2014 6th International Workshop on Empirical Software Engineering in Practice

What Kinds of Refactorings are Co-occurred? An Analysis of Eclipse Usage Datasets

Tsubasa Saika[1], Eunjong Choi[1], Norihiro Yoshida[2], Akira Goto[1], Shusuke Haruna[1], Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University, Japan
{t-saika, ejchoi, a-gotoh, haruna, inoue}@ist.osaka-u.ac.jp
[2]Graduate School of Information Science, Nagoya University, Japan
yoshida@ertl.jp

***Abstract*—Refactoring is the process of changing the internal structure of software without changing its external behavior. So far, numerous tools have been proposed on that refactoring support. However, it is difficult for existing refactoring support tools to support consecutively co-occurred refactorings. To alleviate this problem, a tool which supports consecutively co-occurred refactorings is required. In order to develop a tool support of frequently co-occurred refactorings, this study investigated the frequency of co-occurred refactorings by analyzing usage data of Eclipse. As a result of investigation, it turns out that refactoring pairs (Move, Rename), (Rename, Rename) and (Extract, Move) were more frequently occurred than other pairs. Furthermore, in order to support co-occurred refactorings based on the results of the investigation, this study investigated the detail of co-occurred refactorings and devised suggestions for improving the automated refactoring features in Eclipse. In the investigation of refactoring pair (Move, Rename), we found that program elements were moved and subsequently the names of those elements were renamed. According to this finding, we suggest the need of a tool for supporting this refactoring by recommending appropriate rename for moved elements.**

I. Introduction

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior [1] [2]. Although refactoring is a promising way to improve the maintainability of software, manual refactoring is not only time-consuming but also error-prone [3]. Nowadays, several modern IDEs (e.g., Eclipse, IntelliJ IDEA) support many refactoring that rename, move, or split program elements including methods, classes, and packages. The software engineering community also proposed tool supports for refactoring [4]–[6].

In order to improve refactoring tools, prior studies have been done on the usage instances of the existing refactoring tools [7]. These studies show that developers use those refactoring tools mostly for simple refactorings (e.g., `Rename`, `Extract Local Variable`, `Move`), and often repeat the same kinds of refactorings. This repetition phenomenon provides evidence that the existing tools force developers to repeatedly select, initiate and configure refactoring dialogs, and implies the need of a tool that allows developers to perform those refactorings at one time. On the other hand, refactoring tools also need to support seamless transition among different kinds of refactorings that frequently co-occurred. However, existing tools basically force developers to perform those kinds of refactorings individually. By supporting those co-occurred refactorings, software development efficiency should be improved. In order to realize seamless transition among different kinds of refactorings, a number of usage datasets of IDE should be investigated to know what kinds of refactorings frequently co-occurred [8].

In this study, we investigate the frequency of co-occurred refactoring by analyzing usage data of Eclipse from *the Users* and the *Mylyn* datasets (see Section II), and then discuss future advancement of the refactoring feature of Eclipse based on the investigation result.

The main contribution of this paper is as follows:

- We revealed that the refactoring pairs (`Move`, `Rename`), (`Rename`, `Rename`) and (`Extract`, `Move`) were more frequently occurred than other pairs.
- We performed manual analysis of the instances of above refactoring pairs. As a result, in the case of refactoring pair (`Move`, `Rename`), `Rename` is occurred to change the names of program element that were moved previously. Also, in the case of refactoring pair (`Rename`, `Rename`), `Rename` refactorings are performed together regardless of types of targeted program elements in many cases. In the case of refactoring pair (`Extract Interface`, `Move`), extracted interfaces are moved into other packages in many cases.
- Based on the result of manual analysis, we suggested the directions for the improvement of the refactoring feature of Eclipse. For the refactoring pair (`Move`, `Rename`), recommending appropriate new name for moved elements should be considered. Also, the feature of simultaneous renaming should be improved for the refactoring pair (`Rename`, `Rename`).

II. Analyzed Datasets

To investigate refactoring tool usage, this study selected two usage datasets of Eclipse named *Users* and *Mylyn*. They are selected because they include usage logs of refactoring features in Eclipse. Table I shows a summary of the datasets.

