Comparative Study between Two Approaches Using Edit Operations and Code Differences to Detect Past Refactorings

Takayuki OMORI†(a) and Katsuhisa MARUYAMA††, Members

SUMMARY Understanding which refactoring transformations were performed is in demand in modern software constructions. Traditionally, many researchers have been tackling understanding code changes with history data derived from version control systems. In those studies, problems of the traditional approach are pointed out, such as entanglement of multiple changes. To alleviate the problems, operation histories on IDEs’ code editors are available as a new source of software evolution data nowadays. By replaying such histories, we can investigate past code changes in a fine-grained level. However, the prior studies did not provide enough evidence of their effectiveness for detecting refactoring transformations. This paper describes an experiment in which participants detect refactoring transformations performed by other participants after investigating the code changes with an operation-replay tool and diff tools. The results show that both approaches have their respective factors that pose misunderstanding and overlooking of refactoring transformations. Two negative factors on divided operations and generated compound operations were observed in the operation-based approach, whereas all the negative factors resulted from three problems on tangling, shadowing, and out-of-order of code changes in the difference-based approach. This paper also shows seven concrete examples of participants’ mistakes in both approaches. These findings give us hints for improving existing tools for understanding code changes and detecting refactoring transformations.

key words: software evolution, fine-grained code change, understanding code change, refactoring detection

1. Introduction

Software evolves by source code changes to keep meeting ever-changing requirements. To make evolution of existing software successful, understanding code changes of the software is a significant task. Most approaches to support this task have been based on data extracted from version control systems (VCSs), such as Git and Subversion. In these approaches, a code change is defined as a difference between two source code snapshots. Such difference data can be easily collected since many real-world open source software projects are adopting VCSs.

Unfortunately, these traditional approaches have shortcomings resulting from the granularity of detected differences. Multiple code changes under different intentions are often committed together into a single revision. This problem is called a tangling problem. Moreover, several changes are sometimes overwritten by later changes before their commit. This problem is called a shadowing problem. The order of code changes that were committed together can never be retrieved in the traditional approaches although capturing the order helps us to correctly understand past changes. The authors named it an out-of-order problem.

To address the aforementioned problems, several change-aware tools were proposed, such as SpyWare [1], OperationRecorder [2], Syde [3], Fluorite [4], CheOPSJ [5], and CodingTracker [6]. These tools can record all edit operations that have been actually performed in the past. Moreover, there are several techniques complementing the recording tools. Hayashi et al. proposed edit history refactoring [7], [8], a technique to restructure edit operations to promote Task Level Commit [9]. Matsuda et al. proposed a categorization mechanism for edit operations in order to automatically reorganize them based on change types [10]. Kitsu et al. proposed a technique to group the same kind of edit operations based on the time and place of performed operations to help developers understand tangled changes [11].

To directly support understanding of past code changes, several tools for replaying recorded edit operations have also been proposed. Hattori et al. developed a tool that replays operations recorded by Syde [12]. Yoon et al. devised a tool called Azurite, which visualizes fine-grained code change history and provides the replay function that shows differences between before and after specified code changes [13]. Simmons created a replay tool called CodeSkimmer, which is based on change history recorded by CodingTracker [14]. The authors have already presented a replay tool called OperationReplayer in previous papers [15], [16]. It can completely replay edit operations recorded by OperationRecorder, which records all edit operations that have been performed on the Eclipse Java editor. The authors also presented OperationSliceReplayer [17], [18], which aims at reducing the number of edit operations that have to be replayed to investigate how a specific program element was changed.

Those prior studies on edit operations emphasized that fine-grained code changes at the edit-operation level are more useful than ones at the snapshot-based level in terms of understanding them. However, those studies do not provide enough evidence to demonstrate their effectiveness for understanding past code changes in various development situations. This paper presents a comparative experiment with two approaches: a newer one replaying edit operations (an
operation-based approach) and a traditional snapshot-based one using code differences (a difference-based approach).

In the experiment that the authors conducted, three participants applied several refactoring transformations to source code of existing programs. Next, they investigated refactoring transformations applied by other participants. In this investigation, they used OperationReplayer in the operation-based approach and their favorite diff tools (e.g., GNU diff) in the difference-based approach. Each of the participants tackled detecting the applied refactoring transformations with either of the two approaches. They answered which refactoring transformations were applied during a given development session. Finally, the authors checked whether their answers are correct, and investigated the causes for mistakes in each approach.

Here, the target code change of our experiment is limited to refactoring transformations. In general, code changes have wide variety, and thus we cannot judge whether developers and maintainers correctly figure out such code changes. Fortunately, many refactoring transformations have been well defined in catalogs [19], [20]. Participants in our experiment (pseudo-developers or pseudo-maintainers) can easily explain code changes of interest if all of them are related to a particular refactoring transformation. In other words, this limitation makes it easier to evaluate participants’ answers.

Contributions of this paper are as follows:

- To present results of the comparative experiment with operation-based and difference-based approaches in detecting past refactoring transformations applied by other participants.
- To categorize the causes of mistakes in refactoring detection in both approaches; two specific causes in the operation-based approach and three in the difference-based approach.
- To show seven case studies that cover combinations of the types of mistakes and specific causes.
- To state findings and remarks based on the experiment for improving existing tools for understanding code changes and detecting refactoring transformations.

This paper is structured as follows. Section 2 describes the background of our study. Section 3 describes our previously proposed tool called OperationReplayer. Sections 4 and 5 present the experimental procedures and results, respectively. Section 6 shows more profound analyses of the experimental results, including causal analyses of participants’ mistakes and case studies. Section 7 presents discussions including experimental findings and threats to validity. Section 8 concludes with a short summary and future work.

2. Background

As the background, this section presents three problems that arise in the traditional difference-based approaches and explains the necessity of manual detection of refactoring transformations.

2.1 Problems in the Use of Code Differences

As mentioned in Sect. 1, the simple use of code changes stored in VCSs might involve three problems: tangling, shadowing, and out-of-order.

