# A Study on Inappropriately Partitioned Commits
# — How Much and What Kinds of IP Commits in Java Projects? —

Ryo Arima
Osaka University
Suita, Osaka, Japan
r-arima@ist.osaka-u.ac.jp

Yoshiki Higo
Osaka University
Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

## ABSTRACT

When we use code repositories, each commit should include code changes for only a single task and code changes for a single task should not be scattered over multiple commits. There are many studies on the former violation–often referred to as tangled commits–but the latter violation has been out of scope for MSR research. In this paper, we firstly investigate how much and what kinds of inappropriately partitioned commits in Java projects. Then, we propose a simple technique to detect such commits automatically. We also report evaluation results of the proposed technique.

## CCS CONCEPTS

• **Software and its engineering** → *Software version control*;

## KEYWORDS

Inappropriately partitioned commits, tangled commits, logical coupling

## 1 INTRODUCTION

When we use version control systems, we should pay attention to which changes are committed to the code repository together. The official document of Git says that changes included in every commit should be independent of each other [3]. This means that a semantically cohesive set of code changes such as a bug-fix or a functional addition should be a single commit. In this research, we call such a semantically cohesive set of code changes *task*. A commit including only a single task is called *task level commit* [1]. Previous research studies revealed that commits including multiple tasks, which are so-called *tangled commits*, have negative impacts on the performance of repository analyses [5]. There are also some research studies that proposed techniques to split tangled commits into different appropriate commits [5, 6].

(a) 25th March, 2002 at 06:53 commit.



(b) 25th March, 2002 at 07:00 commit.

**Figure 1: Example of IP commits**

However, at present, there is no research on a set of commits where a task is segmentalized. In this research, we call such small commits *inappropriately partitioned commits* (in short, *IP commits*). Figure 1 shows IP commits in open source software, Apache Commons Collections.[1] In the earlier commit, the return value of the method became unmodifiable. In the latter commit, the Javadoc comment attached to the method was changed. The presence of such IP commits can cause the following issues.

**Degrading Performance of Repository Analysis**
Evolutionary coupling [2, 4] is a kind of logical connection between two modules in source code. It means the two modules tend to be changed together. Evolutionary couplings are utilized in many kinds of research such as suggesting refactorings[4] and preventing change overlooking[7]. The presence of IP commits makes more difficult to detect evolutionary couplings from commit history.

**Degrading Understandability of Past Commits**
Commits include other information such as developer's name and commit message in addition to changed code. Such information is very useful to know the intent of the developer–why he/she made the changes. However, if the

---
[1] https://github.com/apache/commons-collections

```
   * 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) 19th March, 2002 at 13:34 commit.

```
  /**
   * 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) 19th March, 2002 at 22:19 commit.

Figure 2: Type-2 IP commits

changes for a task are scattered over multiple commits, it becomes more difficult to collect the information and understand the intent.

The main contributions of this paper are summarized as follows:

- investigating how many and what kinds of IP commits are included in code repositories, and
- developing a new technique to detect IP commits automatically.

## 2 INVESTIGATION FOR IP COMMITS

As a first step of this research, we manually investigated how many and what kinds of IP commits existed in development histories. In this investigation, we manually checked whether every pair of given two commits are IP commits or not. Please note that, at the time of this investigation, we had not had any clear definition of IP commits such as described in Section 3 yet. In the investigation, we carefully checked whether given two commits have any semantical cohesiveness or not.

The targets of this investigation are two open source software.

**Apache Commons Collections:** this software is a library providing various data structures. Our targets are 1,485 commit pairs before Ver.1 was released.

**Retrofit:** this software is a library providing HTTP connections.[2] Our targets are 229 commit pairs before Ver. 0.5 was released.

As a result, we regarded 49 and 32 commit pairs as IP commits for *Collections* and *Retrofit*, respectively. In the 81 IP commits, 54 and 24 of them have remakeable features, respectively. Each of the 54 IP commits consisted of two commits made by the identical developer. In addition, each of the 18 IP commits consisted of two commits made within a day. In this explanation, $c_1$ and $c_2$ are two different commits in the repository. $c_1$ is older than $c_2$. Such commit pairs can be classified as follows:

[2]https://github.com/square/retrofit

### Table 1: Detected IP commits

| Software | Type-1 | | Type-2 | | Type-3 | |
|----------|--------|--------|--------|--------|--------|--------|
| Apache | 13 | (0.9%) | 8 | (0.5%) | 28 | (1.9%) |
| Retrofit | 4 | (1.7%) | 5 | (2.1%) | 23 | (9.2%) |

```
+  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) 26th April, 2001 at commit.

