2012 19th Asia-Pacific Software Engineering Conference

SATA Workshop

Preprocessing of Metrics Measurement Based on Simplifying Program Strucutres

Yui Sasaki, Tomoya Ishihara, Keisuke Hotta, Hideaki Hata,
Yoshiki Higo, Hiroshi Igaki, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University,
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {s-yui,t-ishihr,k-hotta,h-hata,higo,igaki,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In software maintenance, grasping characteristics of software systems by metrics measurement is a basic activity. However, metrics do not always represent characteristics of software systems. For example, Cyclomatic Complexity is a metric counting the number of branches in a given module, and it does not consider its content. One factor that Cyclomatic Complexity becomes large is the presence of repeated structures such as consecutive if-else structures. However, if such a structure is a repetition of simple operations, humans would not recognize a difficulty to understand the source code. In this paper, we propose performing preprocessing for metrics measurement and a methodology of the preprocessing. The proposed preprocessing simplifies repeated structures in source code. By applying the proposed preprocessing to metrics measurement, we can find low-understandability modules more efficiently. Also, we compared results of metrics measurement with and without the proposed preprocessing on open source software systems. As a result, we confirmed that metrics measurement with the proposed preprocessing was more useful to find low-understandability modules than without it.**

***Keywords*-Empirical Study, Metrics Measurement, Software Maintenance**

I. INTRODUCTION

Measuring software is a fundamental technique of empirical software engineering. For example, many software metrics have been widely studied and proposed in fault prediction studies [1], [2]. *Mining Software Repositories* field has explored software repository data and has proposed many historical metrics for fault prediction models [3], [4]. Recent studies have shown the usefulness of such historical metrics compared to traditional complexity metrics [5], [6].

McCabe's Cyclomatic Complexity is one of the most popular complexity metrics [7]. A variety of empirical studies revealed the usefulness of Cyclomatic Complexity. It is said, however, Cyclomatic Complexity does not exhibit the proper complexity in object-oriented systems. Chidamber and Kemerer defined six complexity metrics in object-oriented designs, and Basili concluded that these metrics were useful to identify fault-prone modules than Cyclomatic Complexity [8], [9].

Not only fault prediction studies, but detecting refactoring candidates also have used software metrics [10]. Refactoring is a technique of improving the structure of a program without changing its external behavior. For example, long or complicated methods are less maintainability, and so these methods should be broken up into simple methods [11]. This operation is known as Extract Method refactoring, which is one of the most popular refactorings, and is often performed with other refactorings [12]. Because identifying the place to be refactored is the first step in refactoring process, long or complicated methods are often identified with Lines Of Code or Cyclomatic Complexity.

In order to perform activities of software maintenance, we must understand source code of the maintaining software. However, software understanding itself is costly. A fundamental task in software maintenance is reading and understanding source code [13], [14], [15]. Consequently, identifying low-understandability modules and improving their understandability lead to efficient software maintenance in the future.

However, Cyclomatic Complexity does not always represent the low-understandability of source code. Buse and Weimer investigated code readability with several software metrics [16]. They reported that though the low-level code features and well-known metrics (code churn, past bugs, warnings with a static analysis tool) had a significant level of correlation with code readability, Cyclomatic Complexity was not strongly related to readability. Also, Jbara et al. mentioned that there were some functions including successive single-instructions such as case-entries in switch-statements and if-statements in Linux Kernel, and these structures did not reflect perceived complexity [17]. Moreover, Cyclomatic Complextiy often underestimates the understandability of modules because it does not count jump instructions such as break- and continue-statements which break the "straight line" flow of execution [18]. As described above, Cyclomatic Complexity does not always work well in the context of finding low-understandability modules. Though there are tools that measure the modified Cyclomatic Complexity such as pmccabe [19], there is no empirical study with these tools.

As well as Cyclomatic Complexity, Lines Of Code metric does not always indicate understandability of source code. The authors think that, one factor of it is the presence of repeated structures. For example, a long method is considered as a typical target of Extract Method refactoring. However, if it consists of a repetition of similar operations, its cohesion should not be low. For example, a long method initializing GUI parts includes many assignment statements.

