Graph-Based Approach for Detecting Impure Refactoring from Version Commits

Shogo Tsutsumi[†], Eunjong Choi[‡], Norihiro Yoshida[*], and Katsuro Inoue[†]

[†]Graduate School of Information Science and Technology, Osaka University, Japan
[‡]Graduate School of Information Science, Nara Institute of Science and Technology, Japan
[*]Graduate School of Information Science, Nagoya University, Japan
[†]{s-tutumi,inoue}@ist.osaka-u.ac.jp, [‡]choi@is.naist.ac.jp, [*]yoshida@ertl.jp

ABSTRACT

Impure refactoring is defined as a refactoring operation that was saved together with non-refactoring changes or several refactoring operations were saved at the same location stored in source code repositories. Many of existing approaches are not correctly viable for detecting impure refactoring. To mitigate this problem, we propose an approach that detects impure refactoring from commits stored in the repositories using a graph search algorithm. In case study, we applied our approach to two actual classes in *Apache Xerces* project and confirmed the feasibility of the approach.

CCS Concepts

•Software and its engineering → Maintaining software;

Keywords

impure refactoring; graph search; refactoring detection

1. INTRODUCTION

Refactoring histories provide insights to not only practitioners but also researchers. In particular, there is an increasing interest in the relationship between refactoring operations and source code quality [1, 11]. So far, a number of approaches that detect refactoring operations between two commits have been proposed [2, 8, 9, 10]. These approaches usually detect each refactoring operation by analyzing source code changes stored at commits in source code repositories.

Many of the existing approaches are not correctly viable when a refactoring operation was saved together with non-refactoring changes or several refactoring operations were saved at the same location. These refactoring operations are called **impure refactoring** [3]. To alleviate this problem, the approaches detecting several refactoring operations conducted at the same location have been proposed [5, 7].

IWoR'16, September 4, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-4509-5/16/09...$15.00
http://dx.doi.org/10.1145/2975945.2975949

```
public TarEntry(File file) {
  this();
  this.file = file;
  String fileName = file.getPath();
    …
  this.linkName = new StringBuffer("");
  this.name = new StringBuffer(fileName);
    …
  if (file.isDirectory()) {
    this.mode = DEFAULT_DIR_MODE;
    this.linkFlag = LF_DIR;

    if (this.name.charAt(this.name.length() - 1) != '/') {
      this.name.append("/");
    }
  } else {
    …
  }
    …
```

(a) Previous commit(755230)

```
public TarEntry(File file) {
  this();
  this.file = file;
  String fileName = normalizeFileName(file.getPath());
  this.linkName = new StringBuffer("");
  this.name = new StringBuffer(fileName);
  if (file.isDirectory()) {
    this.mode = DEFAULT_DIR_MODE;
    this.linkFlag = LF_DIR;
    int nameLength = name.length();
    if (nameLength == 0 || name.charAt(nameLength - 1) != '/'){
      this.name.append("/");
    }
  } else {
    …
  }
  …
}
  …
private static String normalizeFileName(String fileName) {
  …
```

(b) Revised commit(755231)

Figure 1: Example of impure refactoring applied to a constructor of a class `org.apache.tools.tar.TarEntry` in the Apache Ant SVN repository.

For instance, Mahouachi *et al.* proposed a search-based approach that detects a sequence of refactoring operations using structural metrics [7]. Hayashi *et al.* proposed an approach that detects multiple refactoring operations using a graph search technique [5]. This approach considers a commit of a program as a state and detects the refactoring operations by searching appropriate path between two states (i.e., previous and revised commits) and estimating the heuristic distances.

However, to our knowledge, there is no approach that detects refactoring operations that are saved together with non-refactoring changes in the same version. For example, when '*Rename Method*' refactoring and 'the addition of error handling' have been applied to the same method and then saved at the same commit, existing tools only detect '*Rename Method*' operation and fail to detect non-refactoring changes, the addition of error handling. An example of impure refactoring stored in the *Apache Ant* Subversion (SVN) repository is described in Figure 1[1]. Note that we changed layouts of this example to save space. Manual investigation of this impure refactoring code fragments reveals that developers performed '*Extract Method*' refactoring (underlined) along with a non-refactoring change (bold) at the same method in a class named as `TarEntry` at the same commit. However, these non-refactoring changes cannot be detected by existing refactoring detection approaches because existing approaches are only able to detect refactoring operations.

