Reordering Program Statements for Improving Readability

Yui Sasaki, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {*s-yui,higo,kusumoto*}*@ist.osaka-u.ac.jp*

***Abstract*—In order to understand source code, humans sometimes execute the program in their mind. When they illustrate the program execution in their mind, it is necessary to memorize what values all the variables are along with the execution. If there are many variables in the program, it is hard to their memorization. However, it is possible to ease to memorize them by shortening the distance between the definition of a variable and its reference if they are separated in the source code. This paper proposes a technique reordering statements in a module by considering how far the definition of a variable is from its references. We applied the proposed technique to a Java OSS and collected human evaluations for the reordered methods. As a result, we could confirm that the reordered methods had better readability than their originals. Moreover, we obtained some knowledge of human consideration about the order of statements.**

***Keywords*-source code readability, source code analysis, software tool, arranging program statements**

I. Introduction

In order to maintain a software system, we must understand its source code. However, software understanding itself is costly [1], [2]. For example, it is hard to understand the role and the value of a variable whose definition is separated from its reference [3]. In order to relieve the negative impacts caused by separated variable definitions and their references, it would be useful to move the statements of defining and referencing a variable close to each other in source code.

This paper proposes a new technique reordering statements in a module by considering how far the definition of a variable is from its references. Also, we applied the proposed technique to a Java OSS and investigated the usefulness of the proposed technique. As a result, we could confirm that the proposed technique identified reordering opportunities for about 200 methods and the some of the reordered modules had better readability than their originals.

The main contributions of this paper are:

- it proposes a technique to shorten the distance between definition and reference of a variable;
- it confirms that reordering statements can improve readability of Java methods by conducting an experiment with 44 subjects;
- it provides some knowledge of human consideration about the order of statements through the experiment.

We consider that reordering statements for improving source code is a kind of refactoring. There are many tools for automated refactoring, however, recent studies revealed these tools had rarely been used: one of the factors is that it is difficult for developers to predict the outcome of refactoring tools [4]. Therefore, we consider that reordering operation should be performed not automatically for all the modules but interactively for only a module specified by the users. The proposed technique is a first step for providing an interactive refactoring environment.

II. Related Work

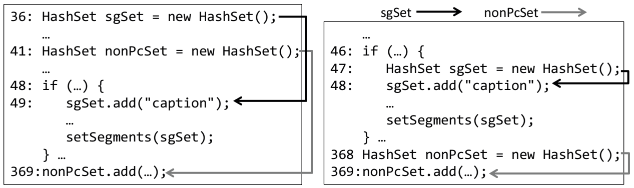
Buse and Weimer investigated code readability with several software metrics [5]. Their investigation result reported that the number of identifiers in each line affected readability. Also, Dunsmore and Roper reported that mental simulation, in which humans read source code and execute the program in their mind, is useful for program comprehension [6]. Nakamura et al. developed a model by representing human short-term memory as FIFO queue, and measured the cost of mental simulation [3]. They reported that appearances of variables not in the queue have negative impacts on understanding the source code.

Recent studies presented techniques for automatically formatting source code in order to improve readability. Wang et al. presented an automatic formatting tool that identified meaningful blocks and inserted blank lines [7]. Also, prettyprinting is one of the most famous techniques to reformat coding style such as indentation and blank lines. These studies improved readability of source code without changing its program behavior. However, it is difficult to reorder program statements automatically because there are various relationships between statements such as control dependence, data dependence, and control flow. Those relationships affect the program behavior. It has not been conducted such a study, and also not been evaluated the readability due to the difference of the order of statements.

III. Motivating Example

We consider variables *sgSet* and *nonPcSet* in the source code of Figure 1. Figure 1(a) shows an example that variable definitions and their references are separated. *sgSet* is referenced only within the if-block beginning at the 48th line, however it is defined outside of the block. It is naturally desirable that scope of variables is small as much as possible, which is a principle of the variable locality. Also, though

1534-5351/13 $26.00 © 2013 IEEE
DOI 10.1109/CSMR.2013.50



sgSet
nonPcSet

```
36: HashSet sgSet = new HashSet();
    …
41: HashSet nonPcSet = new HashSet();
    …
48: if (…) {
49:     sgSet.add("caption");
        …
        setSegments(sgSet);
    } …
369:nonPcSet.add(…);
```

(a) before

```
    …
46: if (…) {
47:     HashSet sgSet = new HashSet();
48:     sgSet.add("caption");
        …
        setSegments(sgSet);
    } …
368 HashSet nonPcSet = new HashSet();
369:nonPcSet.add(…);
```

