Proactive Clone Recommendation System for Extract Method Refactoring

Norihiro Yoshida†, Seiya Numata*, Eunjong Choi‡, and Katsuro Inoue*
†Nagoya University, Japan, yoshida@ertl.jp
*Osaka University, Japan, {s-numata, inoue}@ist.osaka-u.ac.jp
‡Nara Institute of Science and Technology, Japan, choi@is.naist.jp

***Abstract*—"Extract Method" refactoring is commonly used for merging code clones into a single new method. In this position paper, we propose a proactive clone recommendation system for "Extract Method" refactoring. The proposed system that has been implemented as Eclipse plug-in monitors code modifications on the fly. Once the proposed system detects an "Extract Method" refactoring instance based on the analysis of code modifications, it recommends code clones of the refactored code as refactoring candidates. The preliminary user study shows that the users are able to refactor a greater number of clones in less time compared to a code clone analysis environment GemX.**

***Index Terms*—Extract Method Refactoring, Code Clone, Recommendation System**

I. Introduction

One of the causes of increases in software development costs is the existence of code clones within the source code. Code clones refer to identical or similar code fragments within the source code, and these are mainly generated by copying and pasting existing code fragments [1], [2]. Typically, the two code fragments that make up the code clones are referred to as a code clone pair, and the set of all of the code clones is referred to as a code fragment set.

One approach of reducing the cost to maintain code clones is clone refactoring [3], [4]. Clone refactoring performs "Extract Method" and merges the code clones within the same code clone set into a single method [5]. Through suitable clone refactoring, it is possible to prevent the costs associated with code clone maintenance. Thus far, several approaches have been proposed for supporting clone refactoring by extracting code clones that are opportunities for "Extract Method" refactoring from the source code [6], [7]. The clone refactoring support approaches need to not only propose which code clones should be refactored as a priority but also support refactoring during a particular period based on the current activity of the developer. The reason for this is that by recommending refactoring candidates based on the current activity of the developer, the clones to be refactored will be fresh in the mind of the developer and hence the developer can perform the refactoring more efficiently. Furthermore, code fragments are recommended as the refactoring candidates when the process of unit tests for the recommend code fragments is finished and a developer works for other code fragments, the developers memory of the recommended code fragments may be ambiguous. Moreover, as, when refactoring these code fragments, it is necessary to redo the unit tests for confirming the appropriateness of the refactoring, this lead that the maintenance cost related to clone refactoring will increase [8], [9]. Additionally, developers may carry out "Extract Method" refactoring without any awareness of the existence of code clones. In such a case, there is an issue due to the fact that code clones for which "Extract Method" refactoring should be carried out at the same time may be overlooked. However, as far as we know, in the clone refactoring support proposed thus far, clone refactoring support has been carried out without any consideration given to the work content of the developer. To resolve this problem, in this study, we focus on "Extract Method" refactoring and construct a system for supporting code clone refactoring during a suitable period, depending on the work content of the developer. More specifically, this involves monitoring the source code editing work by the developer on the Eclipse. Then, when it is detected that the developer is performing "Extract Method" refactoring, present the code fragment on which this "Extract Method" refactoring was performed to the developer as a code clone and encourage them to consider refactoring at the same time. In this way, the developer is made aware of the existence of the code clone at this point and can consider the refactoring in relation to the presented code clone. In this study, we propose a proactive clone recommendation system for Extract Method refactoring. We also performed a preliminary user study for investigating the effectiveness of clone refactoring support in the proposed system. When we performed a comparative test with *GemX*, which is the GUI of *CCFinderX*, we were able to confirm the effectiveness of the clone refactoring support in the proposed support system.

II. GemX: Code clone analysis environment

The code clone analysis environment *GemX* [10] is the GUI of the token-based code clone extraction tool *CCFinderX* [11], and we were able to analyze the code clones extracted by *CCFinderX*. When providing clone refactoring support using *GemX*, the flow is such that the developer moves away from the IDE during development, and detects code clones from the target projects using *GemX*. Then, they search for "Extract Method" refactoring candidates based on the code clone detection results and then return again to the IDE where the development work was taking place, and consider the clone refactoring. However, as *GemX* detects an extremely high number of code clones, it is not easy for the developer

978-1-7281-2270-0/19/$31.00 ©2019 IEEE
DOI 10.1109/IWoR.2019.00020

to search for the candidates for simultaneous refactoring from detected code clones. Additionally, *GemX* is used by many developers to develop source code, and one problem raised is that it cannot be used on IDE. For this reason, for developers to be able to refactor code clones in parallel to performing development work, they need to interrupt their current development work, and open and operate *GemX*. This is a very laborious work.