To tackle this problem, we propose an approach that detects impure refactoring by extending Hayashi and his colleagues' work. The fundamental difference is that their work only detects several refactoring operations whereas our approach detects not only several refactoring operations but also refactoring operations and non-refactoring changes that were saved at the same location at the same revision. Our approach, at first, detects refactoring operations using a graph search algorithm and test cases from previous and revised versions. Then, it identifies non-refactoring changes based on structural differences between the version yielded by the application of the detected refactoring operations and revised versions. We also applied our approach to actual classes stored in the *Apache Xerces*[2] SVN repository and confirmed the feasibility of the approach. The primary contributions of this paper can be summarized as follows:

- We present an approach to automatically detect impure refactoring. To our knowledge, this is the first attempt to detect impure refactoring from commits.
- We applied our approach to commits of actual classes stored in source code repository and confirmed that our approach can successfully detect impure refactoring.

2. APPROACH

The proposed approach takes the previous and the revised commits and a test suite of previous commit as input, then outputs the information of refactoring operations and non-refactoring changes between the commits. The proposed approach is comprised of two steps; At first, it detects refactoring operations using a graph search algorithm and a test suite. Then, it detects non-refactoring changes by computing the structural differences between state yielded by the application of the detected refactoring operations and the current commit. The approach assumes a commit of the program as a state, and a refactoring operation as a transition operator. Furthermore, a new state is generated by applying refactoring operations to a current state.

[1]https://svn.apache.org/viewvc?view=revision&revision=755231

[2]http://xerces.apache.org/

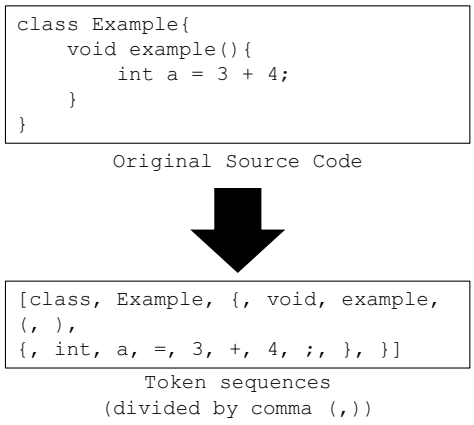

Figure 2: Example of token sequence convert

2.1 Detecting Refactoring with Graph Search

This step detects refactoring operations using a graph search algorithm, A^* search [12]. The A^* search estimates the total cost of a path involving a state n using a heuristic-based evaluation function. In this study, refactoring detections are formalized as states N (a set of program), initial state $s_o \in N$ (previous commit), and finial state s_m (current commit). Moreover, a next state is generated by the value of an evaluation function. The refactoring detection is comprised of following five steps:

Step1. Initialize priority queue and then enqueue a pair of initial state and its value of evaluation function $\langle s_0, e_0 \rangle$ into the queue.

Step2. Dequeue a pair which has the smallest evaluation value and the dequeued state is defined as s_i. The refactoring detection is terminated if the queue is empty.

Step3. Derive the candidates of refactoring operations $\delta_1, \ldots, \delta_n$ by comparing the differences between s_i and s_m. If there is no differences between them, the refactoring detection is terminated and then the s_i value is output.

Step4. Generate new states $\delta_0(s_i)$,$\delta_1(s_i)$,... by applying refactoring operations $\delta_1, \ldots, \delta_n$ into s_i, and then compute the evaluation function of each state.

Step4. Enqueue the gernated states in the previous step and values of evaluation function and then revert to Step 2.

Note that if refactoring detection does not finish within 600 seconds, the detection is terminated and then s_i, which has the smallest evaluation value, is output.

After the detection is finished, a test suite of initial state is exercised with s_i to check the modification of external behaviors of s_0 and s_i. If a test suite is succeed, implies that external behavior of s_0 and s_i was preserved, the approach proceeds to the next step described in Section 2.2. Meanwhile, if external behavior of the program is modified, the detection is terminated because this means that the refactoring operations are falsely detected.

