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.

バージョン管理システムにおける スプリットコミットの調査およびその検出手法の提案

有馬　　諒† 肥後　芳樹† 楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{r-arima,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　バージョン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が，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．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調査し，見つかったスプリットコミットを分割されたタスクの大きさによる 3 つのレベルに分類した．次にスプリットコミットの自動検出手法としてメソッドを頂点としたグラフを構築し，このグラフ上での変更されたメソッド間の距離を利用する手法を提案した．2 つのオープンソースソフトウェアのリポジトリに提案手法を適用した実験では適合率 0.8，F 値 0.7 の性能を示し，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．

キーワード　バージョン管理システム，スプリットコミット

1. ま え が き

バージョン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，ソフトウェア開発においてもソースコードなどの管理に利用されている．代表的なバージョン管理システムとしては CVS，Subversion，Git などがある．これらのバージョン管理システムでは，変更履歴をコミットと呼ばれる単位ごとに管理しており，各コミットは変更内容，日時，変更者，コメントなどから構成される．また，蓄積された変更履歴全体はリポジトリと呼ばれ，複数のコミットから構成される．バージョン管理システムを用いることによって，あるコミットが作成された時点でのファイルの内容やそのコミットでの変更内容を簡単に取り出すことができる．

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がリポジトリ分析の性能に悪影響を及ぼすことが明らかになっており [3]，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]．また，文献 [5] では大規模な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大規模なコミットが生じるかについて調査されている．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．

リポジトリ分析性能の低下：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場合がある．そのようなモジュール間の論理的な結びつきはロジカルカップリング [6] [7] と呼ばれており，修正漏れの検知への利用が研究されている [8]．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結びつきが検出されなくなる可能性がある．

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．

本研究の目的は以下の 2 つである．

- 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調査する．
- スプリットコミットの検出手法を提案する．

以降，2 章ではスプリットコミットの予備調査について述べる．3 章では予備調査の結果に基づくスプリットコミットの 3 つの分類について述べる．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 章では提案手法の評価実験について述べる．6 章では妥当性の脅威について述べ，最後に 7 章で本研究のまとめと今後の課題について述べる．

2. 予 備 調 査

まず初めに，実際のソフトウェアのリポジトリ中にはどのようなスプリットコミットが含まれているかについて目視による調査を行った．

2.1 調 査 方 法

Apache Commons Collections [9] と，Retrofit [10] 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．

まず調査対象の期間にあるコミットについて，各コミットに含まれるタスクを変更内容やコミットのコメントなどから特定した．この結果を用いて同じタスクや関連するタスクを含むコミットの組を見つけ出し，これをスプリットコミットとした．見つかったスプリットコミットについてそれぞれのコミットでの変更の場所や内容，種類（機能追加，バグ修正，コメント追加など）にどのような関係があるか調査した．2 つのリポジトリの調査には約 2 週間を要した．

2.2 結　　果

予備調査の結果，多くのスプリットコミットを 2 つのリポジトリから発見した．発見されたスプリットコミットは以下の特徴によって分類することができた．

- それぞれのコミットで同じメソッドを変更しているスプリットコミット．
- それぞれのコミットで呼び出し関係や継承関係にあるメソッドを変更しているスプリットコミット．
- それぞれのコミットで同じクラスで定義されたメソッドを変更しているスプリットコミット．

3. スプリットコミットの分類

本研究では 2 つのコミットの組 (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

（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．

表 1 調 査 対 象

	対象期間	コミット数	追加行数	削除行数
Apache	2001/04/15 ～2001/07/15	55	15,401	2,055
Retrofit	2010/09/07 ～2011/06/07	45	7,370	2,303

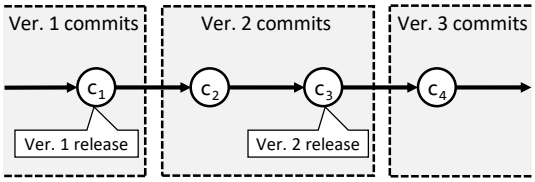

図 1 スプリットコミットの対象とするコミットの組

異なるバージョンの開発にかかわる 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 1 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) の条件において，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，予備調査で見つかったスプリットコミットの特徴をもとに決定した以下の 3 つのレベルでスプリットコミットを分類する．