(b) after

Figure 1. Motivating Example

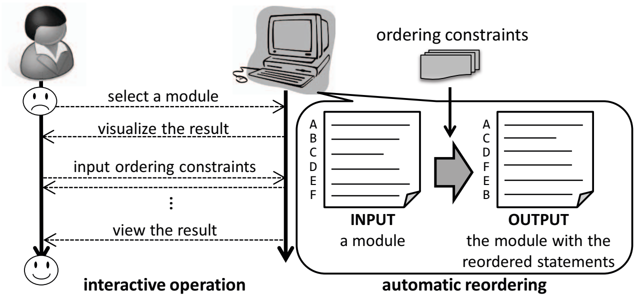


Figure 2. Overview of an Interactive Reordering Environment

scope of *nonPcSet* is not able to narrow down anymore because its definition and reference are at the same block, it is possible to move its definition close to its reference like Figure 1(b). In this paper, we propose a technique reordering program statements for improving readability of the source code. Specifically, we present following two reordering strategies.

- moving statements **into inner blocks** in order to narrow down variable scope (**strategy 1**)
- moving statements close to each other **within the same block** in order to shorten the distance between a variable definition and its reference (**strategy 2**)

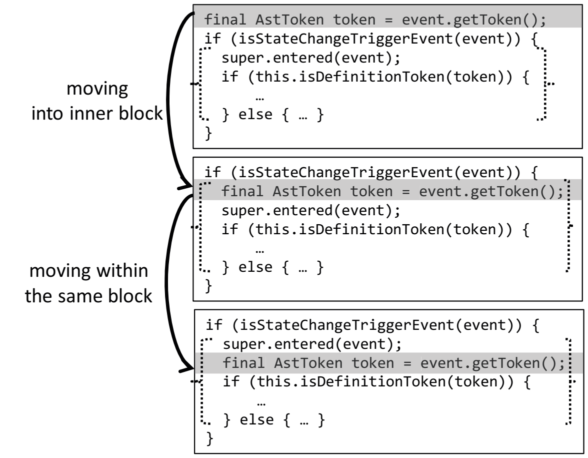
IV. Proposed Method

The purpose of our study is supporting refactoring interactively. Figure 2 shows an overview of an interactive refactoring environment that we are going to develop. At first, a user selects a module where she wants to improve readability. Then, the tool automatically reorders the statements with primary ordering constraints (as described later) and visualizes only one result to her. If she satisfies the result, she adopts it. However, if, not, she inputs her special intention for ordering statements as new ordering constraints. Such an interactive analysis for the module lasts until she satisfies the reordering result.

In the remainder of this section, we describe the automatic reordering technique for program statements.

A. Preliminaries

We identify Def-Use chain (*DUchain*) in a module by using data flow analysis technique. A DUchain is identified from every pair of definition and reference. If a variable is referenced twice, two chains are identified. Then, we use a distance between the statement defining a variable (s_1) and the statement referencing the variable (s_2) of a DUchain as a numeral metric[1]. Herein, distance means the number of statements between s_1 and s_2. Let *distance(c)* be the distance of DUchain c, and *DUchain(B)* be the set of DUchains existing in block B. Then, the total distance of the DUchains in B is defined as the following formula.



moving
into inner block
moving within
the same block

```
final AstToken token = event.getToken();
if (isStateChangeTriggerEvent(event)) {
  super.entered(event);
  if (this.isDefinitionToken(token)) {
      …
  } else { … }
}
```

```
if (isStateChangeTriggerEvent(event)) {
  final AstToken token = event.getToken();
  super.entered(event);
  if (this.isDefinitionToken(token)) {
      …
  } else { … }
}
```

```
if (isStateChangeTriggerEvent(event)) {
  super.entered(event);
  final AstToken token = event.getToken();
  if (this.isDefinitionToken(token)) {
      …
  } else { … }
}
```

Figure 3. An Example of Applying the Strategies

$$DataDistance(B) = \sum_{c \in DUchain(B)} distance(c)$$

B. Overview of the proposed technique

The proposed technique uses AST generated from input source code of a module. Specifically, it applies the two reordering strategies to every program block in the input module with a post-order traversal for the AST. Figure 3 represents an example of applying the two reordering strategies to the if-block beginning at the 2nd line.

- At first, if there are statements whose variable scope can be narrowed down, they are moved into the inner the if-block.
- Then, the proposed technique moves statements close to each other within the if-block in order to shorten *DataDistance* value of the if-block.

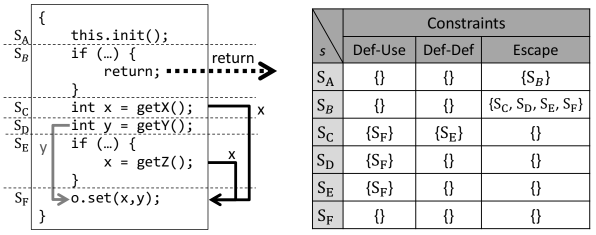
C. Implement of strategies

1) Strategy 1: Statement s is moved into inner block B if all the following conditions are satisfied:

- s locates outside of B;
- s is a variable declaration statement;
- s is included in all execution paths to B;
- all the variables defined in s are referenced only in B;
- all the variables defined in s are not re-defined in all execution paths to B.

