RefactoringScript: A Script and Its Processor for Composite Refactoring

Linchao Yang
Waseda University
Tokyo, Japan
young@fuji.waseda.jp

Tomoyuki Kamiya
Waseda University
Tokyo, Japan
kamiya7140@akane.waseda.jp

Kazunori Sakamoto
National Institute of Informatics
Tokyo, Japan
exkazuu@nii.ac.jp

Hironori Washizaki
Waseda University
Tokyo, Japan
washizaki@waseda.jp

Yoshiaki Fukazawa
Waseda University
Tokyo, Japan
fukazawa@waseda.jp

***Abstract*—Refactoring is widely recognized as a method to improve the internal qualities of source code. However, manual refactoring is time-consuming and error prone. Consequently, many tools to support automated refactoring have been suggested, but most support only unit and simple refactoring, making it difficult to perform composite refactoring (e.g., introducing a design pattern) where a refactoring set is applied at one position or the same refactoring operation is applied at multiple positions. In this paper, we propose a novel script language and its processor to describe how and where to refactor by a model expressing source code[*1]. Evaluations indicate that our language and processor allow refactoring steps to be described as scripts, which can be easily replayed and reused for multiple projects.**

Keywords: Refactoring; Code Manipulate;

I. INTRODUCTION

Refactoring, which is defined as "a technique to improve the design of the internal structure of software without changing its external behaviors", has become commonplace in recent years. Although the most common refactoring techniques (e.g., extract and change field names) have been organized into patterns, manual refactoring is time-consuming and error prone. To resolve this problem, many automatic refactoring tools and methods have been proposed.

Here, we define the following two terms, basic refactoring and composite refactoring, to specify two types of refactoring. Basic refactoring refers to a simple refactoring that cannot be decomposed further. This paper uses refactoring by Eclipse as a standard example. On the other hand, composite refactoring refers to a composite composed of a combination of basic refactorings. Combinations include applying several refactorings in one place, the same refactoring to several places, or both.

Although many current tools support automatic execution of basic refactoring, there is no mechanism to define and apply composite refactoring. Vakilian et al.[8] proposed Compositional Paradigm to implement composite refactoring by setting the operations in detail. They also reported that setting the dialog may hamper coding work, generating more overhead costs and decreasing productivity. Mens et al.[7] reported that even if options can be finely set, extensions and settings to match the domain of the object are insufficient in current tools. Therefore, an interface that can simplify the setting work of detailed refactoring options and easily perform refactoring is required. As suggested by Vakilian et al.[9] in their survey on the trend of invoking refactoring operations in Eclipse by examining the recordings, composite refactoring is very common. Methods used to introduce a design pattern by refactoring are summarized in [3].

Composite refactoring has a high execution cost because a basic refactoring function via a keyboard or mouse must be used to locate the target every time. Additionally, as the numbers of locations and actions increase, performing each basic function correctly becomes more difficult, increasing the likelihood of an omission. Although many patterns have been created for refactoring, many tools cannot record and reuse the processes in the patterns. Therefore, it is difficult to apply frequently used composite refactorings to other projects. Eclipse can record and replay refactoring operations as a script, but it is used to help programmers upgrade an older version of a library once a newer version is distributed. It is impossible to create a script arbitrarily and describe the steps of refactoring freely.

In this paper, we propose a script and its corresponding processor(called RefactoringScript) that can be used to describe the processes of refactoring. RefactoringScript contains a script language and a processor to perform refactoring. We address the following research questions:

- RQ1 Is it possible to script and apply refactoring operations (applying places and actions) concisely and accurately?
- RQ2 Compared to the case without this tool, is composite refactoring executed correctly?
- RQ3 Compared to the case without this tool, is the cost of composite refactoring reduced?
- RQ4 Is it possible to reuse the refactoring operations in other projects?

The contributions of this paper are:

- A RefactoringScript language to describe refactoring operations
- A RefactoringScript processor to apply scripted refactoring operations

*1 The preliminary idea of RefactoringScript has been originally published at [11] in Japanese. In this paper we added the capacity of manipulating statements to make RefactoringScript support more complicated refactoring. Moreover we change the implement of interpreter from JRuby to Scala.