2.1.1 Computing the Value of Evaluation Function

This study uses a evaluation function to select the next state. The value of evaluation function is computed using *Levenshtein distance*, which measures the minimal amount of changes necessary to transform one sequence of items into a second sequence of items [6]. In this study, *levenshtein distance* d_i between token sequences of the methods that shares the same method name is computed.

Let's assume that new states ($\delta_0(s_i)$,$\delta_1(s_1)$,...) generated in Step 4 is (A_1, A_2, ..., A_n) and final states is (B_1, B_2, ..., B_n), the number of tokens of A_1 and A_1 is a_i and b_i, respectively, then the value of evaluation function is defined as follows:

$$\frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} max(a_i, b_i)}$$

Note that a value of evaluation function takes $[0, 1]$.

2.1.2 Terminating Duplicated States

If a method `A` is pulled up after a method `B` is pulled and a method `B` is pulled up after method `A` is pulled, they will arrive at the same state. Therefore, the search should be terminated if it arrive at the same state after the second search. For alleviating this problem, the search is terminated if the hash value of the source code and the value of the evaluation function are exactly matched.

To compute a hash value of source code, code fragment is divided into each line of source code, and then a hash value is generated using method `hashCode()` in `String` Class in Java 8 API. For example, when a line is comprised of the following tokens:

$$(s_0, s_1, ..., s_{n-1})$$

The value of `hashCode()` is computed as follows:

$$31^{n-1}s_0 + 31^{n-2}s_1 + ... + s_{n-1}$$

2.2 Detecting Non-refactoring Changes

After refactoring detection is successfully performed, the structural differences between s_i and s_m are regarded as the non-refactoring changes. In this study, the non-refactoring changes are detected as follows:

1. Convert source code of s_i and s_m into a token sequence, respectively.
2. Compute *levenshtein distance* between the token sequences of matched members

2.2.1 Convert Source Code into Sequence of Tokens

Our proposed approach converts the source code of s_i and s_m into token sequences. The purpose of converting source code into token sequences is to ignore comments and white space. Figure 2 depicts an example of converting a Java source code into token sequences.

2.2.2 Match Members

The order of members in the class is not always fixed between the states. Therefore, this approach only computes structural differences between the matched members. For field declarations, they are defined to be matched if they have the same field name. For methods, they are defined to be matched if they have the same method name and the parameter type.

3. CASE STUDY

To show the feasibility of our approach, we applied it to two actual classes `DocumentImpl` and `CoreDocumentImpl` which are stored between two commits '318022' and '318023' in the *Apache Xerces* SVN repository[3]. This section discusses the result of case study and then explains threats to validity of the case study.

3.1 Result

Figure 3 depicts an overview of the search. Note that the refactoring operations were only detected in the class `DocumentImpl`. In this figure, circles represent searched states and the numbers on the circles represent the value of the evaluation function. Moreover, arrows represent state transitions and the search was conducted in the order of number written on the arrow. The detailed information is

[3]http://svn.apache.org/viewvc/xerces/java/

Table 1: Refactoring operations detected in a class `DocumentImpl`

No	Refactoring	Target member
1	F	Hashtable userData
2	M	setUserData(NodeImpl, Object)
3	M	getUserData(NodeImpl)
4	F	Hashtable userData
5	M	getUserData(NodeImpl)
6	F	Hashtable userData
7	M	getUserData(NodeImpl)
8	F	Hashtable userData
9	M	setUserData(NodeImpl, Object)
10	F	Hashtable userData
11	M	setUserData(NodeImpl, Object)
12	M	setUserData(NodeImpl, Object)
13	M	getUserData(NodeImpl)

Table 2: Non-Refactoring changes detected from a class `DocumentImpl`

Motivation types	#Removed	#Added
Addition of import statements	3	0
Additions of method bodies	9	0
Total	12	0

Table 3: Non-Refactoring changes detected from a class `CoreDocumentImpl`

Motivation types	#Removed	#Added
Addition of import statements	3	0
Additions of method bodies	18	0
Changes of method bodies	12	47
Addition of field declarations	3	0
Addition of methods	585	0
Total	621	47