The goal of the strategy is only to narrow scope of variables down. Consequently, we move such a statement

[1] If a variable points to an object whose state is changed by a method call, we regard that the variable is defined and referenced.



```
{
    this.init();
    if (…) {
        return;
    }
    int x = getX();
    int y = getY();
    if (…) {
        x = getZ();
    }
    o.set(x,y);
}
```

(a) source code

s	Constraints		
	Def-Use	Def-Def	Escape
S_A	{}	{}	$\{S_B\}$
S_B	{}	{}	$\{S_C, S_D, S_E, S_F\}$
S_C	$\{S_F\}$	$\{S_E\}$	{}
S_D	$\{S_F\}$	{}	{}
S_E	$\{S_F\}$	{}	{}
S_F	{}	{}	{}

(b) constraints of each statement

Figure 4. Example of Order Constraints

to the beginning of the inner block without considering its appropriate place in the inner block.

2) Strategy 2: The proposed technique creates all the sequences of statements that satisfy the constraints that does not change the behavior of the program (*primary constraints*) in the first step of the strategy 2.

The primary constraints are the followings. Herein, we describe the constraints with Figure 4 as an example. S_A .. S_F in Figure 4(a) are statements in the focused block. Figure 4(b) represents that each statement has a set of statements which must appear after the statement based on each constraint.

Def-Use Constraints: two statements in a DUchain have to keep their present order: i.e., S_F references y defined at S_D. If their order is reversed, S_F cannot reference y.

Def-Def Constraints: if a statement references a variable defined at multiple statements, the present order of those statements must be kept: i.e., S_F references x defined at S_C and S_E; if the order of S_C and S_E is reversed, S_F cannot reference x defined at S_E because S_C re-defines x.

Escape Constraints: if a statement includes jump-instruction such as return, break, or continue-statement, the present order between each of the other statements and it must be kept: i.e., S_B (which is if-block) includes return-statement, and so S_C, S_D, S_E and S_F are not always executed depending on a conditional expression of B, but S_A is always executed; if the orders between these statements and S_B are reversed, the execution conditions are changed.

Then, we describe how the strategy 2 is performed with the above three constraints as follows:

1) creating all the sequences satisfying all the constraints;
2) extracting the sequences whose *DataDistance* are minimum in all the sequences.

If there are multiple minimum sequences, we apply:

3) choosing only one sequence whose order is the most similar to its original.

In step 3, we use Spearman's rank correlation coefficient. That is, the correlation between each of all the sequences obtained in step 2 and the original one is measured, and then the sequence whose correlation coefficient is the highest, which means the sequence is the most similar to the original one, is selected as a result of the strategy 2.

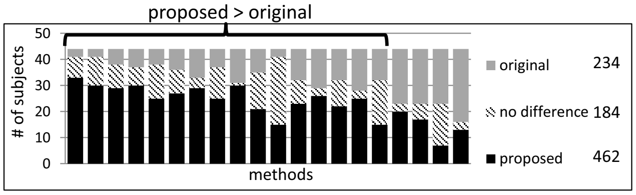


Figure 5. Result of All the Target Methods

V. Controlled Experiment

We implemented a software tool based on the proposed technique. Currently, the tool can handle Java language. We applied the tool a Java system, TV Browser, in this experiment. The system includes approximately 3,700 methods that include two or more statements.

By applying the tool to all the methods, we obtained reordering candidates from 215 methods. In order to evaluate the usefulness of the proposed technique, we selected 20 methods as the evaluation target methods from all the reordered methods because it was unrealistic that humans judged all the 215 methods manually.

Then, we made a questionnaire on the web. In the questionnaire, every subject selected one of the following options for every target method.

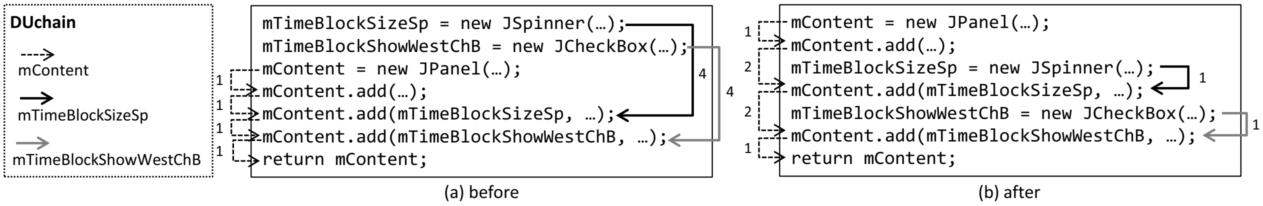
- A is easier to read than B;
- B is easier to read than A;
- There is no difference between A and B in readability.