- Implementation of a RefactoringScript language and its processor as an Eclipse plug-in
- Evaluation of a RefactoringScript language and its processor to show its usefulness

In addition, we use Eclipse JDT to invoke refactoring operations and Scala to develop the plug-in and the DSL. Because Scala and Java can use each other's libraries directly, RefactoringScript is implemented with a low cost.

Our paper is organized as follows. Section II provides motivating examples. Then Section III describes the proposed RefactoringScript language and its processor, while Section IV presents the results and discussion of our experimental evaluation. Finally, Section V concludes the paper.

II. BACKGROUND

We consider three cases as motivating examples.

A. Renaming Relevant Elements

According to [9], a common combination is refactorings regarding renaming a field and it related elements. For example, after changing the field name, the names of the accessor of the field (change the method names) must also be changed (List 1).

However, current refactoring tools only change the definition and references when performing rename refactoring to a field. For example, if rename refactoring is invoked to change the name of field "page" in List 1 to "pageCount", the name of the corresponding accessor is not changed. Hence, programmers must also invoke the rename refactoring for the accessor. Figure 1 is an example of RefactoringScript to do these two refactorings.

B. Applying Coding Conventions

Many projects have their own coding conventions. To enhance maintainability of the entire code, especially for team development, all team members must observe the coding convention established before development. [4] and [10] summarize the underlying rules, which can be used and modified freely. For example, the rule (27) places the underscore prefix or suffix of the name for a private, protected field, while the rule (44) avoids overloading the method.

For a project with an already inflated scale, the following is necessary to apply these conventions:

a) Among the protected or private fields, extract all names without an underscore prefix (or suffix), and execute the rename method.(Example script is shown in Figure 2.)

b) Acquire all methods that have the same name and the same number of arguments from a specific class, and execute the rename method.

Although coding conventions can be used in multiple projects, if a new coding convention is applied or an old coding convention is changed for an existing source code, the execution cost for refactoring all relevant places is very high. Additionally, the more places where refactoring occurs, the likelihood of a mistake increases.

List 1. Example of changing a field name and the corresponding accessor name (Left: original code; Middle: renamed field; Right: renamed the accessor).

private int page;
public int getPage()
{
  return page;
}
	private int pageCount;
	public int getPage()
	{
	  return pageCount;
	}
		private int pageCount;
		public int getPageCount()
		{
		  return pageCount;
		}

```
1  val f = cls.fields.select(By.name("page"))
2  val m = cls.method("getPage")
3  f.rename("pageCount")
4  m.rename("getPageCount")
```

Figure 1. Example script of renaming field and the corresponding accessor

```
1  val fs = cls.fields.select(By.modifier(With.or("protected", "private")))
2
3  fs.foreach(f => {
4      if(!f.name.startsWith("_")) {
5          f.rename("_" + f.name)
6      }
7  })
```

Figure 2. Example script of applying coding conventions

C. Introducing Design Patterns

The transformation to introduce a design pattern involves many iterations of refactoring. For example, introducing a Visitor Pattern includes the following two types of refactoring:

a) Move Method

b) Rename Method (to avoid name collisions, add "visit" to the beginning of the name of the method that has been moved.)

Even if only the above two refactorings are considered, operations like "Find the methods by the specified signature from subclasses of a specified class, and move to another class" and "Rename method that has been moved with a new name based on the name of the target class" are necessary. These operations will be performed repeatedly, which is clearly a high cost. However, these refactorings can be formally scripted.

III. REFACTORINGSCRIPT LANGUAGE AND ITS PROCESSOR

In this section, we describe the design of the proposed RefactoringScript language and its processor.

A. Requirements

Requirements of the RefactoringScript language and its processor are:

R1 Analysis API and Refactoring Function: The refactoring location can be identified, and the refactoring operation can be executed.

R2 Concise Script Expression: The script need only include locations and operations necessary for refactoring.

R3 Immediate Execution: The script can use the plug-in resource easily.

R4 Widely Available: The introduction cost is small and readily available.

B. Overview

In this section, we describe the interactions between the RefactoringScript language, its processor, and users. The RefactoringScript consists of two components.

- RSCore[*2]: The fundamental part, which includes elements for the RefactoringScript language and its processor.
- RSUI: The user interface part, such as an editor[*3] to create or modify script, and a menu[*4] to execute script by inputting script into the interpreter of RefactoringScript in RSCore.