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.59

```
  1: public final Object getValoreIndirizzobenefattotrans(...) {
        ...
        if() { ... if() { ... if() { ... if() { ...
124:        if (soggetto != null) {
               ...
128:           else
129:          {
130:              ArrayIterator iter = ...;
131:              if (iter!=null && iter.size()>0) {
132:                   iter.reset();
133:                   IfIndirizzo recapito = ...;
134:                   if(...)
135:                        return ...;
136:                   else return null;
137:              }
138:              else
139:                   return null;
140:          }
141:        }
142:        else return null;
         } } } }
189: }
```

(a) a method including deep nest structures

```
  1: public static int getColumnIndex(final String sColumnName)
  2: {
  3:    if (sColumnName.compareToIgnoreCase(...) == 0)
  4:         return NDX_TI_ID_TITOLO;
  5:    else if (sColumnName.compareToIgnoreCase(...) == 0)
  6:         return NDX_TI_ID_COMPAGNIA;
        ...
204:    else if (sColumnName.compareToIgnoreCase(...) == 0)
205:         return NDX_FI_DESC_FILIALE;
206:    return -1;
207: }
```

(b) a method including repeated structures

Figure 1. Motivating Example

Such assignment statements can be considered as a repetition of similar operations. Also, we do not consider that a part of such a method should be extracted as a new method for reducing method size.

In this paper, we focus on the context that a user would like to find low-understandability modules, and propose a preprocessing of metrics measurement for finding such modules efficiently. In order to investigate the usefulness of the proposed preprocessing, we conducted an experiment on approximately 13,000 open source software systems. In the experiment, we compared results of metrics measurement with and without the proposed preprocessing. As a result, we confirmed that low-understandability modules were found more efficiently with the proposed preprocessing than without it.

II. Motivating Example

Figure 1(a) is a Java method, which has deep nest structures with many if-statements. The Cyclomatic Complexity of the method becomes 33, which reperesents this method has low understandability.

We found another method as shown Figure 1(b) whose Cyclomatic Complexity was 112 in the same software.

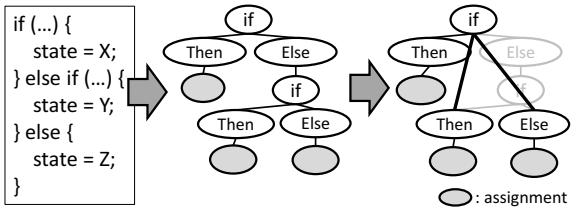


Figure 3. Transformation of else-if Statements

The value 112 of its Cyclomatic Complexity indicates that the structure in this method has lower understandability than Figure 1(a). Though the method certainly includes many branches, it is not difficult to understand because of repetition of simple instructions.

In this paper, we propose a new technique for measuring lower metric values in such a case of Figure 1(b). The key idea is **folding repeated structures**. If a method has repeated structures such as if-statements in Figure 1(b), the structures are folded as a single structure. Here after, we use a term *simplified source code*, which means source code where all the repeated structures in the original source code are folded. These operations are a preprocessing of metrics measurement. Applying the proposed preprocessing, the value of Cyclomatic Complexity of Figure 1(b) becomes lower than the value of Figure 1(a).

III. Proposed Preprocessing

Herein, we describe the operation of folding repeated structures, which is the proposed preprocessing of metrics measurement. The process is performed as follows:

1) constructing ASTs from input source code;
2) folding repeated structures on ASTs;
3) creating simplified source code from the folded ASTs.

Figure 2 shows how the source code of Figure 1(b) is manipulated by the proposed preprocessing. By using Figures 2 and 1(b), we explain the details of the proposed preprocessing.

Phase1. Constructing AST

First, we construct ASTs from input source code. A remarkable point is that, in the proposed preprocessing, else-if structures following if-statement as shown in Figure 1(b) are transformed. The transformation consists of (1) removing a node "Else" if there is only a node "If" following the node "Else", (2) connecting parent and child nodes of the removed node. Figure 3 shows an example of the transformation. By using the transformation, we can regard else-if structures as repeated ones in the Phase2.

Our approach does not focus on the details of statements such as variable names or conditional expressions. We do not consider AST nodes corresponding to such elements in source code.