A tangling problem occurs when code changes with different intentions are committed together into a single revision. Gög and Weißgerber investigated distribution of refactorings among transactions in a VCS using two OSS projects, and the result showed a single transaction (a set of simultaneous commits) sometimes contains a bunch of refactoring transformations [21]. Murphy-Hill et al. presented that most refactoring transformations occur as a part of a commit tangled with other changes [22]. Negara et al. reported that around half of methods were both refactored and additionally changed in the same commit [6]. Herzig and Zeller reported that a certain of bug-fix commits involve multiple changes unrelated or loosely related to each other in open source projects [23]. Kirinuki et al. pointed out difficulties in working with VCSs due to tangled changes [24]. Brindescu et al. reported that distributed version control systems contribute to finer-grained commit, but disentangling changes is still developers’ task [25].

A shadowing problem occurs when several code changes are overwritten by later ones before their commit. Robbes and Lanza presented an example of overlapped refactoring transformations, which let traceability of a program entity lost between code snapshots [1]. Negara et al. revealed that more than a third of all code changes and 30% of refactoring transformations do not reach VCS [6]. Soetens et al. reported that refactoring transformations fail to be reconstructed by difference-based approaches when multiple edit operations were performed on the same code entities [27].

An out-of-order problem indicates that the order of code changes that were committed together can never be restored from difference-based histories. The problem hinders developers and maintainers from keeping track of fine-grained code changes and looking at their chronological sequence. A controlled experiment conducted by Hatrori et al. demonstrated that chronologically replaying code changes outperforms difference-based histories to find answers to questions related to software evolution [12].

2.2 Necessity of Manual Refactoring Detection

Refactoring improves the quality of existing source code, especially readability and maintainability, without changing its observable behavior [19], and is an essential activity in modern software construction. In actual software development, many developers and maintainers want to know how refactoring transformations were applied [28], [29]. It is ideal that automated tools could detect all the conducted refactoring transformations. However, automatic detection is still difficult even with the state-of-the-art tools. Therefore, at this point, manual detection is required to obtain
high accuracy in refactoring detection.

Automatic refactoring transformations are in general activated through a refactoring menu and thus they are easily detected. Mylyn (formerly called Mylar) can record developers’ command executions including refactoring [31]. MolhadoRef can also record automated refactoring transformations and other edits and apply them to merge multiple developers’ code changes on a software configuration management system [32]. However, applicability of such tools is limited since a major part of refactoring transformations is manually performed [22], [26]. Moreover, automatic refactoring detection tools miss up to 60 percent of high-level refactorings in Murphy-Hill’s study [22].

To identify which kinds of refactoring transformations are applied, refactoring detection techniques and tools have been proposed, mainly based on VCS history [28], [29], [33], [34]. Unfortunately, they also suffer from the aforementioned three problems. Moreover, Soares et al. reported that automated refactoring detection approaches were no match for manual ones [30].

There are also refactoring detection techniques based on fine-grained edit operations, which were proposed by Negara et al. [26] and Soetens et al. [27]. Although these techniques improved the accuracy of detection, they still handle limited kinds of refactoring transformations. Therefore, manual investigations of every code change that might involve a refactoring transformation are required.

3. OperationReplayer

Figure 1 shows a screenshot of OperationReplayer. It provides the Replay perspective containing six views: File List, Source Viewer, Control Buttons, Filter, Operation History, and Replay Visualization.

The File List presents the names of all source files that can be restored from recorded operations. The Source Viewer shows the snapshot of source code that has been lastly restored. The Filter is used for selecting particular operations that satisfy a given condition. For example, the filter “file=Frame.java” can pick up all operations that were performed in the source file named Frame.java. The Operation History shows the list of recorded operations. Each line in this view indicates one of the following five operations:

1. A normal operation that presents an insertion, deletion, or replacement (a combination of a deletion and an
insertion) of text on the editor. The inserted and deleted strings stored in this operation can be replayed in the Source Viewer.

(2) A compound operation bundles multiple normal operations resulting from activation of a particular menu item, such as a refactoring menu item. It holds the label indicating the operation kind (e.g., refactoring name). The bundled normal operations are replayed together.

(3) A copy operation presents a copy of text.

(4) A menu operation presents the activation of a menu function.

(5) A file operation corresponds to open, close, activation, or save of a source file.

Each of those operations stores the values of common attributes: the name of a developer who performed it, the time when it was performed, and the name of a source file in which it was performed.

During replaying operations, OperationReplayer designates a single focal operation that is highlighted in the Operation History. The Source Viewer shows a code snapshot immediately after the focal operation was performed. The edited and copied texts of a focal operation are highlighted with their respective colors in the Source Viewer.

Tool users can freely specify a focal operation, using the Operation History, Replay Visualization, or Control Buttons. The first way is to directly select the focal operation in the Operation History. The second way is to double-click on the time line bar in the Replay Visualization that provides an abstract visualization of recorded operations. The focal operation is assigned to the nearest one corresponding to the clicked point. The final way is to use the Control Buttons. The tool users can push one of the six buttons to go back to the first operation, go back to the previous development session, go back to the previous operation, go to the next operation, go to the next development session, or go to the last operation. Unchecked operations in the Operation History are skipped in using these buttons. Here, development sessions are determined based on the time period between two successive operations. In each development session, all the successive operations were performed within 60 minutes as with [35]. When the focal operation is changed, the Source Viewer immediately updates source code and scrolls to show how the code is changed by the focal operation.

4. Experimental Procedures

The authors conducted an experiment of detecting refactoring transformations. This section presents the details of the experimental procedures.

4.1 Phase 0: Data Preparation

Operation history data were obtained from four programs. Table 1 shows the numbers of source files and lines of code (non-blank, non-comment) within the programs. All the programs implement a puzzle game imitating Puyo Puyo†.