The procedure for a user to apply script to a workspace, and the interaction between user and RefactoringScript processor are as follows:

(1) User creates and edits the script in the editor.

(2) User activates the core component by specifying the script file.

(3) Processor inputs script into the interpreter.

(4) Interpreter runs the script and applies it to the user's workspace

(5) User is notified of the script execution result.

C. Language

In this section, we describe the elements of RefactoringScript language.

1) Code Entity and Code Entity Collection

Java elements of JDT provide APIs, which are suitable for searching a particular element from the workspace. However, the Java elements do not have APIs that allow the conditions to be specified in detail to determine the specified elements. Two types of APIs is added to Code Entity(CE) which is a class based on Java element:

- APIs to analyze and search. For example, the select method which we will describe later.
- APIs to trace the tree structure of the code in the description similar to the simple natural language. For example, we prefer to use c.methods rather than c.getMethods() to acquire all methods of c.

Table I shows the correspondence between the Java Element and CE. An indentation in the table represents the containment relationship of the package or the class inheritance. It should be noted that RSWorkspace differs slightly from the other CEs; RSWorkspace represents a reference to the target workspace, and is a starting point to find the other CEs.

The Code Entity Collection (CEC) represents a set of CEs and provides APIs that can search for CEs included in the set. An example script to perform a search is introduced in the next section.

2) Query Selector and Qualifier

*2 https://github.com/hugh3166/RSCore
*3 https://github.com/hugh3166/RSEditor
*4 https://github.com/hugh3166/RSLauncher

```
1  //Qualifier can be omitted.
2  ms.select(By.modifier("private"))
3
4  //Interpret parameters as OR.
5  fs.select(By.modifier(With.or("private", "protected")))
6
7  //Select method can be linked.
8  ms.select(By.modifier("public")).select(By.typename(With.out("int", "String"))
```

Figure 3. Example script of the select method

By using the select method to search for a CE from CEC, we write the script by combining *SearchParams*, *QuerySelector,* and *Qualifier* in the following format:

CEC.select(*QuerySelector* (*Qualifier* (*SearchParams*)))

QuerySelector ::=
"By.name"|"By.namereg"|"By.modifier"|"By.typename"

Qualifier ::= ""|"With.or"|"With.and"|"With.out"

QuerySelector is a keyword that specifies the Search Key, which refers to the four regular representations: names, name of the CE, access modifier, and type name. Table II shows each CE and the corresponding combination of the search key and query selector, where O indicates that it can search CE using the search key, and X indicates cannot. For example, a set of RSProject can be searched by key elements in the name, but not by the key elements in access modifier name. *Qualifier* is a keyword that specifies whether to interpret the given search parameters as OR, AND, or NOT. However, it can be omitted (if there is only one search parameter) when a qualifier is not required. Figure 3 shows three examples of using select method. The select method has also been used in Figure 1 and Figure 2 at the first line.

3) Action

A refactoring operation for CE / CEC is called an action. With parameters(params), an action is expressed in the following format:

CE/CEC.Action(params)

An action parameter may be specified as the minimum required when performing refactoring. Table III summarizes the types of the actions, which have been supported and the parameters can be currently specified. For example, CE.rename("newname") will apply rename refactoring to CE. As we showed at line 3 and 4 in Figure 1, or line 5 in Figure 2.

D. Processor

In this tool, we adopt Scala to implement the processor. Scala is based on JVM, so it can use the assets of Java seamlessly. RefactoringScript language can be regarded as an internal DSL of Scala. Developers only need to focus on the descriptions of the searching CE and handling CE, because Scala expressions and the built-in functions or libraries of both of Java and Scala are available in the script. Additionally, because Scala is adopted, a new interpreter does not have to be implemented, allowing the interpreter to be incorporated into the processor economically.

IV. Evaluation

A. Evaluation Design and Results

To evaluate the describability, the accuracy, execution cost, and reusability of RefactoringScript, we conducted subject

experiments and case studies for the four composite refactorings, which were selected by considering trends of refactoring[9] and coding conventions[4].

EX1. Assign a prefix to the name of every private field for all classes in a specific package.

EX2. Generate template method from subclasses into a superclass.

EX3. Encapsulate classes with Factory.

EX4. Change the name of a specified field in a package and the name of the corresponding accessor.