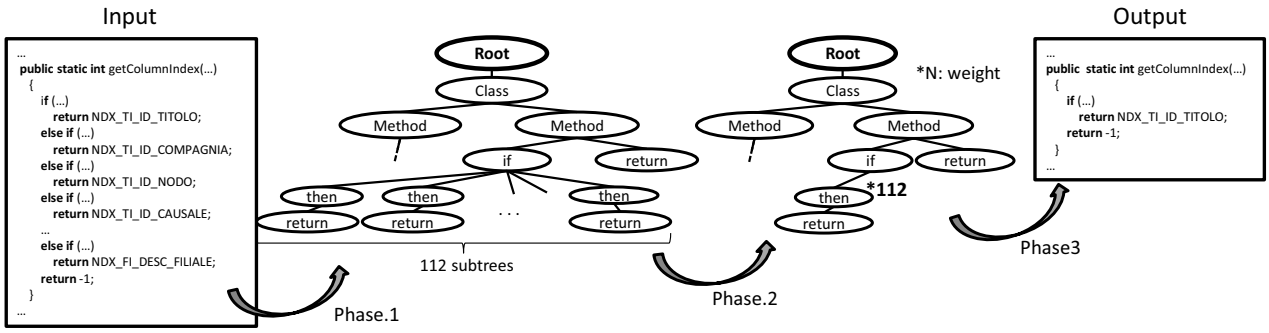
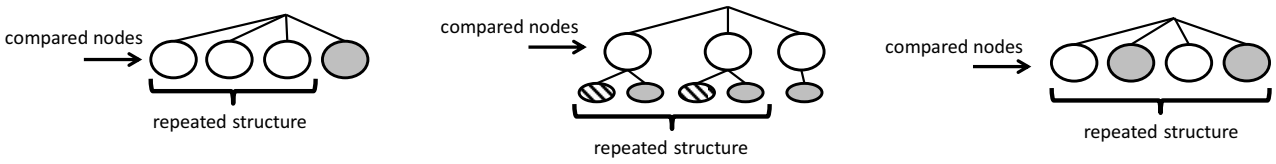


Figure 2. Proposed Preprocessing for Figure 1(b)



(a) pairs of leaves (b) pairs of branches (c) pairs of sequential nodes

Figure 4. Examples of Repeated Structures

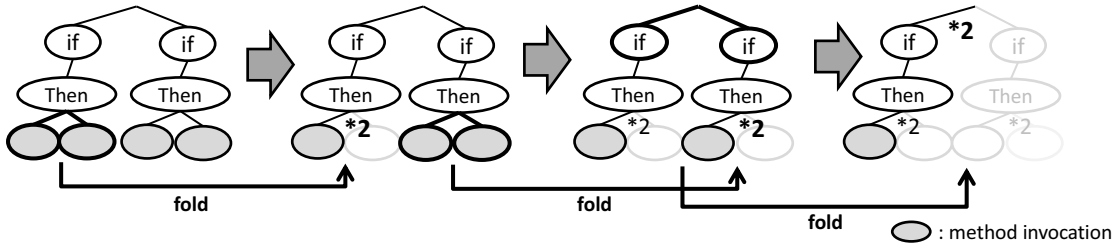


Figure 5. Example of Folding AST

Phase2. Folding Structures

AST sibling nodes are sorted in the order of the appearance on the source code. If there are consecutive similar structures in sibling nodes, they are regarded as repeated structures. If a pair of nodes satisfies the criteria in Table I, the nodes would be regarded as similar to each other. We describe "weight" in Table I later. ASTs in Figure 4(a) and 4(b) are regarded as repeated structures based on the criteria.

In addition, not only a single node but also sequential nodes are compared as one-side of the pair in AST sibling nodes. For example, in the following source code, there are consecutive two statements (assignment statement and method invocation statement).

```
comparator = new ObjectIdentifierComparator();
cb.schemaObjectProduced( this, "2.5.13.0", comparator );
comparator = new DnComparator();
cb.schemaObjectProduced( this, "2.5.13.1", comparator );
```

In order to handle repetitions of such multiple statements, if each pair of nodes in a pair of sequential nodes satisfies the criteria, it would be similar to each other. Consequently, we can regard structures of Figure 4(c) as repeated ones.

Folding repeated structures consists of two operations, which are (1) removing all the elements of the repetitions except the first element, and (2) adding the weight of the number of elements to the first element.