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

予備調査で見つかったスプリットコミットをこの 3 つのレベルに分類した結果を表 2 に示す．

以降，3 つのレベルでのスプリットコミットの違いについて述べる．例として Apache Commons Collections [9] を用いた．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．

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など c_2 単独のコミットである必要がないものを指す．特徴としては，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．我々は，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．

Apache Commons Collections でのレベル 1 スプリットコ

表 2 スプリットコミットの調査結果

	Level-1		Level-2		Level-3	
Apache	13	(0.9%)	21	(1.4%)	49	(3.3%)
Retrofit	4	(1.7%)	9	(3.9%)	32	(14.0%)

```
  /**
   * 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 2　レベル 1 スプリットコミットの例

ミットの例を図 2 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 [6] [7] などのリポジトリ分析性能の向上が期待できる．

Apache Commons Collections でのレベル 2 スプリットコミットの例を図 3 に示す．(a) のコミットではインターフェー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レベル 2 のスプリットコミットに加えて，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．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．

```
     * 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 3　レベル 2 スプリットコミットの例

```
+    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 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 4　レベル 3 スプリットコミットの例

Apache Commons Collections でのレベル 3 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，`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．

4. スプリットコミット検出手法の提案

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 5 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

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．これには Java による Git の実装である JGit [11] を用いた．

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．現在の実装では Java のソースコードに限定し，Eclipse Java development tools (JDT) [12] をソースコードの解析に用いた．JDT によるソースコードの解析で，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同士に張る辺である．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．

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

$$E = E_{same} \cup E_{calling} \cup E_{called} \cup E_{def} \quad (2)$$

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

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

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

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

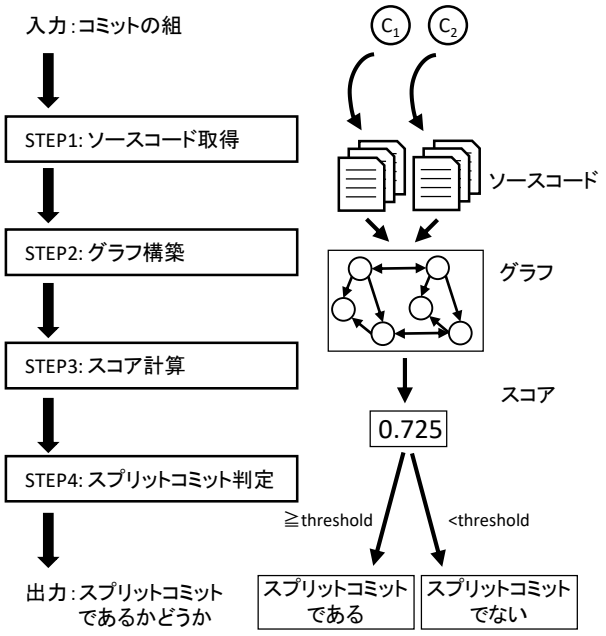

図 5　提案手法の概要

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．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．本研究では一番近いメソッドの探索に Dijkstra 法を用いた．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．

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

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

$$score = \max((S_1 + S_2)/2, S_2) \quad (9)$$

STEP4: スプリットコミットの判定

STEP3 で求めたスコアが閾値よりも高ければスプリットコ

```
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 6　例で用いるソースコード

ミットであると判定し出力する．

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．ここではパラメータとして以下の値を用いた．

$$threshold = 0.7 \tag{10}$$

$$(w_{calling}, w_{called}, w_{def}) = \left(\frac{3}{14}, \frac{3}{14}, \frac{3}{7}\right) \tag{11}$$

これらの値は以下の式を満たすように定めた．

$$threshold = \left(2w_{calling} + 1\right)^{-1} \tag{12}$$

$$= \left(2w_{called} + 1\right)^{-1} \tag{13}$$

$$= \left(w_{def} + 1\right)^{-1} \tag{14}$$

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 6(a)，コミット c_2 の時点でのソースコードを図 6(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 7 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．