1) Describability

For EX1, 2, 3, 4, we measured the lines of code for processing refactorings in the Java language and the RefactoringScript language (Table IV). Note that the Java projects used as experimental objects are also the test data used for testing RSCore.

For EX2, we used the simplified experimental code in List 2. Two subclasses have the same name method "startGame", but they have different conditional structures. For simplicity, we just extract the conditional structures to form two new methods with same name, and pull up the two "startGame" methods and one of the extracted methods into a superclass. Because the two "startGame" methods are the same method in the superclass, they can be regarded as a template method. Afterwards, the subclasses can change the behavior of the "startGame" by overriding the method "start" or "extract" without overriding the "startGame" method directly, such as the class Game2 shown at the right of List 2.

2) Accuracy and Execution Cost

We conducted the following subject experiments to compare the accuracy and execution cost between manual composite refactoring and RefactoringScript. The experimental objects are the sample projects prepared for the experiments. It should be noted that in consideration of the similarity of operation difficulty and the influence of prior knowledge, we only used EX1 and 4 as the experimental objects. In addition, for simplicity, we chose int type fields to be the target fields, and wrote the script to extract the fields by type in EX4.

Experimental Subjects: Five Information Engineering undergraduate and graduate students (P1~P5)

Approach: Divide subjects into two groups. Make one group conduct EX1 manually and then EX4 by RefactoringScript, and make the other group conduct EX1 by RefactoringScript and then EX4 manually. Then measure the time necessary to complete refactorings and the places where refactoring is applied correctly.

Tables V and VI summarize the results of this subject experiment. Table V shows that subject P1 took seven minutes to do EX1 manually, 22 minutes to do EX4 by script, while Table VI shows that subject P1 applied refactoring correctly at 27 of 30 places in EX1 manually, but applied 96 of 96 places in EX4 by script.

TABLE I. CORRESPONDENCE BETWEEN THE ELEMENTS IN JDT AND CE

org.eclipse.jdt.core	
IMember	RSMember
IType	RSClass
IField	RSField
IMethod	RSMethod
IPackageFragment	RSPackage
ILocalVariable	RSParameter
IJavaProject	RSProject
org.eclipse.core.dom	
Statement	RSStatement
org.eclipse.core.resources	
ResourcesPlugin	RSWorkspace

TABLE II. QUERY SELECTOR

Search Key	**Name**	**Regular Expression of Name**	**Access Modifiers**	**Type**
QuerySelector	***By.name***	***By.namereg***	***By.modifier***	***By.typename***
RSStatement	X	X	X	O
RSField	O	O	O	O
RSMethod	O	O	O	O
RSClass	O	O	O	X
RSParameter	O	O	X	O
RSProject	O	O	X	X
RSWorkspace	X	X	X	X

TABLE III. TYPES OF SUPPORTED ACTIONS AND CORRESPONDING PARAMETERS

Action	**Receiver**	**Parameter**
rename	RSField	New name
rename	RSMethod	New name
encapsulate	RSField	-
introduce_factory	RSClass	Destination Class
introduce_factory	RSMethod	Destination Class
introduce_parameter_object	RSMethod	Class Name of Object
pull_up	RSMethod	Destination Class
push_down	RSMethod	-
change_return_type	RSMethod	New Type Name
extract_method	RSStatement	New Method Name
delete	RSEntity	-
move	RSEntity	Destination Class

TABLE IV. COMPARISON OF NUMBER OF LINES OF SCRIPT IN REFACTORINGSCRIPT WITH JAVA

	Java	**RefactoringScript**
EX1	42	10
EX2	143	14
EX3	107	9
EX4	48	12

Unit: Lines

TABLE V. COMPARISON OF EXECUTION TIME FOR REFACTORING BY SCRIPT AND MANUALLY

Experiment	P1	P2	P3	P4	P5	Average
EX1(Manually)	7	-	5	-	-	6.0
EX1(Script)	-	10	-	5	14	9.7
EX4(Manually)	-	17	-	9	13	13.0
EX4(Script)	22	-	10	-	-	16.0

Unit: Minutes

TABLE VI. COMPARISON OF ACCURACY OF REFACTORING BY SCRIPT AND MANUALLY

Experiment	P1	P2	P3	P4	P5	Average
EX1(Manually)	27	-	30	-	-	28.5
EX1(Script)	-	30	-	30	30	30
EX4(Manually)	-	95	-	96	93	94.7
EX4(Script)	96	-	96	-	-	96