![Table 1](https://en.wikipedia.org/wiki/Puyo_Puyo)

<table>
<thead>
<tr>
<th>Program</th>
<th># of files</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog-1</td>
<td>24</td>
<td>2,062</td>
</tr>
<tr>
<td>prog-2</td>
<td>27</td>
<td>1,748</td>
</tr>
<tr>
<td>prog-3</td>
<td>16</td>
<td>1,089</td>
</tr>
<tr>
<td>prog-4</td>
<td>14</td>
<td>1,481</td>
</tr>
</tbody>
</table>

Each of them was written by a single developer, a Ph.D. or M.Sc. student when the program was created. They implemented their independent features in addition to the game’s basic rules and the programs have different sizes. This did not seem to be an obstacle to understand the source code and detect applied refactoring in the experiment.

The authors recruited three research participants: two M.Sc. students and one B.Sc. student. None of them are developers of the programs shown in Table 1. All the participants have experience in Java programming over one year, and have acquired knowledge about refactoring of object-oriented programs in the classroom and through their research activities.

The rest of the experiment consists of the following three phases:

1. The participants refactored the programs.
2. They detected the applied refactoring transformations performed by other participants.
3. The authors checked their answers.

The details of each phase are described below.

4.2 Phase 1: Refactoring Transformations

The research participants conducted refactoring transformations in this phase. This phase is composed of the following three sub-phases.

4.2.1 Understanding the Original Code

The participants tried to roughly understand the source code of all the programs to refactor them later. They read the code displayed in the Eclipse Java editor and ran the programs if needed. The goal of this sub-phase is to identify which code fragments would be refactored. This sub-phase was completed within about two hours for each program.

4.2.2 Refactoring the Original Code

The participants applied refactoring transformations to remove code smells from the original code. Every operation in this phase was recorded by OperationRecorder. The recorded operations were used in later phases. They also wrote down a memo about the applied refactoring transformations every time they refactored the code. The memo contains information about the time, kind, and target program element of each applied refactoring transformation. The time period of this sub-phase was limited to one hour for each program to prevent the participants from overworking.
4.2.3 Completing Refactoring Memos

Each of the participants checked code changes in the applied refactoring transformations by replaying the recorded operations by himself using OperationReplayer and then finished the refactoring memos that he wrote down. In addition to completing the memo, the aim of this sub-phase also includes to let the participants know how to use the replayer. This sub-phase was completed within one hour.

In writing down refactoring memos, the participants were allowed to ignore the minor difference of the applied refactoring transformations. For example, they did not have to write down Rename Class and Rename Local Variable refactoring transformations separately since they often occur in a row. In another experiment of refactoring, it was observed that a developer changed the name Field of a class into Panel (Rename Class) and then changed the name field of a local variable whose type is Panel into panel (Rename Local Variable). In this case, someone would write down both of the two refactoring transformations since he considers them independent. On the other hand, someone else would think them as a single Rename refactoring transformation when he does not think their separation is significant. To avoid confusion, the authors assumed that consecutive multiple refactoring transformations are treated as a single one when their boundary is hard to be determined.

4.3 Phase 2: Detecting Refactoring Transformations

The participants tried to understand code changes performed by other participants and detect the refactoring transformations. The time period of this phase was limited to one hour for each program. In this phase, each of the participants had to do his task for each of the programs, using either of the following two approaches, which was assigned by the authors.

(i) Operation-based approach

Each of the participants used OperationReplayer to replay edit operations performed by other participants. They chronologically replayed operations one-by-one without any filtering. Rewinding was performed when they wanted to carefully investigate particular operations.

(ii) Difference-based approach

Each of the participants used his favorite diff tool to compare the snapshots of source code. In the experiment, one of them (participant “A” in Table 3) used the GUI-based compare tool of Eclipse and the other participants used the CUI-based GNU diff tool with the -r (recursive) option. In addition, they were allowed to examine source code snapshots by using a plain text editor if needed.

Prior to this phase, the authors made a list of names of all refactoring transformations appearing in the refactoring memos written in Phase 1. The participants could refer the list and select a proper name from it as their answer. Thus, they created detection memos that describe which kind of refactoring transformation was performed. In addition, they also wrote the time information when the refactoring transformation was performed in the operation-based approach. In the difference-based approach, they wrote a program element as a refactoring target, so that the authors could correctly match the refactoring memos and detection memos in Phase 3.

To avoid biases caused by individual participants and detection methods, two of the participants investigated the same program refactored by the other participant using different approaches. For example, in case that participant “A” refactored program “prog-1”, participant “B” investigated it with the operation-based approach and participant “C” did with the difference-based approach, as shown in Table 3.

4.4 Phase 3: Checking Answers

After the participants’ refactoring detection, the authors cleaned the obtained data and checked the answers.

4.4.1 Cleaning Data

After the detection of refactoring transformations by all the participants, the authors cleaned data in the collected refactoring memos. In the experiment, some of the memos included refactoring transformations that are not listed in existing major refactoring catalogs. For example, every participant deleted unused program elements to clean code (it was called Code Clearance in the experiment). Although these code changes seem to be refactoring transformations without changing their observable behavior, they are very trivial. Therefore, they were removed from the memos. Moreover, one of the participants created new packages and moved existing classes into them. Those changes are hard to be detected with either of the two approaches since they appear in non-intuitive forms in code change data in our experiment. Hence, they were also removed from the memos. Code changes that did not preserve the behavior of code (e.g., bug fix) and ones caused by undoing previous refactoring transformations were also removed.

A refactoring sometimes involves sub-refactorings [19]. In the experiment, for example, an Extract Superclass refactoring transformation was observed as a parent, consisting of a Pull Up Method and Pull Up Field as its children. In this case, the participant might detect only the parent since he might consider that the detection of the children is trivial. Hence, the authors omitted refactoring transformations from the memos if they are enclosed in another one. Therefore, the existence of sub-refactorings never affected the resulting memos.

4.4.2 Checking the Answers

The authors checked whether participants’ answers are correct or not by comparing the detection memos and their corresponding refactoring memos. In the difference-based approach, no answers have time information. Hence, the authors found corresponding refactoring transformations be-
between detection and refactoring memos based on the refactored program elements. In addition, the answers corresponding to the refactoring transformations removed in 4.4.1 (e.g., Code Clearance) were ignored.