III. Proposed system

Many approaches have already been proposed for automatically detecting for code clones from the source code [11]–[13]. However, many of the code clone maintenance work support approaches proposed thus far involve the developers separating themselves from their current work, detecting code clones in relation to the source code, and performing maintenance support. As far as we are aware, there are no approaches to support code clone maintenance work by analyzing the work content. As in source code for the version control system, if the developer is aware of the existence of code clones in source code for which unit tests are complete, and he/she performs clone refactoring, it will be necessary to repeat unit tests. This will significantly increase development costs. Additionally, as the developer performs the development work without grasping the existence of all of the code clones, it is considered possible that he/she may not be aware of all of the code fragments that can be refactored simultaneously.

Therefore, in this study, we propose a system in which, by monitoring the development work of the developer, "Extract Method" refactoring are detected, and the candidates for code clones refactoring are recommended to the developer. In this way, there is no need to separate the current work of the developer from the work to detect code clones, and this promises to enable software maintenance work to be carried out efficiently. In other words, we believe that when a method is extracted, the extracted code block is cloned elsewhere in a system. More generally, our idea is that clones recommended to developers should be dependent on the current activity of each developer.

An overview of the proposed system is shown in Figure 1. The proposed system generally has two functions. Function 1 is a monitoring and analysis function, and function 2 is a code clone recommendation function. Function 1 is realized from step 1 to step 3, and function 2 is realized in step 4 and step 5. We shall now explain these steps in more detail.

A. Keystroke tracking

To monitor the development environment of the developer, in the proposed system, the keystrokes of the developer are tracked. The necessary information when performing "Extract Method" refactoring is one line or more of line differential information. Therefore, the proposed system tracks the keystrokes of the developers and, using said keystrokes, take the line differential every time a line is changed, enabling detection of changes in 1 or more lines. When taking the line differential, we applied the Myers differential detection algorithm [14].

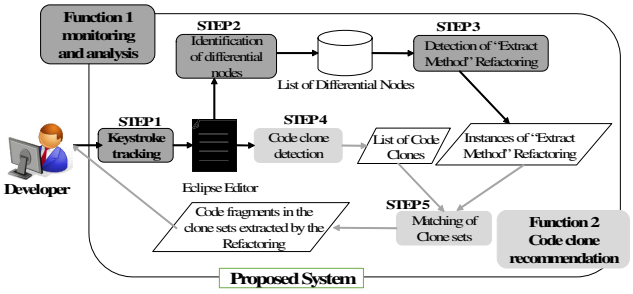

Fig. 1. Overview of the proposed system

B. Identification of differential nodes

In case one or more line is changed in relation to the source code, this differential is mapped to an AST node and identified. We used the Eclipse ASTParser [15] for constructing AST. The differential is constructed from the pair $(delta\_type, ast\_node)$. The value taken from delta_type is either Insert or Delete. Insert expresses that one or more lines have been added, whereas Delete expresses that 1 or more lines have been deleted. AST created using the ASTParser can be reached by implementing the ASTVisitor class. In this way, the node can be identified from AST. Therefore, an AST that matches the line differential identified using the differential detection algorithm can be identified from AST. ast_node is formed from information from the identified AST node, and this includes information such as the node name, node type, node position, and node contents. Here, the node types include information such as MethodDeclaration, MethodInvocation, and VariableDeclarationStatement. To detect "Extract Method" refactoring, as the three items of information, $(Insert, MethodDeclaration)$, $(Insert, MethodInvocation)$, and $(Delete, code\_block)$ are required, in this study we identify only these three types of AST nodes, and save these as a node list.

C. Detection of "Extract Method" Refactoring

After the identification of differential nodes, it is necessary to identify whether the accumulated differential nodes represents "Extract Method" refactoring. If differential nodes satisfy the following condition, the proposed system recognizes that "Extract Method" refactoring is performed.