We had manually confirmed that every target method had kept their behavior before the questionnaire because currently the proposed technique does not completely guarantee behavior preservation. In the questionnaire, we did not inform the subjects about what strategies of reordering were and which the original was. Additionally we removed all blank lines and comments for all the target methods and standardized their formats such as indents and line breaks.

We gathered subjects by social networks, and 44 subjects were joined. Eight subjects had programming experiences with less than 1,000 lines of Java code, 23 subjects from 1,000 to 10,000 lines, and 13 subjects more than 10,000 lines. Also, at least 28 subjects had used Java in their academic research and at least 12 subjects in their work.

A. Result

Figure 5 shows the breakdown of the answers from the 44 subjects for each target method. All the answers of all the subjects are aggregated based on the values. We found that 16 out of the 20 methods had more subjects who answered reordered method was easier to read than their originals. Herein, we performed statistical testing for the result by Wilcoxon signed-rank test, and then we confirmed that there is a significant difference between the number of subjects who selected reordered method and who selected original one. This result shows that the proposed technique enables to improve the readability of source code.



```
mTimeBlockSizeSp = new JSpinner(…);
mTimeBlockShowWestChB = new JCheckBox(…);
mContent = new JPanel(…);
mContent.add(…);
mContent.add(mTimeBlockSizeSp, …);
mContent.add(mTimeBlockShowWestChB, …);
return mContent;
```

(a) before

```
mContent = new JPanel(…);
mContent.add(…);
mTimeBlockSizeSp = new JSpinner(…);
mContent.add(mTimeBlockSizeSp, …);
mTimeBlockShowWestChB = new JCheckBox(…);
mContent.add(mTimeBlockShowWestChB, …);
return mContent;
```

(b) after

Figure 6. A Method Where Most of the Subject Judged as Difficult

B. Discussion

In the experience, original methods were judged to be easier to read than their reordered ones in 4 out of the 20 methods. We investigated details of these 4 methods carefully, and obtained following characteristics.

- There are consecutive invocations of the same method for the same object.
- There are consecutive declarations of variables whose names are similar to one another.

They have a common point that a similar sequence of statements is considered to be one of the factors improving the readability. For example, Figure 6 shows a method where most of the subjects judged that the original was easier to read than the reordered. The arrows in the figure represent DUchains, and their labels represent their distances. In Figure 6(a), there are consecutive invocations of the same method for an object named *mContent*. The proposed technique moved two variable definitions at the beginning of the method close to their references. Many of the subjects judged Figure 6(a) that included consecutive *mContent.add*(...) to be easier to read than Figure 6(b) in which definitions and references of the two variables were placed consecutively.

Hence, it seems that not only the distance between variable definitions and their references but also similarity of consecutive statements is an important factor of appropriateness of statement reordering.

VI. Future Work

First of all, we are going to collect the other factors on ordering statements affecting the readability of source code with more experiments, and brush up the proposed reordering technique with them. Besides, we are going to develop an interactive environment for supporting reordering statements. In the environment, we are also going to put a function that users can specify their special intentions for ordering statements as new ordering constraints, such that consecutive statements in the original source code are kept the original order during the reordering.

VII. Conclusion

This paper presented a technique reordering statements to improve readability of source code. We applied the proposed technique to a Java system, and then we could identify opportunities reordering statements from 215 methods. Besides, we conducted an experiment with 44 subjects.The experimental results revealed that reordered methods had better readability than their originals in most cases. In addition, we obtained a knowledge that not only the distance between variable definitions and their references but also similarity of consecutive statements are important factor in considering the order of statements.

In the future, we are going to:

- collect the more factors on ordering statements;
- improve our technique based on them;
- develop an interactive reordering environment.

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] A. Goldberg, "Programmer as reader," *IEEE Software*, vol. 4, no. 5, pp. 62–70, Sep. 1987.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[3] M. Nakamura, A. Monden, T. Itoh, K. Matsumoto, Y. Kanzaki, and H. Satoh, "Queue-based cost evaluation of mental simulation process in program comprehension," in *Proc. of 9th IEEE International Software Metrics Symposium*, Sep. 2003, pp. 351–360.

[4] 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 the 2012 International Conference on Software Engineering*, Jun. 2012, pp. 233–243.

[5] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng*, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[6] A. Dunsmore and M. Roper, "A comparative evaluation of program comprehension measures," *The Journal of Systems and Software*, vol. 52, no. 3, pp. 121–129, Jun. 2000.

[7] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic segmentation of method code into meaningful blocks to improve readability," in *Proc. of the 18th Working Conference on Reverse Engineering*, Oct. 2011, pp. 35–44.