Move Code Refactoring with Dynamic Analysis

Shuhei Kimura, Yoshiki Higo, Hiroshi Igaki and Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {s-kimura, higo, igaki, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In order to reduce coupling and increase cohesion, we refactor program source code. Previous research efforts for suggesting candidates of such refactorings are based on static analysis, which obtains relations among classes or methods from source code. However, these approaches cannot obtain runtime information such as repetition count of loop, dynamic dispatch and actual execution path. Therefore, previous approaches might miss some refactoring opportunities. To tackle this problem, we propose a technique to find refactoring candidates by analyzing method traces. We have implemented a prototype tool based on the proposed technique and evaluated the technique on two software systems. As a result, we confirmed that the proposed technique could detect some refactoring candidates, which increase code quality.**

***Keywords*-refactoring; dynamic analysis; Move Method refactoring; software maintenance**

I. Introduction

In the case of a large project, it is difficult to keep high quality of software design because many programmers develop code to satisfy various requirements [1]. To solve the problem, we apply refactoring to source code. Refactoring is a technique for improving software quality of design and is defined as "does not alter the external behavior of the code, yet improves its internal structure" [2]. Until now, many refactoring support techniques have been proposed, and they are based on static analysis [3], [4].

Dynamic analysis is a technique that analyzes the data gathered during "program executions", instead of analyzing its source code. Through executing a program, we can obtain an actual program behavior, which cannot be obtained from its source code. For example, the number of invocations of every method, results of dynamic dispatch, actual method invocation sequences, and so on.

In this paper, we propose a technique to detect refactoring candidates by analyzing method traces. A method trace is a log containing sequential method invocations in program execution. Our proposed technique detects irregular methods in the same class as refactoring candidates. Herein, an irregular method means that it appears in different phases [5]. The quality of source code increases by moving these methods to appropriate classes because they cooperate with one another in a program execution. In other words, static structure of source code correspond to dynamic functionalities.

We have implemented a prototype tool based on the proposed technique and evaluated it on two software systems. As a result, we confirmed that the proposed technique can detect some refactoring candidates which can increase code quality by moving them. This paper makes the following contributions:

- **A new refactoring support technique.** Our technique analyzes method traces to detect refactoring candidates. This approach enables us to detect refactoring candidates based on the relationship between methods in program executions. It means that we can refactor elements detected in a program execution (*e.g.* types, methods actually called).
- **Visualization.** We visualize the phase data of methods to find refactoring candidates easily. The visualization also makes possible to understand characteristics of refactoring candidates.

II. Background

Tsantalis et al. proposed a methodology to identify *Move Method* refactoring opportunities by solving many common *Feature Envy* bad smells from source code [3]. They defined the *Jaccard distance* between an entity (attribute or method) and a class enabling the direct extraction of refactoring suggestions. Besides, Tip et al. proposed refactoring methodology based on type constraint [4]. This technique calculates relations between types of variables in source code. They present the *Extract Interface*, *Generalized Declared Type*, *Infer Generic Type Arguments* refactorings in the paper.

Like those works, previous approaches detect refactoring candidates by analyzing source code. However, those methods, which are based on static analysis, did not consider how program elements cooperate with one another in program execution, so that those techniques might miss some refactoring opportunities. For example, there is a problem that we cannot obtain results of dynamic dispatch by using static analysis. We show an example code of this problem in Figure 1, which is a piece of code that a dynamic dispatch appearing in a clone detector. This code shows a problem of static analysis that we cannot get information of "which instance of class Comparator⟨T⟩ is assigned to s".

```
class SearchFiles{
    static class FileSort implements Comparator<File>{
        public int compare(File src, File target){
            int diff = src.getName().compareTo(target.
                getName());
            return diff;
        }
    }
    ...
    private static void ListPath(File dir) {
        // dynamic dispatch
        if( target.isFile() )
            s = new FileSort()
        else
            s = new ArraySort()
        Arrays.sort(fs, s);
    }
}
```

Figure 1. An example of code that using dynamic dispatch.

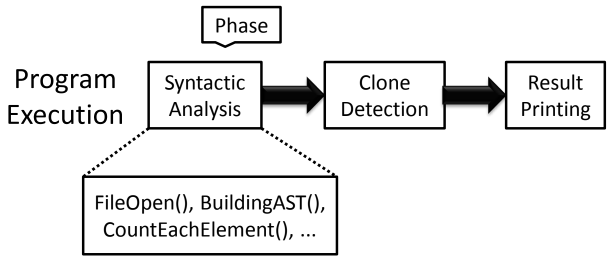

Figure 2. An image of phase.