The first dataset is called *Users* which contains the refactoring history of 41 developers from July 2005 to September 2005. It contains total 3494 refactoring instances that were comprised of 22 types of refactoring.

The second dataset is called *Mylyn* which contains the refactoring history of 8 developers from February 2006 to

978-1-4799-6666-0/14 $31.00 © 2014 IEEE
DOI 10.1109/IWESEP.2014.15

TABLE I. STATISTICS OF THE DATASETS

Name	# Developers	Period	# Refactoring Instances	# Types of Refactoring
Users	41	Jul. 2005-Sep. 2005	3494	22
Mylyn	8	Feb. 2006-Aug. 2009	4637	19

TABLE II. REFACTORINGS IN THE *Users* DATASET

Refactoring Type	# Instances	Proportion
`Rename`	1992	57.0%
`Extract Local Variable`	449	12.9%
`Extract Method`	305	8.7%
`Move`	180	5.2%
`Inline`	174	5.0%
`Promote Local Variable`	95	2.7%
`Extract Constant`	59	1.7%
`Modify Parameters`	40	1.1%
`Pull Up`	36	1.0%
`Convert Local To Field`	29	0.8%
`Convert Anonymous To Nested`	29	0.8%
`Introduce Parameter`	25	0.7%
`Extract Interface`	24	0.7%
`Change Method Signature`	16	0.5%
`Move Static Member`	12	0.3%
`Convert Member Type to Top Level`	8	0.2%
`Encapsulate Field`	8	0.2%
`Use Supertype`	6	0.2%
`Externalize Strings`	4	0.1%
`Change Type`	1	0.0%
`Generalize Type`	1	0.0%
`Infer Type Arguments`	1	0.0%
Total	3494	100.0%

August 2009. It contains total 4637 refactoring instances that were comprised of 19 types of refactoring.

The details of each dataset are described in the following subsections.

A. Users Dataset

Users dataset contains refactoring histories of the 41 developers who uses Eclipse for developing. It is originally collected by Gail Murphy at el. [9]. Developers in this dataset work at different organizations with various backgrounds. We expect this dataset shows general tendency of software development since the developers are not specialists of refactoring.

The format of histories in this data is an XML file that is created separately for each commit by the Mylyn monitor tool. This includes histories of configuration changes of development environment, implementations of commands, editions of source codes, and other actions. For each refactoring recorded in this dataset, its types and when it was performed can be found. However, the information about program elements affected by the refactorings was unavailable in this dataset.

Table II shows the number of instances of refactoring performed in the *Users* dataset and the proportion to the total in descending order to the numbers of instances. As shown in this table, the most frequently occurred refactoring in *Users* dataset is `Rename` which changes a name of a program element and its references. It is commonly known as the most popular type of refactoring.

B. Mylyn Dataset

Mylyn dataset contains refactoring histories of the developers who maintain the Mylyn monitor tool, the task management plug-in for Eclipse. Developers in this dataset may not be experts of refactoring. However, we assume that they know much about refactoring features of Eclipse than general developers because they develop Mylyn monitor tool.

TABLE III. REFACTORINGS IN THE *Mylyn* DATASET

Refactoring Type	# Instances	Proportion
`Rename`	2401	51.8%
`Move`	691	14.9%
`Extract Method`	305	6.6%
`Extract Local Variable`	254	5.5%
`Extract Constant`	245	5.3%
`Move Static Member`	235	5.1%
`Change Method Signature`	191	4.1%
`Inline`	110	2.4%
`Convert Member Type to Top Level`	44	0.9%
`Infer Type Arguments`	30	0.6%
`Extract Interface`	29	0.6%
`Encapsulate Field`	29	0.6%
`Convert Anonymous To Nested`	23	0.5%
`Extract Superclass`	16	0.3%
`Promote Local Variable`	15	0.3%
`Pull Up`	14	0.3%
`Push Down`	3	0.1%
`Use Supertype`	1	0.0%
`Introduce Parameter`	1	0.0%
Total	4637	100.0%

We use database which is originally collected by Emerson Murphy-Hill at el. [7]. This database contains details of each refactoring instance such as type, date and time, and affected program elements. In addition to this, we also check Git repository of Mylyn project for the detailed analysis.