There are nested repeated structures in the source code. For example, in the following source code, there are repetitions of method invocations and repetitions of if-statements including the method invocations.

Table I
CRITERIA FOR SIMILARITY OF NODE

compared nodes	criteria for similarity
a pair of leaves	type of statements and weights are the same
a pair of branches	its subtrees are isomorphic

```
if (null != storepass) {
  cmd.createArg().setValue("-storepass");
  cmd.createArg().setValue(storepass);
}
if (null != storetype) {
  cmd.createArg().setValue("-storetype");
  cmd.createArg().setValue(storetype);
}
```

In the case of nested repeated structures, the folding operation is applied to nodes in order from leaf to root. In this example, firstly method invocations are folded, then if-statements are folded.

This process is achieved by a postorder traversal. If a traversed node contains children, we get them. Then, if we find a pair of similar (sequential) nodes in the children, they are folded. These operations are repeatedly performed until there are no similar nodes on sibling nodes. Figure 5 shows how the above source code is folded.

There are various repeated structures in source code such as introduced by [20]. We confirmed that all of the repeated structures in [20] were folded with this approach.

Though Figure 1(b) has many branches, each branch node has similar statements as its children. Therefore, all the structures are folded.

Phase3. Providing Simplified Code

Finally, the proposed preprocessing generates simplified source code from the simplified ASTs.

IV. CASE STUDY

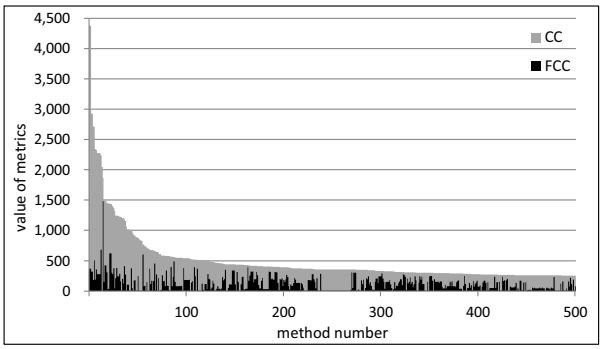
We implemented a prototype tool for Java source code, and evaluated the proposed preprocessing by applying the tool to about 13,000 software systems (see Table II), which are open to the public in [21]. In this evaluation, we calculated some metrics for each method in all the target software. The purpose of the case study is to reveal the answers to following questions.

1) Do results of metrics measurement change by using the proposed preprocessing?
2) Is there any relationship between metrics measurement result and understandability of the source code?

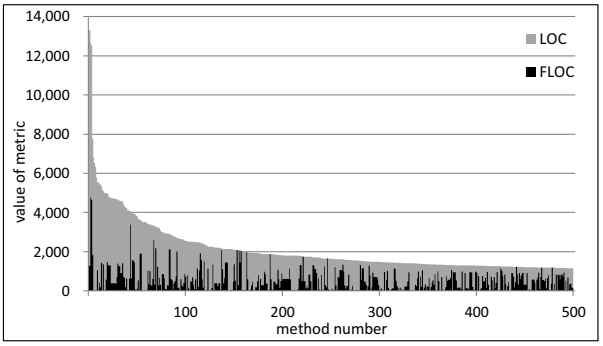
In the remainder of this section, we describe experimental results and answers for each question.

SetUp

In this experiment, we measured Cyclomatic Complexity and Lines Of Code. Cyclomatic Complexity was measured by counting branches in source code. We regarded that the following blocks were branches.



(a) Cyclomatic Complexity



(b) Lines Of Code

Figure 6. Difference of Metrics Values

- case-entry in switch-statement
- do-statement
- for-statement
- foreach-statement
- if-statement
- catch-statement in try-statement
- while-statement

Also, Lines Of Code was measured without comment lines and blank lines. In the remainder of this section, metrics used in this experiment are called as shown in Table III.