Unit: Places

TABLE VII. APPLIED REFACTORINGSCRIPT TO OPEN SOURCE PROJECTS

Project	Experiment	Number of Files	Number of Lines	Applying Places
P1	EX1	3	68	16 fields
P1	EX4	2	18	6 fields 12 methods
P2	EX3	6	20	6 classes 6 methods

```
1  import org.eclipse.jdt.core.dom.ASTNode
2
3  val proj = RSWorkspace.project("Ex0")
4
5  val cls1 = proj.pkg("p").classes.select(By.Name("Game1")).first
6  val mhd1 = cls1.method("startGame").first
7  val ifStmt1 = mhd1.body.statements.findByKind(ASTNode.IF_STATEMENT)
8
9  val cls2 = proj.pkg("p").classes.select(By.Name("Game2")).first
10 val mhd2 = cls2.method("startGame").first
11 val ifStmt2 = mhd2.body.statements.findByKind(ASTNode.IF_STATEMENT)
12
13 ifStmt1.extractMethod("extracted")
14 ifStmt2.extractMethod("extracted")
15
16 val mExtr1 = cls1.method("extracted").first
17 val ms = new RSCollection(Array(mExtr1, mhd1, mhd2))
18 ms.pullUp(cls2.superclass)
```

Figure 4. Example script of EX2

1) Reusability (Case Study)

We applied the EX1, 3, 4 to open source project P1[*5], P2[*6] to determine the mechanical differences with the source code applied to manual refactoring. By verifying the results with the goals, it is confirmed that the proposed method is able to process the object refactoring. Because the focus is on only the refactoring operation, we selected projects with a moderate scale as the experimental material. Table VII lists the projects, kinds of experiments, numbers of files, numbers of lines affected by refactoring, and the applied places.

B. Discussion

1) Describability

RQ1 Is it possible to script and apply refactoring operations (applying places and actions) concisely and accurately?

*5 https://github.com/shigenobu/acbook-wa710

*6 http://code.google.com/p/jslideshare/

List 2. Example to generate template method (Left: before refactoring; Right: after refactoring)

```
package p;

public class Game{
   protected int playerCount = 0;
   public void start(){
      System.out.println("Game Start");
   }
}

public class Game1 extends Game{
   public void startGame(){
      start();
      if (playerCount != 0){
         System.out.println("Restart");
      }
   }
}

public class Game2 extends Game{
   public void startGame(){
      start();
      if (playerCount >= 0){
         playerCount++;
         System.out.println("Join");
      }
   }
}
```

```
package p;

public class Game{
   protected int playerCount = 0;
   public void start(){
      System.out.println("Game Start");
   }

   public void startGame(){
      start();
      extracted();
}

   protected void extracted(){
      if (playerCount != 0){
         System.out.println("Restart");
      }
   }
}

public class Game1 extends Game{

}

public class Game2 extends Game{
   protected void extracted(){
      if (playerCount >= 0){
         playerCount++;
         System.out.println("Join");
      }
   }
}
```

In all four cases, the number of lines of script written in RefactoringScript is 1/4 to 1/10 of the script written in Java. The reduction is attributed primarily to two reasons:

- API allows CE to be flexibly searched, and conditional statements are less likely to nest.
- Processes not directly related to refactoring (e.g., acquiring workspace) do not have to be described.

In EX2, the project, package, class, and method entities are to search by name, but a statement is searched by the type defined at ASTNode class in JDT. Figure 4 shows the example script of EX2. Additionally, even on a scale like EX3, the number of lines of script written in RefactoringScript can be as few as 10. Because the RefactoringScript language is specialized to describe the processes and search loctions for refactoring, concise scripting can be realized.

2) Accuracy and Execution Cost

RQ2 Compared to the case without this tool, is composite refactoring executed correctly?

RQ3 Compared to the case without this tool is the cost of composite refactoring reduced?

With regard to the execution cost, manual refactoring required slightly less time in each experiment. Based on the feedback from the subjects, this is likely because learning the

RefactoringScript takes some time. In fact, some of the subjects commented:

- I was confused by the script language idiom.
- I think that RefactoringScript can reduce the time once I learned how to write with it. (In this experiment, the answers to the examples and script pieces required by the experiment were distributed as material.)