Table III shows the number of instances of refactoring in the *Mylyn* dataset and the proportion to the total. This table is also sorted according to the numbers of refactoring instances.

III. INVESTIGATION STEP

This section illustrates our investigation in order to devise suggestions for developing tools that support frequently co-occurred refactorings. The investigation is comprised of two steps. In the first step, we investigated the number of instances for all combinations of co-occurred refactorings from the datasets. In the second step, we further investigated the instances by manual analysis of the commit logs as well as source code.

Step1. Investigating frequencies of co-occurred refactorings

In this step, we identified consecutively co-occurred refactorings based on time stamps which indicate when each refactoring was performed. Then, we investigated frequency for all combinations of co-occurred refactorings found in the datasets. In this step, if refactorings were occurred within 90 seconds, we defined them as co-occurred refactorings. We set up 90 seconds as threshold of determining co-occurred refactorings based on the our preliminary experiment.

In the preliminary experiment, we investigated repeatedly performed different kinds of refactoring in relation to time-interval. We found that it seems to be too short to complete the refactoring in less than 60 seconds and also longer time-interval is likely to include falsely refactorings. We also found that types of refactorings appeared within 60, 90, and 120 seconds are similar to each other. Therefore, we selected 90

seconds as the time-interval because it is long enough to extract about 60 presents of all refactorings consecutively performed in the data sets.

However, there is a possibility the refactorings consecutively performed may not be refactorings which the developer performed as one task. To alleviate this problem, we further investigated relationship between co-occurred refactorings by analyzing their details in the next step.

Step2. Investigating detailed work of frequently co-occurred refactoring

In this step, we investigated the detailed works of frequently co-occurred refactorings. We analyzed how program elements, the targets of co-occurred refactorings, were modified. The goal of this investigation is to identify how developers changed program with the combinations of refactorings and to consider support for performing those refactorings. However, this investigation was carried out only for the *Mylyn* dataset because information such as the target of refactoring has not been included in the *Users* dataset as mentioned in Section II-A. We analyzed targets of co-occurred refactorings by investigating 10 instances of detailed refactoring histories for each combination of the top 10 high frequencies. Moreover, we examined the targets of co-occurred refactorings are whether the same or whether or not related. A combination of co-occurred refactorings is determined to be related when their targets have similar name or belong to the same class or package.

IV. Results

This section shows results of our investigation. Section IV-A explains the results of the investigation on frequencies of co-occurred refactorings. Then, Section IV-B explains the results of the investigation on the detailed work of frequently co-occurred refactorings.

A. Frequently Co-occurred Refactorings

This section describes the results of **Step1** which investigates frequencies of co-occurred refactorings. Table IV and V shows the investigation result of the *Users* dataset and *Mylyn* dataset respectively. In both tables, "Refactoring Type" column shows types of co-occurred refactorings. "# Instances" column represents the number of refactoring instances of each combinations. Note that these tables are sorted in descending order of the number of refactoring instances and only show the top ten frequently occurred pairs in terms of frequency, because we are only interested in more frequently co-occurred refactorings. As shown in both tables, in most cases, refactorings were subsequently occurred twice. In fact, the numbers of instances of any combinations which are consisted of three or more refactorings are less than five times. These two tables show similar combinations of co-occurred refactorings. That is, most combinations of co-occurred refactorings contain at least one of `Rename`, `Move`, and `Extract`. These refactoring types are also shown as the most frequently occurred refactoring types in Table II and III.

In terms of *Mylyn* dataset, this dataset includes information about targets of all refactoring instances. Therefore, we classified `Rename` refactorings into several types: `Rename Type(Class)`, `Rename Package`, `Rename Field`, `Rename Method`, and `Rename Local Variable`. Table VI shows the result of this classification. This table also shows combinations which are occurred more than 10 times. The result of this classification shows that `Rename` refactorings are performed together regardless of types of modified program elements in many cases.