Condition for "Extract Method" refactoring

$$
\begin{gathered}
(Insert, MethodDeclaration) \\
\wedge (Delete, code\_block) \\
\wedge (Insert, MethodInvocation) \\
where \\
position(code\_block) = \\
position(MethodInvocation) \\
\wedge\ name(MethodDeclaration) = \\
name(MethodInvocation)
\end{gathered}
$$

In other words, with the proposed system, method extraction is detected once a new method is declared, a code fragment is deleted, and a new method invocation statement is added in the position in which the code fragment is deleted. If we look at the above formula, Line 1 $(Insert.MethodDeclaration)$ expresses that a new method has been declared, and the second line $(Delete, code\_block)$ expresses that a particular code fragment has been deleted. The third line $(Insert, MethodInvocation)$ invokes a new method and inserts a statement. However, because this is not enough for us to determine whether "Extract Method" refactoring has performed, further conditions are added and the line 4, 6 $position(code\_block) = position(MethodInvocation)$ determines whether the position of the deleted code fragment and the position where the method invocation statement was inserted are the same. The $name(MethodDeclaration) = name(MethodInvocation)$ in lines 7 and 8 determines whether the newly declared method name and the newly inserted method invocation statement method name are the same. When the above five conditions are satisfied, this is detected as an instance of "Extract Method" refactoring.

D. Code clone detection

The proposed system uses *CCFinderX* in the proposed system to detect code fragment code clones on which "Extract Method" refactoring has taken place [11]. To detect code fragments in the code clones on which "Extract Method" refactoring has performed, it is necessary to detect code clones in source code immediately before "Extract Method" refactoring. It is also necessary, therefore, to detect that "Extract Method" refactoring has started. Thus, in the proposed system, in relation to the timing at which a developer newly creates a method, the possibility is considered that "Extract Method" refactoring will be performed after this; therefore, code clone detection is performed at this time.

E. Matching of Clone sets

Once "Extract Method" refactoring is detected, it is then necessary to detect the code fragment clone set extracted by "Extract Method" refactoring. The code fragment on which the "Extract Method" refactoring took place and the code clone matching the position information can be searched from the *CCFinderX* output results in Section III-D. Then, all of the code clones for the clone set to which this code clone belongs are detected, and the code fragments on which "Extract Method" refactoring was performed at the same time as this clone set are presented to the developer in Eclipse view. As simultaneous refactoring candidates are displayed, the display is split between recently modified code clones within the file (Modified File Clone) and code clones other than those in the modified file (Between Clone Set). This is because the code clones within the file have a higher priority for refactoring than the code clones between the files, and are considered to be easily refactored. Each code clone has a clone set ID, clone information, and line information. The clone set ID is an ID for identifying the clone set to which this code clone belongs, and the clone information expresses information on which file it belongs to and the code clone section offset. The line information expresses the line of the code clone section. Additionally, if the developer selects one code clone from the code clone list, the file that includes that code clone is opened, and that code clone section is highlighted, enabling the developer to easily grasp what code clone it is. Looking at the highlighted code clone, the developer is able to decide whether to perform simultaneous "Extract Method" refactoring.

IV. PRELIMINARY USER STUDY

We conducted a preliminary user study to investigate the effectiveness of the clone refactoring support in the proposed system. In the user study, we compared the results of clone refactoring supports with 8 master course students studying information science using the proposed system and *GemX*, the GUI of *CCFinderX*. All of them have Java programming skill. For the study, we used two datasets from the following two projects written in Java:

- JFreeChart[1] (260 KLOC with 990 classes)
- JUnit[2] (43 KLOC with 449 classes)

JFreeChart is a graph library that allows a user to draw graphs of various statistical charts and functions. JUnit is a framework for automating unit tests. From these projects, we created datasets by selecting methods that are statically called the same number of times and then in-lining these methods In the datasets, the methods that were inlined are defined as targets of "Extract Method" refactoring. The dataset contains three targets for each project.

During the study, we requested participants to perform "Extract Method" refactoring for code clones using the proposed system and *GemX*, respectively, with the datasets. Moreover, we investigated (1) the number of code clones that were refactored and (2) overall time for performing clone refactoring with the datasets. We conducted the study with two phases, the first phase was the code clone refactoring using the proposed system and the second phase was conducted using *GemX*. In the study, we provided one of three candidates for "Extract Method" refactoring and asked participants to identify the other two candidates by using the project system and *GemX*, respectively. Moreover, we did not tell the number of code clones that can be targets of "Extract Method" refactoring.

