ではまず，2 つのオープンソースソフトウェアのリポジトリに対して目視による調査を行い，1,714 組のコミットの組の中から 81 組のスプリットコミットを発見した．また，発見したスプリットコミットを 3 つのレベルに分類した．次に，目視による調査から得られたスプリットコミットの特徴を利用したスプリットコミットの自動検出手法を提案した．提案手法の評価のために行った実験において，目視による調査を行ったコミットに対して提案手法を適用し調査結果と比較した実験では F 値 0.7，より多くのコミットに対して提案手法を適用し結果を目視によって確認した実験では適合率 0.8 の精度でスプリットコミットを検出することができた．

本研究の貢献は以下の通りである．

- 1 つのタスクが複数のコミットに分割されているコミットであるスプリットコミットがリポジトリに含まれていることを示した．
- スプリットコミットの自動検出手法を提案した．
- 評価実験によって，提案手法がスプリットコミットの検出に有効であることを示した．

今後の課題は以下の通りである．

- より多くのリポジトリに対して実験を実行
- 提案手法を実装した Eclipse プラグインの開発
- スプリットコミットの有無による，リポジトリマイニング性能の差を調査

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励まして頂きました楠本 真二 教授に心より感謝申し上げます.

本研究の全過程において，終始熱心なご指導を頂きました，肥後 芳樹 准教授に深く感謝申し上げます.

本研究に関して，的確で丁寧なご助言を頂きました，柗本 真佑 助教に心より感謝申し上げます.

本報告を行うにあたり，快く相談に乗っていただき，的確なご助言を頂きました，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 年の佐飛 祐介 氏，幸 佑亮 氏に心より感謝申し上げます.

本研究を進めるにあたり，様々な形で励まし，協力を頂いたその他の楠本研究室の皆様に深く感謝申し上げます.

最後に，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，この場を借りて心より御礼申し上げます.

参考文献

[1] Git - Book. https://git-scm.com/book/en/v2/.

[2] Stephen P. Berczuk and Brad Appleton. *Software Configuration Management Patterns*. Addison-Wesley, 2002.

[3] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 121–130, 2013.

[4] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 262–265, 2014.

[5] James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th International Workshop on Program Comprehension*, pp. 44–53, 2003.

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

[7] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 429–445, 2005.

[8] Yusuke Sabi, Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. Reordering results of keyword-based code search for supporting simultaneous code changes. In *Proceedings of the 23rd International Conference on Program Comprehension*, pp. 289–290, 2015.

[9] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 466–480, 2005.

[10] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 200–210, 2012.

[11] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the International Conference on Software Maintenance*, pp. 1–10, 2010.

[12] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical*

Software Engineering, Vol. 11, No. 3, pp. 369–393, 2006.

[13] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 61–72, 2016.

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

[15] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the 5th Working Conference on Mining Software Repositories*, pp. 99–108, 2008.

[16] Romain Robbes and Michele Lanza. A change-based approach to software evolution. In *Electronic Notes in Theoretical Computer Science*, Vol. 166, pp. 93–109. Elsevier Science Direct, 2007.

[17] Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pp. 235–238, 2010.

[18] Lile Hattori, Mircea Lungu, and Michele Lanza. Replaying past changes in multi-developer projects. In *Proceedings of the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pp. 13–22, 2010.

[19] 大森隆行, 丸山勝久. 開発者による編集操作に基づくソースコード変更抽出. 情報処理学会論文誌, Vol. 49, No. 7, pp. 2349–2359, 2008.

[20] 林晋平, 大森隆行, 善明晃由, 丸山勝久, 佐伯元司. ソースコード編集履歴のリファクタリング手法. ソフトウェア工学の基礎 XVIII - 第 18 回ソフトウェア工学の基礎ワークショップ予稿集, pp. 61–70. 2011.

[21] 星野大樹, 林晋平, 佐伯元司. ソースコード編集操作の自動グループ化. ソフトウェア工学の基礎 XX - 第 20 回ソフトウェア工学の基礎ワークショップ予稿集, pp. 107–112, 2013.

[22] GitHub - apache/commons-collections: Mirror of Apache Commons Collections. https://github.com/apache/commons-collections.

[23] GitHub - square/retrofit: Type-safe HTTP client for Android and Java by Square, Inc. https://github.com/square/retrofit.

[24] JGit. http://www.eclipse.org/jgit/.

[25] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.