In consequence, previous approaches might miss some of refactoring opportunities.

Our proposed technique uses dynamic analysis for identification of refactoring candidates. This technique may identify refactoring candidates, which are hard to be detected by the technique using static analysis, so our technique is complementary to previous approaches. The objective of this research is to reveal the following question:

Research Question: Can we identify refactoring opportunities by using dynamic analysis?

III. Phase

According to [5], a program execution is composed of multiple functions, and a gathering of methods that represents meaningful processing is named **phase**. Figure 2 is an image of phases. In this figure, the program execution divided three phases: Static Analysis, Clone Detection and Result Printing. Each phase consists of method invocations which is used for realizing a function. In the case of "Static Analysis" phase, there are file-open method, AST building method, element counting method, and so on. The reason described above, a method appearing in different phases means that it is used for different functionalities.

There is a tool to detect phases automatically, named Amida [6]. Amida detects phase-changing-point from a method trace and divides it into multiple phases. By dividing

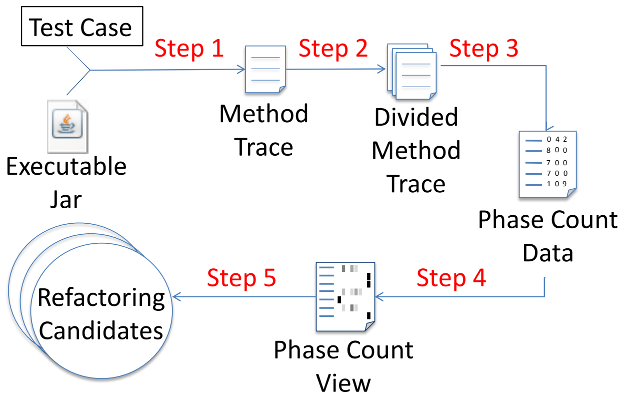

Figure 3. Overview of the proposed technique.

method traces, the phases reflect how a program is executed, such as repetition count and dynamic dispatch.

The large number of divided phases means that each phase represents detailed functions. According to [6], the precision of phase detection is over 90% when the number of output phases is below the actual number of phases in a method trace. In this research, our technique visualizes phase data, so too many number of phases prevents us detecting refactoring candidates. Thus, we adjusted the number of phases to nearly 10, for realizing the two requirements: precise phase division and number which is easy to see after visualizing.

IV. Proposed Technique

In this section, we describe steps for obtaining refactoring candidates. Figure 3 shows the overview of the proposed technique. The inputs of our technique is an executable Jar file and a test case. The proposed technique detects Move Code refactoring candidates with the inputs.

We have to decide test cases in advance because we need test cases to use dynamic analysis techniques. The more test cases we prepare, we can obtain results in more detailed. However, execution of many test cases needs much time. Thereby, we use only one test case in this paper.

The steps of the proposed technique are as follows.

(Step 1&2) Obtain a method trace and divided method trace: We obtain a method trace by running the given program with the given test case. After that, divide it into multiple method traces which represents a phase.

(Step 3) Count invocations of each method in the same phase: In a method trace, there are invocations of the same method repeatedly appears. In this step, we count the number of appearance of each method per phase by using the divided method trace which are obtained in the previous step. As a result, we obtain "phase count data", which is the sequence of counted numbers of method invocations per phase.

(Step 4) Visualize phase count data: In this step, we visualize a phase count data (obtained in Step 3). The technique how to visualize the phase count data are as follows.

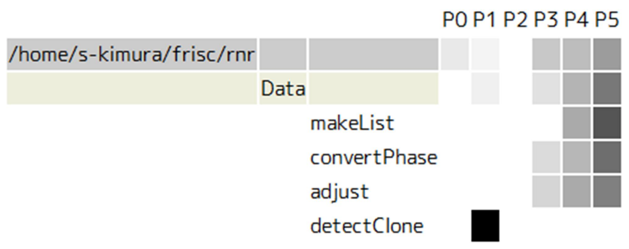

Figure 4. An example of a part of tool output.