[1]http://www.jfree.org/jfreechart/
[2]https://junit.org/junit4/

Therefore, the participants found the targets for clone refactoring using the proposed system and *GemX*.

TABLE I
MINIMUM, MAXIMUM AND AVERAGE VALUES OF THE NUMBER OF CODE CLONES (I.E., CODE FRAGMENTS IN EACH CLONE SET WITHIN A PROJECT) THAT WERE REFACTORED

	Proposed system	GemX
Maximum	3	3
Minimum	1	0
Average	2.75	2

TABLE II
MINIMUM, MAXIMUM AND AVERAGE VALUES OF OVERALL TIME FOR PERFORMING CLONE REFACTORING IN SECONDS

	Proposed system	GemX
Maximum	2,040	2,610
Minimum	101	1,080
Average	1043	1,602

The results of user study are shown in Table I and Table II. Table I shows the minimum, maximum and average values of the number of code clones that were refactored during the user study. Table II presents the minimum, maximum and average values of time for performing clone refactoring.

As you can see in the Table I, the average number of code clones (i.e., code fragments in each clone set within a project) that were refactored for the proposed system is 2.75 and *GemX* is 2. This indicates that the proposed system was able to support "Extract Method" refactoring for code clones more effectively than the *GemX*. We also confirmed the significant differences between these numbers by conducting t-test with a confidence level of 0.05. From these results, we can conclude that the proposed system effectively support clone refactoring regarding finding the targets of clone refactoring.

Moreover, as you can see in the Table II, the average time for performing clone refactoring with the proposed system in seconds is 2,040. Whereas, the average time in seconds with the *GemX* is 2,610. This indicates that it takes shorter time to perform clone refactoring with the proposed system than the *GemX*. We also confirmed the significant differences between these time by conducting t-test with a confidence level of 0.05. From these results, we can conclude that the proposed system can support clone refactoring within less time.

V. Threats to Validity

In the preliminary experiment, we assumed that clones are simultaneously refactored by Extract Method refactoring and created the dataset using Inline Method refactoring. We need to investigate how often clones are simultaneously refactored in practice.

Also, we assumed that if several clones are found, all of the clones can be refactored simultaneously. In other words, we assumed that classes containing clones have a shared association with the class to which the method will be moved. We need to investigate what kinds of clones can be supported by the proposed system.

VI. Summary

In this position paper, to analyze the development work carried out on Eclipse, we searched for "Extract Method" patterns, presented the extracted code fragment code clones to the developer, and constructed a system for supporting clone refactoring. When comparing the time taken for clone refactoring and frequency on the proposed system and the code clone analysis environment *GemX*, we could confirm a significant difference. We plan to perform a large-scale user study to show the effectiveness of the proposed system. One of the future works is increasing the number of supported refactoring patterns that can be applied to reducing code clones.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP25220003, JP18H04094 and JP16K16034.

References

[1] B. S. Baker, "Finding clones with dup: Analysis of an experiment," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 608–621, 2007.
[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of ICSM*, 1998, pp. 368–377.
[3] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, Nov 2015.
[4] N. Yoshida, T. Ishizu, B. Edwards, and K. Inoue, "How slim will my system be?: estimating refactored code size by merging clones," in *Proc. of ICPC*, 2018, pp. 352–360.
[5] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[6] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Proc. of METRICS*, 2005, pp. 16:1–16:10.
[7] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proc. of IWSC*, 2011, pp. 7–13.
[8] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Applying clone change notification system into an industrial development process," in *Proc. of ICPC*, 2013, pp. 199–206.
[9] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1008–1026, 2012.
[10] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," in *Proc. of METRICS*, 2002, pp. 67–76.
[11] 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.
[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. of ICSE*, 2007, pp. 96–105.
[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. of ICSE*, 2016, pp. 1157–1168.
[14] E. W. Myers, "AnO (ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.
[15] Program Creek, "Use JDT ASTParser to parse Single Java files." http://www.programcreek.com/2011/11/use-jdt-astparser-to-parse-java-file/.