```
+  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_V
```

(b) 5th May, 2001, at 01:34 commit.

Figure 3: Type-3 IP commits

**Type-1:** Changes in the $c_2$ *corrects* changes in the $c_1$.
**Type-2:** Changes in the $c_2$ *depends* on changes in the $c_1$.
**Type-3:** Changes in the $c_2$ *collaborates* with changes in the $c_1$.

In the remainder of this section, we describe each of the above types in detail.

### Type-1: Corrective IP Commits

Correcting misspellings, adding extra changes for overlooked code fragments or reverting previous changes were found in the investigation. The IP commit in Figure 1 is a Type-1 commit pair. The developers also found that $c_1$ and $c_2$ had changed code in the same method in the great majority of cases. The developers were not able to find any reason why $c_2$ is a different commit from $c_1$. By merging commits of Type-1 commit pairs as a single commit, understandability of commit histories will get improved.

### Type-2: Dependent IP Commits

In Type-2 commit pairs, the changes in $c_2$ depend on the changes in $c_1$. Figure 2 shows a pair of Type-2 commits. In the earlier commit, the parameter type of method insert was changed to Object from Comparable. In the latter commit, the overriding method was changed for accepting wider types. In most cases of Type-2 commit pairs, two methods having a calling or an inheritance relationship are changed in $c_1$ and $c_2$. If two commits of a Type-2 pair is merged as a single commit, the performance of repository analyses such as finding evolutionary couplings is improved.

### Type-3: Collaborative IP Commits

Two commits in a Type-3 pair is a set of changes to the same functionality. Figure 3 shows a Type-3 pair of commits. In the earlier commit, a test case was added to test a function, and in the latter commit, another test case was added to the same function. In most of the cases, two commits of a Type-3 pair changed the code in the same class. Knowledge about $c1$ and $c2$ are commits for the same task is helpful when developers examine the change history of the functionality.

## 3 DETECTING IP COMMIT AUTOMATICALLY

Herein, we propose a technique to detect IP commits automatically. The proposed technique takes two commits as input and then it decides whether the two commits are IP commits or not. The proposed technique (1) constructs a graph from given two commits
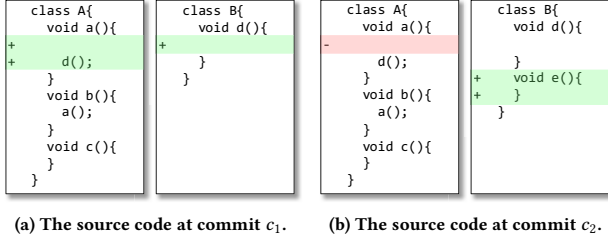
(a) The source code at commit $c_1$.    (b) The source code at commit $c_2$.

**Figure 4: Source code for the example**

and then (2) calculates likelihood of a value for deciding whether the two commits are IP commits or not. In the remainder of this section, we describe (1) and (2) in detail.

## 3.1 Constructing a Graph

Currently, our technique handles Java source code. By analyzing the source code, we extract three kinds of relationships:

- methods and their owner classes,
- calling relationship between methods, and
- overriding relationship between methods.

Then, a following weighted directed graph $G = (V, E)$ is constructed from results of the extracted relationships.

- Each vertex is represented by $m^c$, which means method $m$ at commit $c$.
- Each edge is an ordered pair of vertices $(a, b)$. There are four types of edges, $E_{same}$, $E_{calling}$, $E_{called}$ and $E_{def}$.

The four types of edges are as follows.

- Each edge in $E_{same}$ means that its both ends are the same methods in different commits. The weight of this type edges are one.
- Each edge in $E_{calling}$ means that the method represented with $a$ calls the method represented by $b$. The weight of this type edges are defined $w_{calling}$.
- Each edge in $E_{called}$ means that the method represented with $a$ is called by the method represented by $b$. The weight of this type edges are defined $w_{called}$. We need both $E_{calling}$ and $E_{called}$ because we use different weighted values for the two types of edges.
- Each edge in $E_{def}$ means that two methods of its both ends are defined in the same class. The weight of this type edges are defined $w_{def}$.

## 3.2 Calculating Likelihood

A numerical value is calculated in this process. The numerical value represents how far methods changed at the two commits are on the graph. First, a value from each method changed at commit $x$ to the nearest method in the methods changed at commit $y$ is calculated. This value never become less than one because the shortest path between different-commit vertices includes at least an edge of type $E_{same}$. $s_x$ is an average of the reciprocals of the obtained values. $s_x$ is between zero and one. The values are calculated on the Dijkstra's algorithm.