TABLE IV. Frequently Co-occurred Refactorings in the *Users* Dataset

Refactoring Type		# Instances
Refactoring 1	Refactoring 2	
`Extract Method`	`Rename`	35
`Extract Local Variable`	`Rename`	22
`Extract Local Variable`	`Extract Method`	19
`Rename`	`Extract Method`	17
`Rename`	`Move`	15
`Extract Method`	`Inline`	14
`Extract Local Variable`	`Inline`	14
`Inline`	`Extract Local Variable`	14
`Extract Local Variable`	`Promote Local Variable`	12
`Move`	`Rename`	12

TABLE V. Frequently Co-occurred Refactorings in the *Mylyn* Dataset

Refactoring Type		# Instances
Refactoring 1	Refactoring 2	
`Move`	`Rename`	88
`Rename`	`Move`	62
`Extract Interface`	`Move`	12
`Extract Constant`	`Rename`	11
`Extract Constant`	`Move`	10
`Extract Method`	`Rename`	9
`Extract Method`	`Extract Local Variable`	6
`Extract Local Variable`	`Rename`	6
`Rename`	`Change Method Signature`	5
`Inline`	`Rename`	5

TABLE VI. Frequently Co-occurred Refactorings in the *Mylyn* Dataset in the Case Where Targets of Refactoring are Classified for `Rename` and `Move`

Refactoring Type		# Instances
Refactoring 1	Refactoring 2	
`Rename Type`	`Move`	30
`Move`	`Rename Type`	28
`Rename Field`	`Rename Method`	26
`Rename Method`	`Rename Field`	22
`Move`	`Rename Package`	19
`Rename Type`	`Rename Method`	16
`Rename Method`	`Rename Type`	16
`Rename Type`	`Rename Field`	13
`Rename Field`	`Rename Type`	13
`Extract Interface`	`Move`	12

B. Detailed Co-occurred Refactoring Instances

In this section, we explain the results of **Step 2**. which investigated the detailed works for frequently co-occurred refactorings of the *Mylyn* dataset. We invetigated the details of work of the top 10 most frequently co-occurred refactorings. We checked targets of refactoring and classified them into three types (same, related, and others). In some cases, targets of co-occurred refactorings are the same program element. In other cases, targets are determined as related program elements if they have similar name or belong to the same package or class. In the rest cases, we could not identify the relationship between targets.

Table VII shows the result of investigation on the detailed work for frequently co-occurred refactoring. The order of

TABLE VII. A SUMMARY OF INVESTIGATION INTO DETAIL WORKS OF FREQUENTLY CO-OCCURRED REFACTORINGS IN THE *Mylyn* DATASET

Refactoring Type		# Performed	# Investigated	# Same	# Related
Refactoring 1	Refactoring 2				
Move	Rename Type	58	10	4	1
Rename Field	Rename Method	48	10	0	5
Rename Type	Rename Method	32	10	0	4
Move	Rename Package	29	10	1	7
Rename Type	Rename Field	26	10	0	3
Move Static Member	Rename Field	15	10	6	0
Extract Interface	Move	14	10	7	1
Rename Local Variable	Rename Field	12	10	0	6
Move	Move Static Member	12	10	0	2
Move	Rename Field	12	10	0	0