5. Experimental Results

This section shows experimental results on performed refactoring transformations and their detection.

5.1 Applied Refactoring Transformations

Table 2 shows all the refactoring transformations observed in the experiment (after cleaning data) and the numbers of their respective occurrences. The 16 of 23 kinds of refactorings shown in the upper side of the table are introduced in [19]. The Limit Instantiation with Singleton refactoring is introduced in [20]. The authors should explain the remaining six ones since they are newly introduced in this paper.

**Table 2** Numbers of observed refactoring transformations

<table>
<thead>
<tr>
<th>Refactoring</th>
<th># of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Add Parameter</td>
<td>1</td>
</tr>
<tr>
<td>2 Consolidate Conditional Expression</td>
<td>3</td>
</tr>
<tr>
<td>3 Consolidate Duplicate Conditional Fragments</td>
<td>1</td>
</tr>
<tr>
<td>4 Encapsulate Collection</td>
<td>1</td>
</tr>
<tr>
<td>5 Encapsulate Field</td>
<td>5</td>
</tr>
<tr>
<td>6 Extract Class</td>
<td>2</td>
</tr>
<tr>
<td>7 Extract Interface</td>
<td>1</td>
</tr>
<tr>
<td>8 Extract Method</td>
<td>12</td>
</tr>
<tr>
<td>9 Extract Superclass</td>
<td>2</td>
</tr>
<tr>
<td>10 Inline Class</td>
<td>1</td>
</tr>
<tr>
<td>11 Introduce Explaining Variable</td>
<td>3</td>
</tr>
<tr>
<td>12 Pull Up Method</td>
<td>2</td>
</tr>
<tr>
<td>13 Remove Parameter</td>
<td>1</td>
</tr>
<tr>
<td>14 Rename</td>
<td>6</td>
</tr>
<tr>
<td>15 Replace Magic Number with Symbolic Constant</td>
<td>2</td>
</tr>
<tr>
<td>16 Substitute Algorithm</td>
<td>10</td>
</tr>
<tr>
<td>17 Limit Instantiation with Singleton</td>
<td>2</td>
</tr>
<tr>
<td>18 Introduce Enum</td>
<td>2</td>
</tr>
<tr>
<td>19 Make Class Top-Level</td>
<td>1</td>
</tr>
<tr>
<td>20 Make Field Final</td>
<td>1</td>
</tr>
<tr>
<td>21 Make Field Static</td>
<td>1</td>
</tr>
<tr>
<td>22 Replace For-Loop with Enhanced For-Loop</td>
<td>2</td>
</tr>
<tr>
<td>23 Substitute Data Structure</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 3 shows the results of checking participants’ answers. Table 3 (a) and (b) correspond to the operation-based approach and difference-based one, respectively.

5.2 Results of Refactoring Detection

Table 3 shows the results of checking participants’ answers. Table 3 (a) and (b) correspond to the operation-based approach and difference-based one, respectively.

The Program column denotes the refactored programs. The number assigned to each program corresponds to one in Table 1. The Ref. and Det. columns show participants who refactored each of the programs and detected refactoring transformations from it, respectively. CA, NA, and IA columns indicate the numbers of correct answers, no answers (overlooking), and incorrect answers, respectively. Consider, for example, the second row of Table 3 (a). In this case, 6 (= 3 + 1 + 2) refactoring transformations were totally performed to “prog-1” by participant “B”, and participant “C” correctly detected three of them, using the operation-based approach. On the other hand, he overlooked one of

<table>
<thead>
<tr>
<th>Program</th>
<th>Ref.</th>
<th>Det.</th>
<th>CA</th>
<th>NA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog-1</td>
<td>A</td>
<td>B</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>prog-1</td>
<td>B</td>
<td>C</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>prog-1</td>
<td>C</td>
<td>A</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog-2</td>
<td>A</td>
<td>B</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>prog-2</td>
<td>B</td>
<td>C</td>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>prog-2</td>
<td>C</td>
<td>A</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>prog-3</td>
<td>A</td>
<td>C</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>prog-3</td>
<td>B</td>
<td>A</td>
<td>4</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>prog-3</td>
<td>B</td>
<td>C</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog-4</td>
<td>A</td>
<td>C</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog-4</td>
<td>B</td>
<td>A</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog-4</td>
<td>C</td>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>51</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Ref.</th>
<th>Det.</th>
<th>CA</th>
<th>NA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog-1</td>
<td>A</td>
<td>C</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>prog-1</td>
<td>B</td>
<td>A</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>prog-1</td>
<td>C</td>
<td>B</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>prog-2</td>
<td>A</td>
<td>C</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>prog-2</td>
<td>B</td>
<td>A</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>prog-2</td>
<td>C</td>
<td>B</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>prog-3</td>
<td>A</td>
<td>B</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>prog-3</td>
<td>B</td>
<td>C</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>prog-3</td>
<td>C</td>
<td>A</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>prog-4</td>
<td>A</td>
<td>B</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>prog-4</td>
<td>B</td>
<td>C</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>prog-4</td>
<td>C</td>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>42</td>
<td>17</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
them and detected two of them incorrectly. Note that there is no answer associated with a refactoring transformation that was not conducted, that is, none of the participants misunderstood non-refactoring code changes as refactoring transformations.

Overall, the operation-based approach is superior to the difference-based one in terms of the number of correct answers; the total numbers of CA are 51 (77%) and 42 (64%) out of 66, respectively. This could be due to the number of NA. The number of NA in the operation-based approach is quite smaller than that in the difference-based approach. Negatively, the total number of IA increased when using the operation-based approach.

5.3 Other Results

After the completing experimental procedures, the authors conversed with participants about the experiment. Then, one of the participants deposed that replaying requires more time than investigating results of diff tools. He said that the operation-based approach took about one hour (that is a limited time period) in many cases, whereas one hour is enough for all the tasks in the difference-based approach.