Ex1.Comparison between Metrics Values

At first, we compared values between metrics with and without the proposed preprocessing for each method. Figure 6 shows the top 500 methods on all the target software without the proposed preprocessing. We found that particularly high CC or LOC values were greatly reduced with the preprocessing. That means these methods include

Table II
OVERVIEW: UCI SOURCE CODE DATA SETS

# of software	13,193
# of methods	18,366,094
total lines of code	361,663,992

Table III
TARGET METRICS

	Cyclomatic Complexity	Lines Of Code
without the preprocessing	CC	LOC
with the preprocessing	FCC	FLOC

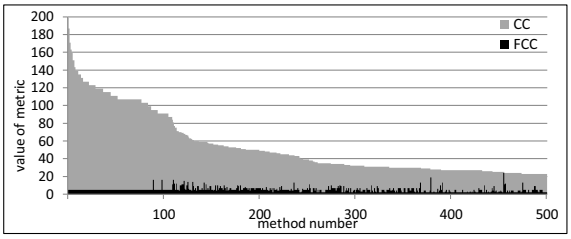


Figure 7. Difference of Metrics Values on One Software

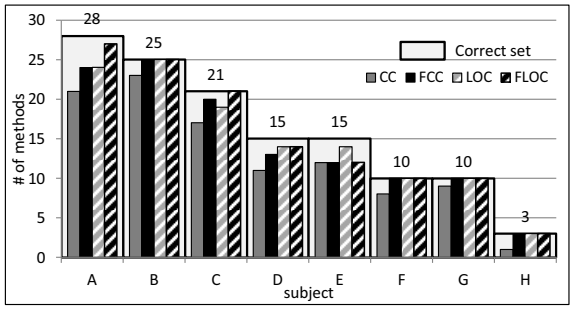
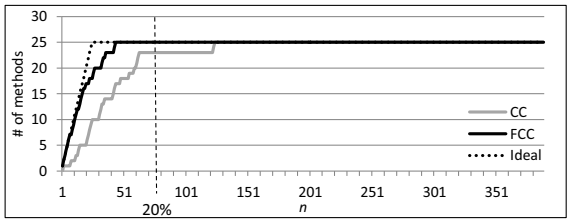
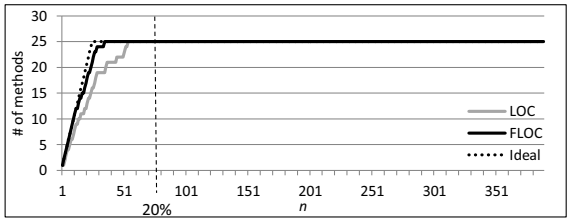


Figure 8. Comparison between Human Consideration and Metrics



(a) Cyclomatic Complexity



(b) Lines Of Code

Figure 9. The Number of Correct Methods in Top n

many repeated structures. The highest CC is 4,371, and its method includes 200 if-statements. 191 out of the 200 if-statements have only one large switch-statement including so many case-entries. The case-entries in each of such switch-statements were folded.

Each software has a different gap between CC and FCC measurement. For example, some software have few methods including repeated structures, and the others have many methods including repeated structures as shown in Figure 7. Though this software has many methods whose CC are over 100, almost all of them have repeated if-structures, and so those FCC are only 5. Furthermore, a method with the highest FCC ranks at the 455th in the CC order, which means that it is not easy to find this method with CC.

In order to grasp how measurement results for every target system were changed by the proposed preprocessing, we used an indicator, *ConcordanceRate*. *ConcordanceRate* means the rate how measurement results with and without the preprocessing share the same modules in top n. *ConcordanceRate* for Cyclomatic Complexity is defined as the following formula.

$$ConcordanceRate(n) = \frac{|CC(n) \cap FCC(n)|}{n}$$

$CC(n)$ and $FCC(n)$ are sets of methods that are top n in the CC or FCC order, respectively. The more there are common methods between $CC(n)$ and $FCC(n)$, the higher *ConcordanceRate* becomes. We determined the threshold n as the number of the top 20% methods for each software. If there is no difference between top 20% CC and top 20% FCC methods, *ConcordanceRate* becomes 1.

Investigating for all the target software, we found that about 85% of the target software had at least one different method in their top 20%.