We define a class as c, and methods belonging to class c as $m_0, m_1, ..., m_n$, the number of phases as N, the number of invocations of m_i in the phase a as p_{i_a}. Additionally, the sum of all counts of a method as P_i, given by

$$P_i = \sum_{k=0}^{N} p_{i_k} \tag{1}$$

We convert the number of methods invocations per phase into the depth of color. When the number of invocation of phase a in m_i is p_{i_a}, the ratio of the color depth (p'_{i_a}) is given by

$$p'_{i_a} = \frac{p_{i_a}}{P_i} \tag{2}$$

We render $p'_{i_0}, p'_{i_1}, ..., p'_{i_N}$ in the same line as cells which are filled with each color depth. We repeat this step with incrementing i from 0 to n.

As a result, we obtain "phase count view". An example of this output is Figure 4. As we can see, a method name and a sequence of colored cells are displayed in each line. Additionally, deep color represents many invocations of the method in the phase, so the method is specialized the function of the phase.

(Step 5) Detect refactoring candidates: We detect refactoring candidates by using the phase count view (obtained in Step 4). The detailed step how to detect refactoring candidates are as follows.

First, we classify methods in a class with their pattern of the depth of color. For example, there are four methods in Figure 4. "makeList", "convertPhase", "adjust" have a similar pattern, and the pattern of "detectClone" is different from those three methods. Thus, there are two classification: one is a pattern that P3-P5 are filled, and another is a pattern that only P1 is filled. Next, we detect minority classification as irregular methods, and they are refactoring candidates. In Figure 4, "detectClone" is an irregular method and a refactoring candidate of Move Method refactoring.

Methods which have different patterns are used in different functionalities, thus, they should be moved to different classes if they are minority in a class. We discover irregular methods in terms of appearance of the phase count view, and each of them is regarded as a refactoring candidate.

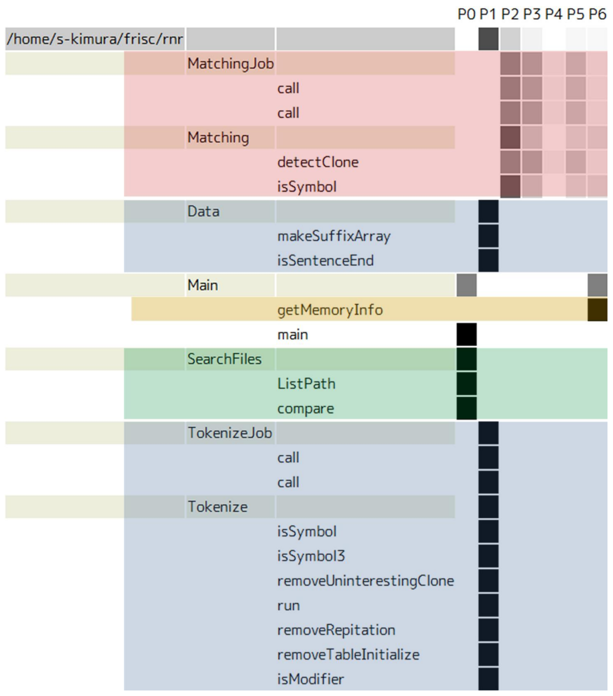

Figure 5. The output for FRISC.

After the technique: If the candidates which have a unique pattern of all patterns, they should be moved to a new package. If the pattern of the candidates exists in other class, the candidates should be moved to the class which has the same pattern (Move Method refactoring). The candidates are methods, but if all methods in a class are the same candidates, the class should be moved to other package (that is to say, Move Class refactoring). Additionally, when all classes in a package are the same candidates, it can say the same thing (that is to say, Move Package refactoring). Like this, our technique supports detecting candidates of refactorings which moves a piece of code (Move Code refactoring), and the unit of moving is method, class or package.

V. Evaluation and Discussion

A. Experimental Design

The proposed technique has been evaluated on two software systems, "FRISC" [7] and "MASU" [8]. FRISC is a code clone detector written in Java language. That is a relatively small software which has 8 files, 846 lines of code, and 21 methods. On the other hand, MASU is a metrics measurement tool written in Java language. That is a larger software than FRISC, which has 1,153 files, 133,045 lines of code, and 3,957 methods in the newest version. We prepared one test case for each target with their basic functionality. We suggest refactoring candidates detected by our technique to developers, and questioned whether the refactoring candidates are appropriate.