6. Profound Analyses

This section describes deeper analyses of the experimental results, including causal analyses and case studies of participants’ mistakes.

6.1 Causal Analyses

The authors deeply investigated all the false answers (NA and IA) and estimated why the participants made their mistakes. In the causal analyses, the authors investigated the total of the 39 ($= 5 + 10 + 17 + 7$) answers. Figures 2 and 3 show how causes are estimated in difference-based and operation-based approaches, respectively. Note that the figures do not show the actual procedures of causal analyses, but the formalized cause estimation method. In other words, the authors completed the required process, such as informal interviews and collecting refactoring operations, in prior to the causal analyses.

Before explaining the cause estimation method in detail, the meanings of resultant cause categories (shown as rounded rectangles in Figs. 2 and 3) are explained as follows:

(1) Causes in the operation-based approach

The operation-based approach involves two specific causes: “Dividing a single code change into operations” and “Generating compound operations”.

Dividing a single code change into operations: This cause represents unwanted separation of code changes. In the operation-based approach, a single code change is often divided into multiple operations although they are strongly related with each other. The authors consider that the separated operations make it harder to grasp the whole picture of code changes and thus easily induce incorrect detection of refactoring transformations.

Generating compound operations: This cause represents unwanted composition of code changes. The activation of an Eclipse’s menu function (e.g., automated refactoring) produces a compound operation bundling multiple normal operations. The bundled operations are sometimes distributed in the extended area of a source file or multiple source files. Nevertheless, OperationReplayer initially shows the code change at the place selected by the developer when the function was activated. The authors consider that this makes it harder to detect refactoring transformations and then easily induces overlooking them.

(2) Causes in the difference-based approach

The difference-based approach involves three specific causes: “Tangling multiple code changes”, “Shadowing code changes”, and “Out-of-order interclass code changes”. These causes are all related to the problems described in 2.1.

Tangling multiple code changes: This cause associates with the tangling problem. For example, the adjustment of white space characters in automatic reformat might widely change source code. In the difference-based approach, these code changes are tangled with others since all differences are extracted based on the locations the changes were made within a development session. Like this, it is hard to de-
tect refactoring transformations when multiple changes under different intentions tangle with each other.

**Shadowing code changes**: This cause associates with the shadowing problem. If a refactoring transformation produces code changes and some of them are canceled or overwritten later within the same development session, such code changes can be never observed in the difference-based approach.

**Out-of-order interclass code changes**: A refactoring transformation might change code fragments in various locations. Particularly, code changes straddling over multiple source files induce a serious problem. For example, a **Pull Up Method** refactoring is usually associated with two different classes (source files); a child class having the moved method before the refactoring and a parent class having it after the refactoring. In the difference-based approach, it is hard to find an association between these two classes since information about the order of code changes occurring in those classes is lost (the out-of-order problem). If this information (ideally, the time information about when the code changes are made) remains, the association would be easily revealed. This is because code changes in the same refactoring transformation are in general intensively performed in a short time period.

(3) **Causes in both approaches**

The following two causes were observed in both approaches.

**Lack of knowledge and skills**: During checking answers, the authors found that several mistakes seem to be caused by participants’ lack of knowledge and skills. This is because all of the participants are students. Industrial developers and maintainers seldom make such mistakes. All of the observed cases are shown below:

- Confusing “**Inline Class**” with “**Extract class**”.
- Confusing “**Extract Superclass**” (extracting an abstract superclass) with “**Extract Interface**”.
- Considered an addition of private static field named **instance** (a part of “**Limit Instantiation with Singleton**”) as “**Make Field Static**”.
- Confusing “**Consolidate Conditional Expression**” (without code movement beyond classes) with “**Pull Up Method**”.
- Confusing “**Encapsulate Collection**” with “**Parameterized Method**”.

**Careless mistakes**: This cause is selected when the case corresponds to none of the other causes. Since the authors could not find factors that seriously hamper participants from correctly answering in those cases, the cause was considered as careless mistakes.

6.1.1 **Cause Estimation**

This section explains how the causes are estimated in each approach.

First, the result from informal interviews is considered. When the authors had trouble with estimating the cause, they conducted an informal interview to the participant who was in charge of the refactoring detection. If the participant deposed that it was just a careless mistake, the cause was determined as “Careless mistakes”.

Then, the participant incorrectly answered is considered. In several cases, the authors considered the mistake was caused by the participant’s unskillfulness as explained earlier. In this judging, the authors used OperationReplayer to check the content of refactoring operations (ROPs). ROPs indicate operations composing the refactoring transformation and are the key to notice the refactoring transformation. For example, in a refactoring transformation that moves a program element, a deletion (cut) of the element and its insertion in another place would be ROPs. Operations that update references to the element are excluded from ROPs. Other kinds of refactoring transformation that replaces program elements are in similar way. When the authors did not consider that the ROPs did not affect the mistake, the cause is determined as “Lack of knowledge and skills”. This is because the participants who made the false answers seemed to confuse refactoring names or design pattern. Note that this determination depends on the authors’ conjecture and the mistakes might be actually due to just careless mistakes or other reasons.

Otherwise, the cause is estimated by characteristics of ROPs. If the ROP is a single edit operation, the cause is “Generating compound operations”. Usually a refactoring transformation cannot be done with a single edit operation except for a compound change that corresponds to an automated refactoring or other Eclipse’s function. If ROPs are multiple edit operations which are separated by other edit operations, the cause is “Dividing a single code change into operations”. If the case does not fall into the above ones, the cause is determined as “Careless mistakes”. This is because the author could not find any obstacles against the correctly answering in those cases.

The cause estimation in the difference-based approach is in the similar way. Figure 3 shows the estimation method for the difference-based approach. In the difference-based approach, the step of informal interview is omitted from the figure since no case is categorized as careless mistakes here.

The method employs refactoring code changes (RCCs). RCCs indicate code differences composing the refactoring transformation and the key to notice the refactoring transformation, in similar to ROPs. Since diff-tools used in the experiment show a code difference as a hunk (inserted lines, deleted lines, or their combination), a hunk in a RCC may include other kind of code change.