Similarly, the distance value from each method changed at commit $y$ to the nearest method in the methods changed at commit $c_x$ is calculated. $s_y$ is an average of the reciprocals of the obtained distance values. Please note that a value of the path from method
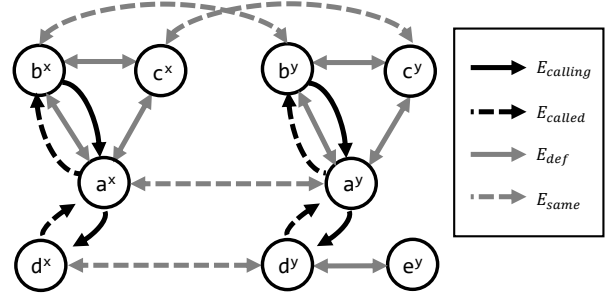


**Figure 5: The graph constructed from source code**

$m_1$ to method $m_2$ can be different from a value of the path from method $m_2$ from $m_1$ because we use different weighted values for $E_{calling}$ and $E_{called}$.

Finally, a score is the higher of $s_y$ and the average of $s_x$ and $s_y$. We do not use $s_x$ and $s_y$ symmetrically because we found that a part of the methods changed at an older commit $c_x$ was also modified at a newer commit $c_y$ in many IP commits.

Let $V_x$ be a set of methods changed at commit $x$, $V_y$ be a set of methods changed at commit $y$ and path$(a, b)$ be the path value from vertex $a$ to vertex $b$ in the constructed graph, then formal definitions are as follows:

$$s_x = \frac{1}{|V_x|} \sum_{v_1 \in V_x} \frac{1}{\underset{v_2 \in V_y}{\operatorname{argmin}} \text{path}(v_1, v_2)} \tag{1}$$

$$s_y = \frac{1}{|V_y|} \sum_{v_2 \in V_y} \frac{1}{\underset{v_1 \in V_x}{\operatorname{argmin}} \text{path}(v_2, v_1)} \tag{2}$$

$$score = \max(\frac{1}{2}(s_x + s_y), s_y) \tag{3}$$

## 3.3 Example

Herein, we show an example of *score* calculation. The following parameters are used in this example.

$$threshold = 0.7 \tag{4}$$

$$(w_{calling}, w_{called}, w_{def}) = \left( \frac{3}{14}, \frac{3}{14}, \frac{3}{7} \right) \tag{5}$$

Those weight values satisfy the following formulas.

$$threshold = \left(2w_{calling} + 1\right)^{-1} \tag{6}$$

$$= \left(2w_{called} + 1\right)^{-1} \tag{7}$$

$$= \left(w_{def} + 1\right)^{-1} \tag{8}$$

Figure 4 shows the source code at two commits $x$ and $y$. The lines starting with "+" and "-" indicate appended and deleted lines at the commit, respectively. Figure 5 shows the graph constructed from the source code.

Distance values between the changed methods on the graph are calculated as follows.

- The nearest method from $a^x$ of methods changed at commit $y$ is $a^y$, and the path value from $a^x$ to $a^y$ is 1.
- The nearest method from $d^x$ of methods changed at commit $y$ is $a^y$, and the path value from $d^x$ to $a^y$ is 17/14 = 1.21.
- The nearest method from $a^y$ of methods changed at commit $x$ is $a^x$, and the path value from $a^y$ to $a^x$ is 1.

- The nearest method from $e^y$ of methods changed at commit $c_y$ is $d^x$, and the path value from $e^y$ to $d^x$ is $10/7 = 1.43$.

Therefore, $s_x = \frac{1}{2}(1 + 14/17) = 0.91$, $s_y = \frac{1}{2}(1 + 7/10) = 0.85$. Finally, a score is calculated: $score = max((s_x + s_y)/2, s_y) = 0.88$. Because the score is higher than the threshold, the input pair of commits is regarded as a pair of IP commits.

## 4 EVALUATION

To evaluate our proposed technique, we conducted following two experiments.

**Experiment 1:** we applied our proposed technique to the golden set obtained by the investigation of the two repositories in Section 2. As evaluation measures of this experiment, precision, recall and F-measure were calculated from the output of the proposed technique. Precision is defined as a proportion of real IP commits in the pairs of IP commits detected by our proposed technique. Recall is defined as a proportion of IP commits detected by our proposed technique in real IP commits in the dataset. F-measure is defined as a harmonic mean of precision and recall.

**Experiment 2:** we applied our proposed technique to a larger dataset than the golden set. The dataset consisted of 18,619 pairs of commits in Apache Commons Collections and Retrofit repositories. Then, we checked manually whether each commit pair is real IP commit or not and calculated precision.

### 4.1 Results

Table 2(a) shows the results of Experiment 1. F-measure of Apache was 0.714 and F-measure of Retrofit was 0.745. In cases where our proposed technique classified detected IP commits by mistake, there were pairs of commits that methods changed at many commits were changed at both commits. For example, entry points (files including *main* methods) were changed at many commits. In cases where our proposed technique was not able to detect IP commits, the commits included multiple tasks and only a part of the tasks was included in both the commits.