B. Result

Figure 5 is a visualization result of our technique for FRISC. We found 7 methods in FRISC and 58 methods in MASU as refactoring candidates.

1) FRISC: According to Figure 5, there are four groups: classes used only in P0 (*green*), classes used only in P1 (*blue*), classes used in P2-P6 (*red*), methods only using in P6 (*yellow*). Blue pattern has 11 members of 18 methods, so other groups (7 methods) are judged minority and refactoring candidates. For example, "SearchFiles.java" showed as green in Figure 5 is a refactoring candidate, and suggest moving them into a new package to separate functionalities. We suggested those refactoring candidates to the developers, as a result, we obtained developers' consent to our refactoring candidates. Note that we remove "main" method from refactoring candidates because it is called only once in program execution.

2) MASU: We found various refactoring candidates for MASU. In this paper, we discuss detected refactoring candidates only in a package "main.data.target.unresolved". In this package, we found 5 refactoring candidates. Three candidates are classes, "JavaUnresolvedExternalFieldInfo", "JavaUnresolvedExternalClassInfo", and "JavaUnresolvedExternalMethodInfo". The residual candidates are methods, "addModifier" and "getModifiers". We discuss them as a special feature of class names of this package.

This package has 99 classes, and 58 classes appeared in the method trace. 3 classes have a prefixed string "JavaUnresolvedExternal", 54 classes have a prefixed string "Unresolved", and the last is "NameResolver". We have detected all files prefixed "JavaUnresolvedExternal" as refactoring candidates, and we obtained developers' consent to these candidates. However, we also detected getter and setter methods as refactoring candidates. In general, setter methods are invoked in the initial stage of program execution, and getter methods are invoked in the latter half of program execution. Such methods separately appearing in method trace, but it is better design to gather them in the same class. Hence, such methods deem to be refactoring candidates by using our technique, but they are false positive.

C. Answer to the research question

We discuss the research question, which is mentioned in Section II. In the proposed technique, we use a method trace which represents program behavior. In the evaluation, we found 7 (7) methods in FRISC and 58 (39) methods in MASU as refactoring candidates. Note that the numbers of refactoring candidates consented by developers are parenthesized. Thus, our proposed technique is effective in the detection of refactoring candidates.

D. Threats to Validity

In the evaluation, we use only one test case. However, method traces depend on test cases; thus, we obtain different method traces by running different test cases. Additionally, the number of phases affects the detection of refactoring candidates. Thus, the precision of detecting refactoring candidates depends on how to detect phases.

VI. Conclusions and Future Work

This paper addresses the question whether we can detect refactoring candidates by using dynamic analysis. To answer this question, we proposed the technique and evaluated it on two real software. As a result, we confirmed that our proposed technique could detect refactoring candidates.

Our future work focuses on four directions. First, we will have quantitative comparison to the other result based on static analysis. Second, we will create a tool to combine results which can be obtained by using multiple test cases, for improving the precision of our technique. Third, we will apply dynamic analysis to other execution traces (*e.g.* access trace of variables). Finally, we will experiment with large open source software.

Acknowledgment

This study has been supported by Grants-in-Aid for Scientific Research (A) (21240002), Grant-in-Aid for Exploratory Research (23650014, 24650011), and Grand-in-Aid for Young Scientists (A) (24680002) from the Japan Society for the Promotion of Science.

References

[1] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Trans. Software Eng.*, vol. 27, no. 1, pp. 1–12, Jan/Feb 2001.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Pearson Education, 1999.

[3] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 347–367, May/June 2009.

[4] F. Tip, R. M. Fuhrer, A. Kiezun, M. D. Ernst, I. Balaban, and B. D. Sutter, "Refactoring using type constraints," *ACM TOPLAS*, vol. 33, no. 3, April 2011.

[5] S. P. Reiss, "Dynamic detection and visualization of software phases," in *WODA*, May 2005, pp. 1–6.

[6] T. Ishio, Y. Watanabe, and K. Inoue, "Amida: a sequence diagram extraction toolkit supporting automatic phase detection," in *ICSE*, May 2008, pp. 969–970.

[7] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Folding repeated-instructions for improving token-based code clone detection," in *SCAM*, Sep 2012, to be published.

[8] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue, "A pluggable tool for measuring software metrics from source code," in *IWSM/MENSURA*, Nov 2011, pp. 3–12.