First, incorrect answers and RCCs are used for determining “lack of knowledge and skills”. The process is as the same as the operation-based one except for using the corresponding diff tool. The authors used the same tool as the participant used in each case.

After that, the cases are estimated based on characteristics of RCCs. As the Fig. 3 shows, the causes derived from
### Table 4 Causes of NAs and IAs

<table>
<thead>
<tr>
<th>Cause</th>
<th>NA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Operation-based approach</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dividing a single code change into operations</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Generating compound operations</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Careless mistakes</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Lack of knowledge and skills</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cause</th>
<th>NA</th>
<th>IA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Difference-based approach</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tangling multiple code changes</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Shadowing code changes</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Out-of-order interclass code changes</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Careless mistakes</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Lack of knowledge and skills</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>17</td>
<td>7</td>
</tr>
</tbody>
</table>

### 6.2 Case Studies

This section presents seven case studies of incorrect detections of refactoring transformations. These case studies were selected by the authors to cover every observed combination of causes (except for “Careless mistakes” and “Lack of knowledge and skills”) and error types (NA and IA).

Cases 1 and 2 show incorrect detections in the operation-based approach where participants used OperationReplayer. Cases 3 to 7 are in the difference-based approach. The CUI-based GNU diff tool was used in Cases 3 to 6, whereas the GUI Eclipse compare tool was used in Case 7.

#### 6.2.1 Case 1: Operation-Based Approach

**Refactored:** SUBSTITUTE DATA STRUCTURE  
**Detected:** SUBSTITUTE ALGORITHM (IA)  
**Cause:** Dividing a single code change into operations

In this SUBSTITUTE DATA STRUCTURE, an array (java.awt.Point[] offset) was replaced by two arrays (int[] dx, dy). Figure 4 shows the relevant operations and source code fragment that appeared just after the operations (a) were performed. This refactoring transformation introduced the new declarations of dx and dy between 16:37:35 and 16:37:52 (a), then it deleted the declaration of offset at 16:38:28 (b). Although the textual insertions and deletion were recorded in a relatively short time period, the participant could not understand the intention behind the code changes. In this case, there are 10 edit operations recorded between the insertions and deletion. The interrupting operations are caused by the SUBSTITUTE DATA STRUCTURE refactoring since they were actually performed for fixing references to offset. However, the participant considered them as SUBSTITUTE ALGORITHM since such a code change that affects a method body is often performed for altering the algorithms. The authors think the key to understand this refactoring transformation is recognizing the relationship between the insertions of dx and dy and the deletion of offset. However, interrupting operations prohibited the participant from correctly recognizing it. Therefore, the cause of this misunderstanding is considered as dividing a single code change into operations.

#### 6.2.2 Case 2: Operation-Based Approach

**Refactored:** EXTRACT METHOD  
**Detected:** None (NA)  
**Cause:** Generating compound operations

In this EXTRACT METHOD, a code fragment was extracted from a complicated method drawNext(), and a new method draw1pDoubleNextCell() including the extracted code fragment was created. Figure 5 shows the relevant operation and source code fragment that appeared just after the operation (a) was performed. According to the recorded oper-
6.2.3 Case 3: Difference-Based Approach

**Refactored:** Extract Method  
**Detected:** None (NA)  
**Cause:** Tangling multiple code changes

The applied refactoring transformation was the same as Case 2. The code differences in this case are shown in Fig. 6. In Extract Method refactoring, two hunks (a) and (b) containing similar code fragments can be found in the code differences between code snapshots before and after the refactoring transformation.

The participant scanned every code change to detect refactoring transformations in the difference-based approach. Unfortunately, he failed to find similar hunks in the code differences. In this case, similar parts derived from this refactoring transformation were separated by lines derived from multiple kinds of other refactoring transformations. The author thought this is the cause why the participant overlooked the similarity. This is a typical case of entanglement of code differences. The entanglement often causes spatial separation of code changes derived from the same refactoring transformation in the diff result, like this case.

Since the name of the extracted method was changed, the diff result shows the different name (drawDoubleNextCell). However, missing the original name (draw1pDoubleNextCell) itself does not significant to recognize the Extract Method refactoring. Therefore, the authors did not consider the name change as loss of RCC in estimating the cause.

6.2.4 Case 4: Difference-Based Approach

**Refactored:** Substitute Data Structure  
**Detected:** Substitute Algorithm (IA)  
**Cause:** Tangling multiple code changes

The applied refactoring transformation was the same as Case 1. The code differences in this case are shown in Fig. 7. The declaration of offset existed in 8th line in a source file before the refactoring transformation (a), and the new declarations (dx and dy) are located in 32nd and 33rd lines in the same source file (b). Thus, the deletion and insertion are presented in different hunks when the participant uses the textual diff tool. Moreover, there are other 5 hunks derived from other refactoring transformations separating off those two hunks in this case. Most code changes (including the introduction of dx and dy) were performed in a method body. The authors thought the tangled operations hindered the participant from correctly recognizing the Substitute Data Structure. Moreover, the author considered that the participant confused intentions of code changes since operations in a method body are often observed in Substitute Algorithm refactoring.

6.2.5 Case 5: Difference-Based Approach

**Refactored:** Rename Method  
**Detected:** None (NA)  
**Cause:** Shadowing code changes

This Rename Method changed the name of a method gameStart() to beforeGame(). In fact, gameStart() was extracted from run() in a previous refactoring transformation. In this case, the participant looked at the following textual difference when he used the GNU diff tool.
This difference denotes a new method `beforeGame()` was inserted in the source file. We cannot know that `gameStart()` was actually extracted from this difference since the method name was completely overwritten by later code changes (the shadowing problem). If a single development session contains this kind of shadowing, no one can detect the `RENAME METHOD` by looking at only the textual difference no matter how long it took. In the experiment, the participant could detect only the `EXTRACT METHOD` but not the `RENAME METHOD`. Interestingly, the shadowing in this case made the participant detect only the `EXTRACT METHOD` performed earlier than the `RENAME METHOD`. When the shadowing problem occurs, the earlier operation is overwritten by later ones in the textual level. However, the earlier refactoring transformation is not always hidden by later one when we focus on the kind of the refactoring.