以上より，$s_1 = 0.91$，$s_2 = 0.85$ である．

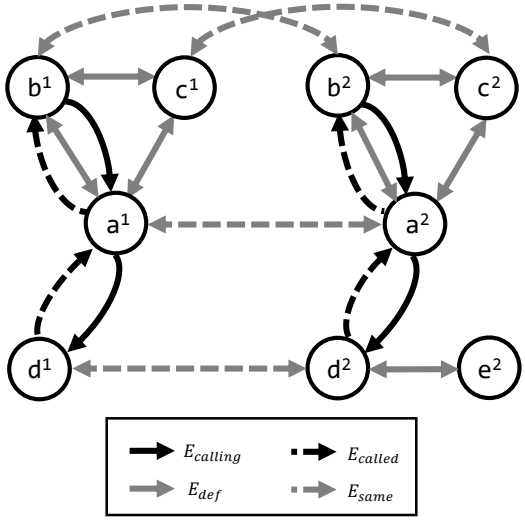

図 7　ソースコードから生成されたグラフ

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　　験

5.1 実験方法

提案手法の評価のため，以下の 2 種類の実験を行った．

実　験　1

2 章で行った調査によって得られた 2 つのリポジトリのデータセット (100 コミット, 1,714 組, うち 81 組のスプリットコミット) 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．この判定結果から，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である適合率，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である再現率，およびこれら 2 つの調和平均である F 値を評価指標として求めた．

実　験　2

調査区間よりも広い区間（331 コミット，18,619 組）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．この判定結果のうち，提案手法がスプリットコミットであると判定したコミットの組が本当にスプリットコミットであるか目視で確認し，適合率を評価指標として求めた．

5.2 実験結果

実験 1 の結果を表 3 に示す．F 値は，Apache では 0.714，

表 3　データセット 1 での実験結果

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

Retrofit では 0.745 であった．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．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．

実験 2 の結果を表 4 に示す．適合率は，Apache では 0.822，Retrofit では 0.884 であった．よって，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．

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．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．
- Apache Commons 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 5 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．

6. 妥当性の脅威

対象コミット：本研究ではリポジトリ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．開発後半のコミットでは本研究と違う結果となる可能性がある．

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，データを作成した者の主観に依存している．

7. 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．提案手法を用いた実験では F 値 0.7，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，2 つのリポジトリを用いて評価を行った．

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情報を用いたコミットの自動クラスタリング手法の検討

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：JP25220003）の助成を得て行われた．

表 4 データセット 2 での実験結果

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

表 5 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

文 献

[1] “Git - Book”. https://git-scm.com/book/en/v2/
[2] S. Berczuk and B. Appleton, Software Configuration Management Patterns, Addison-Wesley, 2002.
[3] K. Herzig and A. Zeller, “The impact of tangled code changes,” Proceedings of the 10th Working Conference on Mining Software Repositories, pp.121–130, May 2013.
[4] 切貫弘之，堀田圭佑，肥後芳樹，楠本真二，“ソースコード中の変数間のデータ依存関係を用いたコミットの分割，” 電子情報通信学会技術研究報告，vol.113，no.269，pp.67–72，Oct. 2013.
[5] A. Hindle, D. M. German, and R. Holt, “What do large commits tell us?: A taxonomical study of large commits,” Proceedings of the 5th Working Conference on Mining Software Repositories, pp.99–108, May 2008.
[6] J. M. Bieman, A. A. Andrews, and H. J. Yang, “Understanding change-proneness in OO software through visualization,” Proceedings of the 11th IEEE International Workshop on Program Comprehension, pp.44–53, May 2003.
[7] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” Proceedings of the International Conference on Software Maintenance, pp.190–198, Bethesda, Maryland, USA, Nov. 1998.
[8] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” IEEE Transactions on Software Engineering, vol.31(6), pp.429–445, June 2005.
[9] “GitHub - apache/commons-collections: Mirror of Apache Commons Collections”. https://github.com/apache/commons-collections
[10] “GitHub - square/retrofit: Type-safe HTTP client for Android and Java by Square, Inc.”. https://github.com/square/retrofit
[11] “JGit”. http://www.eclipse.org/jgit/
[12] “Eclipse Java development tools (JDT)”. http://www.eclipse.org/jdt/