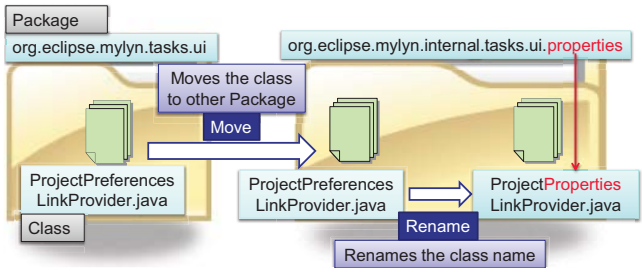

Fig. 1. An Instance of Detailed Work for (Move,Rename) from the Mylyn Dataset

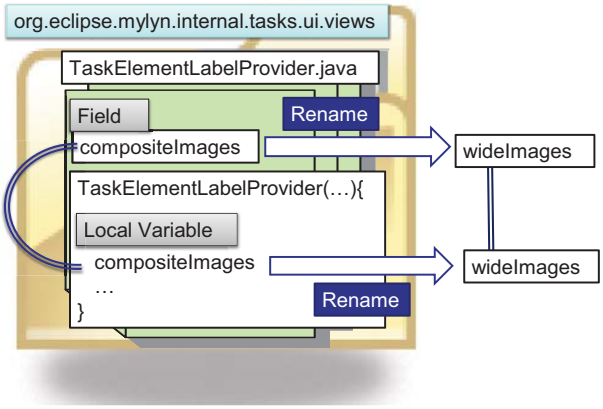

Fig. 2. An Instance of Detailed Work for (Rename,Rename) from the Mylyn Dataset

each combination is ignored. Same as shown in the Table VI in Section IV-A, Rename and Move refactorings are classified according to their applied types. The numbers of performed refactoring, investigated refactoring, same targets, and related targets are shown in Table VII. More than half of all investigated refactoring-pairs are same or related.

Hereinafter, we summarize an overview of the detailed works for the most frequently co-occurred refactorings: (Move, Rename), (Rename, Rename), and (Extract, Move).

1) (Move, Rename)

About refactoring pair of Move and Rename, it turned out that Rename are occurred to change the names of program element that were moved previously. In the case where targets of Move and Rename refactorings were the same program element, program elements such as packages, classes, and fields were renamed immediately after they were moved. In the case where targets of Move and Rename refactorings had similar name or belong to the same package or class, the names of classes and packages were changed after program elements such as classes fields, and methods were moved into it.

Figure 1 depicts an instance of detailed work of Move and Rename Type refactorings simultaneously performed in Mylyn project. This instance is found from a commit recorded in the Git repository of Mylyn project[1]. In this figure, a class named ProjectPreferences-LinkProvider was moved from package named org.eclipse.mylyn.tasks.ui to package named org.eclipse.mylyn.internal.tasks.ui.properties. Moreover, its class name is changed from ProjectPreferencesLinkProvider to ProjectPropertiesLinkProvider using Rename Type refactoring. Its new name includes a word properties (shown in red font) come from its new package named org.eclipse.mylyn.internal.tasks.ui.properties.

2) (Rename, Rename)

About refactoring pair of Rename and Rename, it turned out that Rename refactorings are performed together regardless of types of targeted program elements in many cases. In addition, those program elements tend to have similar names and some relationship. For example, a local variable of constructor, setter method, or getter method and a relating field variable are often renamed together.

Figure 2 shows an instance where Rename Field, Rename Local Variable refactorings were used together in Mylyn project. This instance is found from a commit recorded in the Git repository of Mylyn project[2] In this figure, a field variable named compositeImages is renamed to wideImages

[1]http://git.eclipse.org/c/mylyn/org.eclipse.mylyn.tasks.git/tree/org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal/tasks/ui/properties/ProjectPropertiesLinkProvider.java?id=a4907cc0d72cdc9e6eb7c1e62234edd01214b072

[2]http://git.eclipse.org/c/mylyn/org.eclipse.mylyn.tasks.git/diff/org.eclipse.mylyn.tasks.ui/src/org/eclipse/mylyn/internal/tasks/ui/views/TaskElementLabelProvider.java?id=62f11037485847c03b5e9f7c309ac1f25ee9a242

using `Rename Field` refactoring. Moreover, a local variable also named compositeImages is similarly renamed to wideImages using `Rename Local Variable refactoring`.

3) (`Extract Interface`, `Move`)

About refactoring pair of `Extract Interface` and `Move`, it turned out that extracted interfaces are moved into other packages in many cases.

V. Discussion

As a result of the investigation, the most frequently co-occurred refactoring is revealed. Based on the investigation results, this section explains our suggestions for improving refactoring features of Eclipse as follows.