In this refactoring transformation, the operation-based approach brought the correspondent participant to a successful detection. This is because he could easily keep track of every code change derived from the applications of both `EXTRACT METHOD` and `RENAME METHOD`. From this perspective, the operation-based approach seems to be better than the difference-based one. However, this conclusion might provoke an argument. If developers and maintainers want to know only the code change not including overwritten ones, the difference-based approach is useful enough.

6.2.6 Case 6: Difference-Based Approach

**Refactored:** `EXTRACT SUPERCLASS`  
**Detected:** None (NA)  
**Cause:** Out-of-order interclass code changes

This `EXTRACT SUPERCLASS` introduced a new class `AbstractPanel` as the common parent of four subclasses. The code changes of this refactoring transformation were performed in five classes: `AbstractPanel` and its children. With the recorded operations, we can know that they were performed intensively in a short period of time, and the participant in the operation-based approach correctly answered in this case. In contrast, with the difference-based approach, it is very hard to understand that those changes are derived from the same refactoring transformation since they are distributed in textual differences in five source files.

A part of code differences in this case is shown in Fig. 8. Figure 8 shows the diff result of `NextPuyoField.java` (one of children of `AbstractPanel`) and source code `AbstractPanel.java` that appeared after the refactoring session. During the refactoring transformation, code fragments (a) were moved to `AbstractPanel` which became the parent class of `NextPuyoField`. Like the example shows, most difference hunks composing the refactoring application are small. The four subclasses originally have the similar structure and the same kind of code changes were repeated in those files. The similarity of code changes can be a clue for detecting the `EXTRACT SUPERCLASS`. However, the changes are scattered in multiple files and too small, and thus the participant easily overlooked them.

6.2.7 Case 7: Difference-Based Approach

**Refactored:** `PULL UP METHOD`  
**Detected:** `SUBSTITUTE ALGORITHM` (IA)  
**Cause:** Out-of-order interclass code changes

This refactoring transformation pulled up descriptions in two methods (`DisturbancePuyo.init()` and `NormalPuyo.init()`) to the common parent class named `Puyo`. However, the participant who used the difference-based approach answered as `SUBSTITUTE ALGORITHM` in
7. Discussion

This section discusses results of the experiment, causal analyses, and case studies and describes several threats to validity.

7.1 Findings and Remarks

In the difference-based approach, the three causes (“Tangling multiple code changes”, “Shadowing code changes”, and “Out-of-order interclass code changes”) were observed. As observations in Cases 3 to 7 in Sect. 6.2 imply, snapshot-based differences are still insufficient to detect refactoring transformations if their tasks involve the tangling, shadowing, and out-of-order problems. As precedent studies claimed, the operation-based approach using fine-grained operations has a beneficial effect on alleviation of the tangling and shadowing problems [6]. Moreover, replaying edit operations helps to understand the order of code changes. This might bring results that the participants could easily identify code changes performed at the same time even if they straddle multiple source files, as shown in Cases 6 and 7. Here, recent VCSs enable us to know which source files were changed at the same time (within the same commit). However, they cannot completely solve the out-of-order problem since a single commit does not keep information on the change order of multiple files.

Here, we consider how existing difference-based tools can be improved based on the experimental results. Since several kinds of refactoring transformations involve move of code fragments, there are similar (or identical) code fragments in different hunks, as shown in Figs. 6 and 8. To find similar code fragments, code clone detection tools can be used. Unless additional changes are performed to the moved code, clone detection would work well for detecting refactoring transformations with moving code, such as Extract Method. However, the tangling and shadowing problems often pose further changes to the moved code, and the similarity degrades in real refactoring situations. For example, the extracted method was modified by following refactoring transformations in Fig. 6. (Mass is renamed into Cell, and several expressions are modified.) Therefore, we have to note that detecting similar code differences is difficult even with clone detection techniques in such cases. Moreover, such move of code fragments is often beyond the boundary of source files, as shown in Fig. 8. Therefore, showing code differences only in a single file might result in overlooking significant information.

In addition, time information of code changes would be also valuable when refactoring transformations are performed intensively in a short period of time. It would be more effective particularly when the operations straddle multiple source files as shown in Cases 6 and 7. However, current textual diff tools cannot show the time when each of code changes was performed since they do not record such fine-grained time information. The authors consider time information attached to each hunk would help us know associations among multiple separated hunks derived from the same refactoring. To achieve that, operation recording and attaching the time information to each hunk are required, though the replaying code changes is not a must in this case.

Lastly, the difference-based approach cannot prevent the shadowing problem in principle. It is because textual differences do not contain required information as shown in Case 5. If investigating canceled operations is required, the operation-based approach should be adopted before the development gets started.

The operation-based approach is likely to address all the three problems in the difference-based one. This improvement mainly depends on the granularity of code changes in the authors’ opinion. If a developer or maintainer performs a commit after every code change (under a single change reason), the stored “fine-grained” code changes with their commit times could be close to edit operations. In other words, the two approaches are not distinguished. However, such a commit policy imposes much burden on developers and maintainers. Therefore, this policy can seldom be accepted in real software development.

Unfortunately, the operation-based approach is not perfect, either. It poses new problems caused by “Dividing a single code change into operations” and “Generating compound operations”. The former one means that recorded operations are sometimes too fine-grained for understanding code changes. As shown in Case 1, separating the deletion and insertion in code replacement (i.e., the same intention) causes misunderstanding. This kind of problem has been pointed out in former studies [36]. Techniques to restructure recorded operations (e.g., [7], [11]) might alleviate this problem. On the other hand, the cause “Generating compound operations” is derived from a policy of operation recording tools. OperationRecorder used in the experiment records multiple edit operations as a single compound one, which are related to a single refactoring transformation performed through its menu activation. This is effective for understanding an automated refactoring transformation since the code changes of the transformation are grouped.
and the kind of refactoring is shown as a label. However, this grouping sometimes poses the developers’ overlooking of the refactoring application as shown in Case 2. When too many operations are grouped, the number of operations shown in the Operation History excessively decreases, and the resulting operation tends to be overlooked. Thus, it is harder to detect the boundary of refactoring transformations within the operation history in this case.