On the other hand, feedback regarding manual refactoring indicated a desire for an automatic method:

- I do not want to refactor more complex objects manually. (For example, when the applied places are enormous).
- Firstly, I prefer not to do manually simple mechanical work.

Therefore, we believe that once developer become familiar with RefactoringScript, the burden of refactoring can be reduced.

With regard to accuracy, all the scripts written by the subjects worked properly using the script case, whereas the manual refactoring contained the following mistakes:

- Renamed fields that are not specified (EX1).
- Field renamed correctly, but the name of the accessor was incorrect (EX4).

These errors indicate that the script contributes to correct composite refactoring.

3) Reusability

RQ4 Is it possible to reuse the refactoring operations in other projects?

The scripts used in EX1 and EX4 can be applied to other projects without substantial modification. Most projects require the following changes:

- Package name of the object.
- Action parameters (For example, in EX4, P1 changes field names 'created', 'updated', 'executed' to 'createdAt', 'updatedAt', 'executedAt' as well as the corresponding accessor names).

However, these elements are project specific and are the minimum parameters that user have to specify for per project.

C. Limitations

1) Lazy evaluation

In this tool, it is impossible to reflect the effect of refactoring on the CE. When multiple actions are performed on the same CE, the specified CE must be searched in each case. This problem should be resolved by introducing the lazy evaluation, which is a mechanism to set aside the query search for CE until execution. In addition, there is almost no issue to apply many basic refactorings to a particular CE.

2) Error Handling

This tool does not provide error handling when the script is running. The information of the notifying dialog lets users know the script is successfully executed, but it cannot understand the cause of a failure. Additionally, if the prerequisites of the refactoring are not satisfied, refactoring will not be performed if an error is detected internally. However, this tool will not notify the user why refactoring failed. Strengthening the user notifications should resolve these issues.

3) Threats To Validity

In the evaluation, we selected four refactorings that can be implemented relatively easily with RefactoringScript, but did not select experimental material that is impossible to implement. Hence, it is possible that the evaluation experiments used in other studies cannot be implemented in RefactoringScript. Additionally in the subject experiments, the ratio of learning cost of the script within working time increased. In future, we aim to measure the pure working time, by subtracting accurately estimated learning costs.

V. CONCLUSION AND FUTURE WORK

We have proposed a RefactoringScript language and its processor to script refactoring processes and apply to appropriate places. Using CE searching API and Scala, we realized a user-friendly script. This tool should significantly reduce the cost of applying refactoring to many places or repeatedly applying refactorings across projects. Additionally by sharing accumulated scripts, it is possible to summarize combinations of refactorings and the specific remedies. Thus, refactoring should be become more common. Although the refactoring types supported by RefactoringScript are limited to the functions provided in Eclipse, we intend to expand it and realize more flexible code deformation.

REFERENCES

[1] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/.

[2] Martin Odersky. *Scala.* http://www.scala-lang.org/.

[3] Joshua Kerievsky. *"Refactoring to Patterns."* Prentice Hall. 2004.

[4] Oracle, *Code Conventions for the Java Programming Language.* http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.

[5] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. *"Jungl: a scripting language for refactoring."* In Proceedings of the 28th international conference on Software engineering, ICSE '06, pp. 172–181, New York, NY, USA, 2006. ACM.

[6] Mark Hills, Paul Klint, and Jurgen J. Vinju. *"Scripting a refactoring with Rascal and Eclipse."* In Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12, pp. 40–49, New York, NY, USA, 2012. ACM.

[7] T. Mens and T. Tourwe. *"A survey of software refactoring." Software Engineering*. IEEE Transactions on, Vol. 30, No. 2, pp. 126 – 139, feb 2004.

[8] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. *"A compositional paradigm of automating refactorings."* Technical report, University of Illinois, 2012.

[9] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. *"Use, disuse, and misuse of automated refactorings."* In Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 233–243, Piscataway, NJ, USA, 2012. IEEE Press.

[10] Kenji Hiranabe, Object Club. *Java Coding Standard.* http://www.objectclub.jp/community/codingstandard/CodingStd.pdf.(in Japanese)

[11] Tomoyuki Kamiya, Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa, "Refactoring Script: A script for composite refactoring and its processor," IPSJ Transactions on Programming, Vol. 6, No.3, 2013. (in Japanese)