1) (`Move`, `Rename`)

Based on the results of the investigation, a refactoring tool needs to supports developers to consecutively perform `Rename` refactoring after `Move` refactoring. Currently, Eclipse does not provide interaction between `Move` and `Rename` refactorings. Therefore, developers need to perform `Move` and `Rename` refactoring separately and configuration dialog appear twice even though developers often do not configure refactoring tools [7]. For supporting this combinations of refactorings, making a new menu which simultaneously performs `Move` and `Rename` refactorings in Eclipse can be useful. If the name of the program element can be changed in the dialog of `Move` refactoring, it is unnecessary to perform `Rename` refactoring separately. Recommending appropriate new name for moved elements is also can be useful for supporting this combinations.

2) (`Rename`, `Rename`)

For supporting these refactorings, tool should support rename several program elements regardless of their types can be useful for supporting this refactoring pair. In the current specification of refactorings of Eclipse, for only the `Rename` refactoring intended for class, elements with the similar names can be automatically renamed together. However, there is no such support for `Rename` refactoring intended for other types of program elements such as fields or methods. Therefore, it is necessary to support them in the same way.

3) (`Extract Interface`, `Move`)

Based on the investigation results, a refactoring tool needs to supports developers to consecutively perform `Move` refactoring after `Extract Interface` refactoring. In the current specification of refactorings of Eclipse, `Move` refactoring of a java file can be carried out easily with drag-and-drop from the Package Explorer. Therefore, it is considered that sufficient support is being made already for this combination. But if the location to create new Java file can be selected in the dialog of `Extract Interface` refactoring, it is unnecessary to perform `Move` refactoring separately.

Some limitations exist in this study because we investigated the data sets in the period of 2005 to 2009. Refactoring features in version of Eclipse at the time would be different from those of the latest version. Therefore, we need to investigate a recent data set to examine how developers have changed the way of refactoring. However, the usability problems revealed in this study still exists in the current version of Eclipse.

VI. Related Work

Murphy et al. reported on usage data collected from 41 Java software developers using Eclipse, providing a glimpse into their work habits [9]. According to their report, the most common refactoring commands were `Rename`, `Move`, and `Extract`. Murphy-Hill and Black observed 11 programmers perform a number of `Extract Method` refactorings, and then revealed room for two types of improvements to `Extract Method` refactoring tools [10]. First, to prevent a large number of errors in the first place, programmers need support in making a valid selection. Second, to help programmers successfully recover from violated preconditions, programmers need expressive, distinguishable, and understandable feedback that conveys the meaning of precondition violations. Based on the revealed room, they presented three tools that help programmers avoid selection errors and understand refactoring precondition violations. Also, Murphy-Hill et al. revealed that programmers often repeat the same types of refactorings [7]. According to their investigation, about 40% of refactorings performed using tool occur in batches. Also, they found that about 90% of configuration defaults in refactoring tools are not changed when programmers use the tools. We investigated repeatedly performed different kinds of refactorings based on the usage instances of the existing refactroing tools.

Vakilian et al. collected a set of interaction data from about 1268 hours of programming using our minimally intrusive data collectors [11]. Their quantitative data shows that programmers prefer lightweight methods of invoking refactorings, usually perform small changes using the refactoring tool, proceed with an automated refactoring. They found that programmers use predictable automated refactorings even if they have rare bugs or change the behavior of the program. Vakilian and Johnson proposed an approach to discovering the usability problem on the refactoring feature of Eclipse by collecting alternate refactoring paths (i.e., a sequence of user interactions that contains cancellations, reported messages, or repeated invocations of the refactoring tool) [12]. Their approach revealed 15 usability problems, 13 of which were previously unknown. We would like to investigate whether or not our suggestion in Section V is able to reduce the usability problem on the refactoring feature of Eclipse.

Kim et al. presented a field study of refactoring benefits and challenges at Microsoft [13]. Their survey found that the refactoring definition in practice is not confined to a rigorous definition of semantics-preserving code transformations. We need to investigate refactoring instances in industrial projects as well as in OSS projects. Also, Kim et al studied API-level

refactorings and bug fixes in three large open source projects, totaling 26523 revisions of evolution [14]. One of their findings is that an increase is confirmed in the number of bug fixes after API-level refactorings. We hope to investigate whether or not co-occurred refactorings affect bug fixes.