Obviously, both of the causes in the operation-based approach relate to the granularity of recorded operations. That is, “Dividing a single code change into operations” is caused by too fine-grained recording, whereas “Generating compound operations” is caused by too coarse-grained grouping. To improve existing operation-based tools in future, recording and showing operations in an appropriate granularity is a key. Note that the best granularity depends on the context of use of operations. In future, we need to explore this issue along with how to use recorded operations. In addition, from the point of view of user interface, the replayer should highlight the operation that makes major code changes to prevent developers from overlooking it.

The number of “Careless mistakes” in the difference-based approach is less than that in the operation-based approach. Though the difference is not statistically significant, the authors consider this result is affected by the difference of burden in participants’ tasks between the two approaches, as the statement in 5.3 implies. To reduce the number of the mistakes, making the unit of replay coarser (like Hattori’s work [12]) and slicing code change history [17], [18] might be effective.

The cause “Lack of knowledge and skills” affected the results in both approaches. If a participant has not enough knowledge to correctly identify the applied refactoring transformations, his refactoring detection cannot be sufficiently supported whichever approach is chosen.

7.2 Automated Refactoring Detection

After the experiment, the authors applied RefFinder[34] to the experimental data to compare performances between the automatic refactoring detection and manual ones. As the result, the tool detected supported kinds of refactorings efficiently. However, the tool does not support several kinds of refactoring, such as SUBSTITUTE ALGORITHM. Moreover, it detected many small refactoring transformations so that the authors could not check correspondence to participants’ refactoring memos that represent the performed refactoring transformations much coarser.

7.3 Threats to Validity

A part of incorrect answers in the experiment is derived from our experimental design, especially cases of “Lack of knowledge and skills”. All the participants and developers who wrote the original source code in the experiment are students. Moreover, they used their respective favorite diff tools in the experiment. The results might depend on what kind of tools were used and which options were specified. For example, the context format of the GNU diff tool can show not only changed lines but also unchanged ones. This might facilitate their understanding of the context of code changes.

In addition, how to prepare programs might become a threat. In the experiment, all the programs were the same kind of game applications and their sizes are not large. This is because the operation-based approach requires edit-operation history data recorded in the past development or maintenance. Unfortunately, existing open source projects do not provide such data. Thus, the authors prepared it from scratch. As a result, the data size is not large. The authors consider that the experimental results are reasonable since the observed refactoring transformations are not specific to game applications and general in various kinds of applications. Moreover, in realistic development and maintenance, a refactoring transformation is applied within a limited scope of source code such as a package or module since it must preserve the observable behavior of the code. However, small development in single kind applications might limit kinds of refactoring transformations applied to their code. Therefore, a large number of experiments with various kinds of programs are required to generalize the experimental results.

In the difference-based approach, committing massive code changes often exacerbates the three problems mentioned in Sect. 1. It is natural that many code changes performed together get tangled and shadowed each other and let the change order lost. In our experiment, 1 to 10 refactoring transformations were performed in each session, and this is not unrealistic. Görg’s study [21] shows that a certain number of sessions contain tens of refactoring transformations.

There is still room for argument about how the participants detect refactoring transformations. In the experiment, they in advance knew all changes are refactoring transformations and could select their answers from the given refactoring list. Therefore, they did not answer an unexpected kind of refactoring. This regulation explicitly had an impact on experimental results of participants’ correct or incorrect answers. In addition, the authors determined whether the participants’ answers are correct based on the kind of refactoring. Therefore, how the participants understood the details of refactoring transformations could not be evaluated. To remove such a threat, the authors have to conduct another experiment with a variety of code changes including refactoring transformations, and consider how understanding refactoring transformations should be evaluated.

Another non-negligible threat is related to the results of the causal analyses in this paper. As mentioned in 6.1.1, they are based on the criteria authors determined and contains some subjective view on incorrect refactoring detections. The authors tried to remove subjectivity by formalizing the cause estimation and conducting informal interviews. However, the subjectivity cannot be completely removed. To alleviate this threat, the authors had to check
whether each of the estimation results fit the respective participant’s opinion while his memory was fresh.

8. Conclusion

This paper described a comparative study of detection of refactoring transformations between the operation-based approach using fine-grained code changes at the edit-operation level and difference-based approach using code changes at the snapshot-based level. Two causes of incorrect detection are specific to the operation-based approach, whereas three causes are specific to the difference-based approach. In addition, this paper showed seven concrete examples of mistakes that were actually observed in the experiment. The authors believe the experimental results and findings contribute to future studies in terms of improving techniques and tools that support for detecting past refactoring.

Future work includes experiments with a broad range of code changes, not limited to refactoring transformations, under more sophisticated design. Moreover, the authors are planning to do further investigations of effectiveness of the operation-based approach. The current replay tools should be improved as well.

Acknowledgments

This work was partially sponsored by the Grant-in-Aid for Scientific Research (26730042, 15H02685) from the MEXT/JSPS.

References

Takayuki Omori is a lecturer at the Department of Computer Science, Ritsumeikan University. He obtained his Ph.D. in Engineering from Ritsumeikan University in 2008. From September 2013 to March 2014, he was a visiting assistant professor at the University of British Columbia. His current research interests include software evolution, and fine-grained code changes in software development.

Katsuhisa Maruyama received his B.E. and M.E. degrees in electrical engineering and Ph.D. in information science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is a professor of the Department of Information Science and Engineering, Ritsumeikan University. He has worked for NTT (Nippon Telegraph and Telephone Corporation) and NTT Communications Corporation before he joined Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI). His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE Computer Society, ACM, IPSJ, and JSSST.