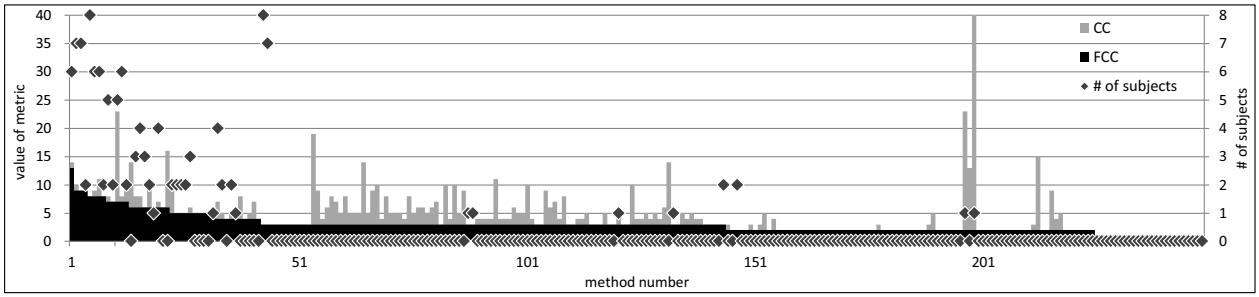
Ex2.Comparison between Understandability of Source Code and Metrics Values

We evaluated metrics with the preprocessing based on the human consideration. In this experiment, 8 people (faculty staffs and students) in the department of computer science of Osaka University were joined as subjects. This experiment was conducted along with the following steps.

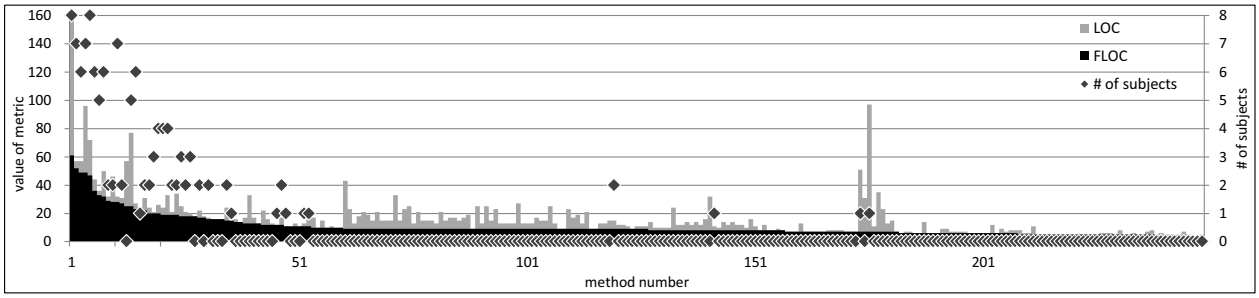
1) Every subject judge whether each method is difficult to understand or not.
2) Every method judged as difficult to understand is called a **correct method**, and its set is called **correct set**.
3) Methods in the target systems are sorted in the order of each metric with and without the proposed preprocessing. We compare the number of correct methods included in top 20% of the sorted methods.

The subjects judged all the methods by browsing the source code of them. We did not provide metrics values to the subjects during the experiment. We selected not so large software from 13,000 software systems as the target because the subjects checked the source code manually. The target software is JCap, which includes 389 methods. The values of ConcordanceRate of Cyclomatic Complexity and Lines Of Code for this software are 69% and 67%, respectively.

Figure 8 summarizes the result. In the case of subject "A", there are 28 correct methods. The value of Y-axis represents the number of correct methods included in the top 20%. As shown of this graph, both Cyclomatic Complexity



(a) Cyclomatic Complexity



(b) Lines Of Code

Figure 10. Relationship between Metrics Value and The Number of Subjects

and Lines Of Code with the preprocessing present more correct methods than without it in the top 20%. That is, for subject "A", the preprocessing is useful to detect low-understandability methods efficiently. Totally, in the case of Cyclomatic Complexity, for 7 out of 8 subjects, the number of correct methods with the preprocessing is larger than without it. On the other hand, in the case of Lines Of Code, the preprocessing is more useful for only 2 out of 8 subjects. For 4 out of remaining 6 subjects, both results with and without the preprocessing included all the correct methods, so that we could not see differences between them.

Next, we investigated the number of correct methods included in top n by changing n. Figure 9 shows the investigation results for subject "B". The dotted line is the ideal curve, which means that all the correct methods appear in the top without including other methods. We can see that (1) the curve of FCC is nearer than CC, and (2) the curve of FLOC is nearer than LOC, respectively. That means the preprocessing is useful for efficient detection of low-understandability methods. Especially, in the case of Cyclomatic Complexity, there is a significant difference with and without the preprocessing. In the top 10 methods, only 2 correct methods were included without the preprocessing whereas 9 correct methods were included with it. We conducted the same investigation for the other subjects. As a result, for 7 out of 8 subjects, we found that the proposed preprocessing was useful for detecting low-understandability methods efficiently.