Several studies have been done on seamless support for refactoring [5], [6], [15], [16]. BeneFactor [6], GhostFactor [16] and WitchDoctor [5] monitor code modification on the fly, and then dynamically utilize the monitoring result for refactoring support. They detect the beginning of refactoring, and pro-actively recommend a series of code transformation to complete the refactoring. Lee et al. proposed Drag-and-Drop Refactoring, which allows programmers to directly manipulate program elements (e.g., variables, expressions, statements, methods) in IDE [15]. We suggested seamless support for co-occurred refactorings in Section V.

VII. Summary

To develop a tool that supports different kinds of co-occurred refactorings, this study preliminary investigates repeatedly performed different kinds of refactorings based on the developers' refactoring tool usage.

In summary, the most co-occurred refactorings are (`Move`, `Rename`), (`Rename`, `Rename`), and (`Extract`, `Move`). Moreover, based on the investigation of the detailed works for these co-occurred refactorings, we suggested several improvements of refactoring feature in Eclipse for supporting them. In order to support the refactoring pair (`Move`, `Rename`), the refactoring feature needs to support developers to simultaneously perform this combination. For the refactoring pair (`Rename`, `Rename`), the tool must be able to perform multiple `Rename` refactorings together regardless of types of their targeted program elements.

As future work, it is necessary to clarify benefits of supporting co-occurred refactorings. We expect that supporting co-occurred refactorings certainly improves software development efficiency. However, it has not been investigated yet for the effect on occurrence of bugs. Also, we are going to analyze additional software systems to achive generality of our findings. In addition, we are interested in comparing batched refactorings between legacy software projects including a number of bad smells and well-maintained software project. We expect that such legacy project tends to cause higher number of batched refactorings compared to well-maintained projects. Especially, we would like to investigate whether or not combined bad smells [17] and/or the amount of code clones [18] increase the number of batched refactorings. Also, industrial filed study [13] on batched refactorings is an interesting challenge. Finally, we would like to develop a tool for supporting refactoring according to the findings from the investigations.

VIII. Acknowledgment

We express our great thanks to Dr. Gail C. Murphy, Dr. Mik Kersten and Dr. Leah Findlater at the University of British Columbia for providing the *Users* dataset, and Dr. Emerson Murphy-Hill at North Carolina State University, Dr. Chris Parnin at the Georgia Institute of Technology and Dr. Andrew P. Black at Portland State University for providing the *Mylyn* dataset. This work was supported by JSPS KAKENHI Grant Numbers 25220003 and 26730036.

References

[1] M. Fowler, *Refactoring:Improving the Design of Existing Code*. Addison Wesley, 1999.

[2] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[3] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Proc. of SCAM*, 2012, pp. 104–113.

[4] R. Tairas and J. Gray, "Increasing clone maintenance support by unifying clone detection and refactoring activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, 2012.

[5] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. of ICSE*, 2012, pp. 222–232.

[6] X. Ge, Q. L. DuBose, and E. R. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. of ICSE*, 2012, pp. 211–221.

[7] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, 2012.

[8] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson, "A compositional paradigm of automating refactorings," in *Proc. of ECOOP*, 2013, pp. 527–551.

[9] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the Eclipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, 2014.

[10] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proc. of ICSE*, 2008, pp. 421–430.

[11] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proc. of ICSE*, 2012, pp. 233–243.

[12] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proc. of ICSE*, 2014, pp. 1106–1116.

[13] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. of FSE*, 2012, pp. 1–11.

[14] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proc. of ICSE*, 2011, pp. 151–160.

[15] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop refactoring: Intuitive and efficient program transformation," in *Proc. of ICSE*, 2013, pp. 23–32.

[16] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. of ICSE*, 2014, pp. 1095–1105.

[17] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: an empirical study," in *Proc. of ICSE*, 2013, pp. 682–691.

[18] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.