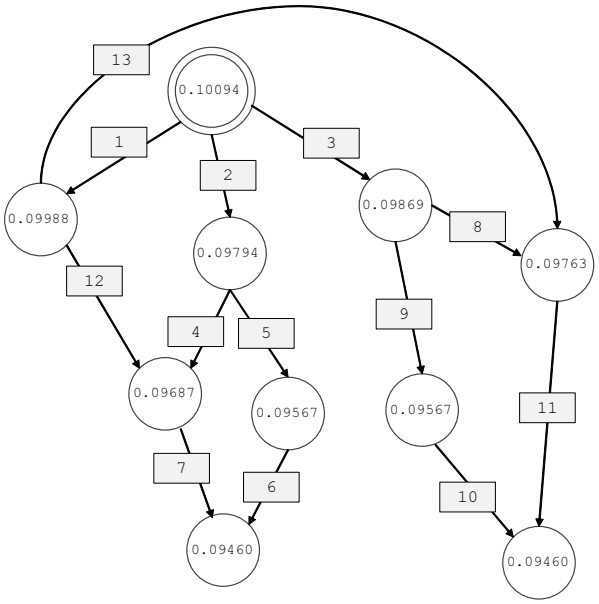

Figure 3: Overview of the search

shown in the Tables 1, 2, and 3. Table 1 illustrates the state transition diagram on the search. In this table, each number in the column '*No*' corresponds to the order number shown in Figure 3 and The column '*Refactoring*' represents the detected refactoring operations. In this column, 'F' represents '*Pull Up Field*' and 'M' represents '*Pull Up Method*'. As seen in this table, '*Pull Up Field*" and '*Pull Up Method*' refactoring operations were detected by our approach.

Moreover, Tables 2 and 3 depicts non-refactoring changes detected from `DocumentImpl` and `CoreDocumentImpl` classes, respectively. In these tables, column '*#Removed*' and '*#Added*' represents a number of removed and added tokens, respectively.

After the detection, we manually analyzed the source code of the classes `DocumentImpl` and `CoreDocumentImpl` and confirmed that our approach accurately detected '*Pull Up Field*' and '*Pull Up Method*' refactoring operations. For non-refactoring changes, we also manually checked the source code and confirmed that 12 tokens (class `DocumentImpl`) and 621 tokens (class `CoreDocumentImpl`) were really deleted and 47 tokens (class `CoreDocumentImpl`) were really added between the commits.

3.2 Threats to Validity

The result of case study might be lack of generalities. Even though our approach accurately detects impure refactoring, the result might change because we only apply our approach to two classes in the *Apache Xerces* project. As a future work, we plan to apply the approach to other commits and software systems to achieve the generality of our proposed approach.

4. SUMMARY

We proposed an approach that detects impure refactoring using a graph search algorithm, a test suite, and structural differences. We applied our approach to two actual classes in the *Apache Xerces* SVN repository and confirmed the feasibility of the approach.

For future work, we are plan to apply our proposed approach to additional open source software systems and industrial software systems.

5. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP25220003, JP26730036, JP16K16034, JP15H06344.

6. REFERENCES

[1] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.*, 107(C):1–14, Sept. 2015.

[2] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proc. of MSR*, pages 53–62, 2011.

[3] C. Gőrg and P. Weissgerber. Detecting and visualizing refactorings from software archives. In *Proc. of IWPC*, pages 205–214, 2005.

[4] A. E. Hassan. The road ahead for mining software repositories. In *Proc. of FoSM*, pages 48–57, 2008.

[5] S. Hayashi, Y. Tsuda, and M. Saeki. Search-based refactoring detection from source code revisions. *IEICE Trans. Inf. Syst.*, E93-D(4):754–762, apr 2010.

[6] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[7] R. Mahouachi, M. Kessentini, and M. Ó Cinnéide. Search-based refactoring detection. In *Proc. of GECCO Companion*, pages 205–206, 2013.

[8] N. A. Milea, L. Jiang, and S.-C. Khoo. Vector abstraction and concretization for scalable detection of refactorings. In *Proc. of FSE*, pages 86–97, 2014.

[9] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM*, pages 1–10, 2010.

[10] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *Proc. of WCRE*, pages 263–274, 2006.

[11] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue. Revisiting the relationship between code smells and refactoring. In *Proc. of ICPC*, pages 48–57, 2016.

[12] W. Zeng and R. L. Church. Finding shortest paths on real road networks: The case for A*. *Int. J. Geogr. Inf. Sci.*, 23(4):531–543, Apr. 2009.