V. DISCUSSION

Figure 10 shows metrics values of all the methods included in the target system of Ex2. In addition, we plotted the number of subjects who judged the method as difficult to understand. In X-axis, the methods are arranged in the descending other of FCC/FLOC. We confirmed that there was a correlation between FCC/FLOC and the number of subjects who judged the method as difficult.

However, there are some outliers. First, there are false-positive methods. Though these methods had high FCC or FLOC values, all the subjects judged as not difficult to understand. For example, there is a method which includes 3 switch-statements, and the number of case-entries in each of the switch-statements is different. This method has high FCC and FLOC, because 3 switch-statements are not regarded as repeated structures. As shown this example, difference of the repeat count may be not significant for humans. Hence, it would be required not to consider the difference of the repeat count.

Also, there are other kinds of false-negative methods. These methods were judged as difficult to understand by some subjects, and their metrics values were reduced by applying the preprocessing. From this result, there is possible that the length of source code affects to understandability. Simplified source code has no information about the repeat count and metrics values of the original source code. Hence, it is necessary to use not only metrics values with the preprocessing but also ones without the preprocessing or

the other metrics related to readability of source code for identifying low-understandability source code.

VI. THREDS TO VALIDITY

In Ex2, we selected only a single software system. This system is relatively small, and each method included in the system is not so long. Also, in the selected system, most of repeated structures were repetition of simple case-entries and simple if-else-statements. Thus, we could not evaluate multiple statements folding and recursive folding. If we apply the proposed preprocessing for more systems, we can evaluate the folding operation for a variety of repeated structures.

Subjects in Ex2 have knowledge of software engineering, which probably has an impact on understandability of source code. If we conduct the same experiment for other people (for example, people who hardly have experiences of programming), we may obtain a different result.

VII. CONCLUSION

This paper addressed that the presence of repeated structures have a negative impact on metrics measurement, and proposed a technique that remove repeated structures. The proposed technique is used as a preprocessing of metrics measurement from source code. We compared results of metrics measurement with and without the proposed preprocessing on approximately 13,000 open source systems. The comparison results showed that most of the target methods included repeated structures and metrics values were significantly different between with and without the preprocessing. Also, we investigated the relationship between subject's understandability of target methods and their metrics values with and without the preprocessing. We confirmed that metrics values with the preprocessing had stronger correlation with the understandability than ones without the preprocessing.

In the future, we are going to this research as follows:

- We tune the proposed technique. For example, not considering the difference of repeat count in each repeated structure will have a significant impact on metrics measurement.
- By using the key idea that folding repeated structures, we are going to develop a software tool for supporting software development and maintenance. For example, visualizing structures in a given method with the folded functionality will help users to understand the method.

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] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[2] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of 28th International Conference on Software Engineering.*, May 2006, pp. 452–461.

[3] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic review of fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. PrePrints, no. 99, pp. 1–31, 2011.

[4] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of 34th International Conference on Software Engineering.*, Jun. 2012, pp. 200–210.

[5] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of 26th IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[6] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of 30th International Conference on Software Engineering*, May 2008, pp. 181–190.

[7] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[8] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[9] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[10] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, Mar. 2001, pp. 30–38.

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

[12] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 287–297.

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

[14] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, 2010, pp. 223–226.

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

[16] 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.

[17] A. Jbara, A. Matan, and D. G. Feitelson, "High-mcc functions in the linux kernel," in *Proceedings of 34th International Conference on Program Comprehension*, Jun. 2012.

[18] J. J. Vinju and M. W. Godfrey, "What does control flow really look like? eyeballing the cyclomatic complexity metric," in *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 154–163.

[19] "pmccabe," http://http://manpages.ubuntu.com/manpages/lucid/man1/pmccabe.1.html.

[20] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Information and Software Technology*, vol. 49, no. 9–10, pp. 985–998, Sep. 2007.

[21] C. Lopes, S. Bajrachaya, J. Ossher, and P. Baldi, "Uci source code data sets," http://www.ics.uci.edu/~lopes/datasets/.