Table 2(b) shows the results of Experiment 2. Precision of Apache was 0.822 and precision of Retrofit was 0.884. Our proposed technique is effective even for the larger dataset.

Examples of detected IP commits are shown in the following.

- A pair of commits on 23rd November at 08:09, and 08:23 in the Retrofit repository was detected as IP commits. In this pair of commits, source code of methods changed at the 08:09 commit were cleaned up at the 08:23 commit. Merging these commits, commit history became simpler.

**Table 2: Precision, Recall, and F-measure**

(a) Experiment 1

| Software | Precision | Recall | F-measure |
|----------|-----------|--------|-----------|
| Apache   | 0.714     | 0.714  | 0.714     |
| Retrofit | 1.000     | 0.594  | 0.745     |

(b) Experiment 2

| Software | Detected | Correct | Precision |
|----------|----------|---------|-----------|
| Apache   | 416      | 342     | 0.822     |
| Retforit | 95       | 84      | 0.884     |

- In the six commits from 3rd March to 6 in the Apache Commons Collections repository, 13 pairs of commits obtained by these six commits were classified as IP commits. Because all these commits included the changes on *ComparatorChain* class as shown in Table 3, these IP commits were Type-3. As indicated by this example, Type-3 IP commits are effective to search commits related to a function.

## 5 THREAT TO VALIDITY

Our research has the following threats to validity.

**Target commits:** in the investigation, the authors used commits of an early date in the development. If we used later commits, we might have obtained different results.

**Manual working:** in the experiment, the authors manually constructed oracle data to measure recall and precision. Of course, we did that very carefully, but we cannot deny that some our subjective views affect the results of the manual working.

## 6 CONCLUSION

In this research, firstly we manually checked two source code repositories of open source software. As a result of checking 1,174 commit pairs, we found that 81 pairs of inappropriately partitioned commits. We also classified such pairs into three categories. Then, we proposed a technique to find such commit pairs automatically by using the heuristics that we had derived from the manual checking. In our evaluation, F-measure of the proposed technique on two open source software was 0.714 and 0745, respectively.

The main contributions of this paper are as follows.

- We confirmed that IP commits, which are inappropriately scattered changes in multiple commits, surely exist in source code repositories.
- We proposed a new technique to detect IP commit automatically and we evaluated it with two open source software.

The followings are directions our future research.

- We are going to conduct more experiments on a variety of software.
- We are going to improve the proposed technique by using metadata of commits such as developers, timestamps and messages.
- We are going to develop an Eclipse plugin that is an implementation of the proposed technique.

**Table 3: Changes at the detected IP commits**

| Date | Change description |
|------|--------------------|
| at 08:29 on 2nd March | adding ComparatorChain class. |
| at 08:40 on 2nd March | adding two methods. |
| at 08:48 on 2nd March | changing to a defensive list copy. |
| at 04:18 on 5th March | adding comments and methods. |
| at 07:25 on 20th March | adding methods and test cases. |
| at 09:25 on 20th March | changing a comparator method. |

## REFERENCES

[1] Stephen P. Berczuk and Brad Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[2] James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. 2003. Understanding Change-proneness in OO Software through Visualization. In *Proceedings of 11th International Workshop on Program Comprehension.* 44–53.

[3] Scott Chacon and Ben Straub. 2014. *Pro Git* (2nd ed.). Apress, Berkely, CA, USA.

[4] Harald Gall, Karin Hajek, and Mehdi Jazayeri. 1998. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance.* 190–198.

[5] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories.* 121–130.

[6] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! Are You Committing Tangled Changes?. In *Proceedings of the 22nd International Conference on Program Comprehension.* 262–265.

[7] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. 2005. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.