Refactoring Effect Estimation based on Complexity Metrics

Yoshiki Higo[†] Yoshihiro Matsumoto[†] Shinji Kusumoto[†] Katsuro Inoue[†]

[†]Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{higo, y-matsu, kusumoto, inoue}@ist.osaka-u.ac.jp

Abstract

Refactoring is a set of operations to improve maintainability or understandability or other attributes of a software system without changing the external behavior of it, and it is getting much attention recently. However it is difficult to perform appropriate refactorings since the impact of refactoring should justify the cost. Therefore, before a refactoring is really performed, the effect and the cost of it should be estimated. The estimation makes it possible for us to adequately assess whether each refactoring should be performed or not. This paper shows that it is difficult for developers to perform appropriate refactorings, and proposes a method estimating refactoring effect. The method has been implemented as a software tool, and a case study was conducted with it. The result of the case study showed that the estimation of the tool helped a developer of the target software system to perform an appropriate refactoring.

1 Introduction

Refactoring is one of trenchant countermeasures to handle software systems getting bigger and more complex recently. The concept of refactoring is to improve the internal structure of a software system for future development or maintenance without the external behavior of it. Nowadays, the importance of refactoring has been accepted widely.

Since a certain cost is required to complete a refactoring, the impact of refactoring should justify the cost. However, it is difficult to precisely estimate the effect of refactorings and inappropriate refactorings may be performed instead of appropriate ones. Inappropriate refactorings become software systems less maintainable, or requires much cost to operate source code change and regression test.

This paper proposes a method estimating refactoring effect. After developers input refactorings that they are going to perform, the method outputs the effect estimation result of each of the refactorings. Using the proposed method, developers can approximately know the effect of the refactorings that they are going to perform, which makes it possible to objectively assess whether or not they should really perform the refactorings.

The method uses CK metrics suite, which is one of the most popular and widely-accepted software complexity indicators. In other words, the method estimates refactoring effect by considering structural changes of the source code, which is described as follows,

- how coupling between classes will change,
- how cohesion of each class will change,
- how inheritance relationships between classes will change.

The method outputs quantitative result of the effect estimation from the viewpoint of the three type changes.

The method has been already implemented as a software tool, and a case study was conducted by using the tool. The target of the case study is a software system developed by a master student of our lab. The result of the case study indicates the followings.

- It is difficult for developers to assess which refactoring is better than the other refactorings.
- By using the proposed method, the developer of the target software system could perform effective refactorings.

In Section 2, this paper describes about CK metrics, and the proposed method is presented in Section 3. Section 4 introduce the tool that we have developed. In Section 5, we show the case study, and in Section 6, we discuss the proposed method and the case study. In Section 7, related works are presented. Finally, we conclude our paper with a few remarks in Section 8.

2 CK Metrics suite

Measuring complexity metrics of a software system is one of usual practices to evaluate maintainability of it. The greater measurement result is the more complex the software system is, in other words, the more difficult it is to maintain the software system. One of the most popular and widely-accepted complexity metrics is CK metrics suite, which is proposed by Chidamber and Kemerer [3].

CK metrics suite consists of six metrics, and each of them measures complexity of object-oriented software system from the different viewpoints. Basili et al. experimentally evaluated CK metrics suite and some other software metrics, and concluded that CK metrics suite is a better indicator to estimate occurrences of faults than other metrics [1].

Each of CK metrics is count by classes, briefly described as follows.

WMC (Weighted Methods per Class) : This metric is the sum of the complexities of the methods defined in the target class. Up to now, several methods measuring method complexity have been proposed, and *Cyclomatic number* [17] and *Halstead complexity measurement* [10] are commonly used. Sometimes, this metric is simply the method count for the class.

DIT (Depth of Inheritance Tree) : This metric represents the depth of the target class in the class hierarchy.

NOC (Number Of Children) : This metric represents the number of classes directly derived from the target class.

CBO (Coupling Between Object classes) : This metric represents the number of classes coupled with the target class. In the definition of this metric, there is a coupling between two classes if and only if a class refers to methods or fields of the other class.

RFC (Response For a Class) : This metric is the sum of the number of local methods and the number of remote methods. A local method is a method defined in the target class, and a remote method is a method invoked in any of local methods with the exception that local methods invoked in any of local methods are not counted as remote methods.

LCOM (Lack of Cohesion in Methods) : This metric represents how much the target class lacks cohesion. This metric is calculated as follows;

Take each pair of methods in the target class. If they access disjoint set of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

$$LCOM = \begin{cases} P - Q & (if P > Q) \\ 0 & (otherwise) \end{cases}$$

It is noted that the definition of LCOM has some drawbacks, and other researchers have re-defined LCOM with other definitions [11, 12].

3 Proposed Method

This section describes the proposed method. The method uses CK metrics to estimate the effect of refactorings. Figure 1 illustrates the overview of the proposed method. The proposed method takes the source code of the target system and the refactoring that developers are going to perform, and it outputs how much the refactoring is effective based on the difference of the metrics between the original source code and the revised one. The remainder of this section describes each step marked on Figure 1.

STEP1

The whole of the target program is parsed and a structure representing it is constructed. The structure includes all information required to calculate CK metrics.

STEP2

CK metrics are calculated from the structure constructed in STEP1. These metrics represent the complexity of the original program.

STEP3

The refactorings that developers are going to perform are input. They have to input two kinds of information for each refactoring: one is where of the target program is refactored; the other is how the part is refactored.

STEP4

The structure is changed based on the refactorings input in the previous step. However it is impossible to automatically perform all operations forming the refactoring. Some operations require developer's intention. Here, we assume that a developer is going to move method *a1* in class *A* to class *B*, shown in Figure 2.

Firstly, all classes affected by the refactoring are identified automatically. In this example, all classes

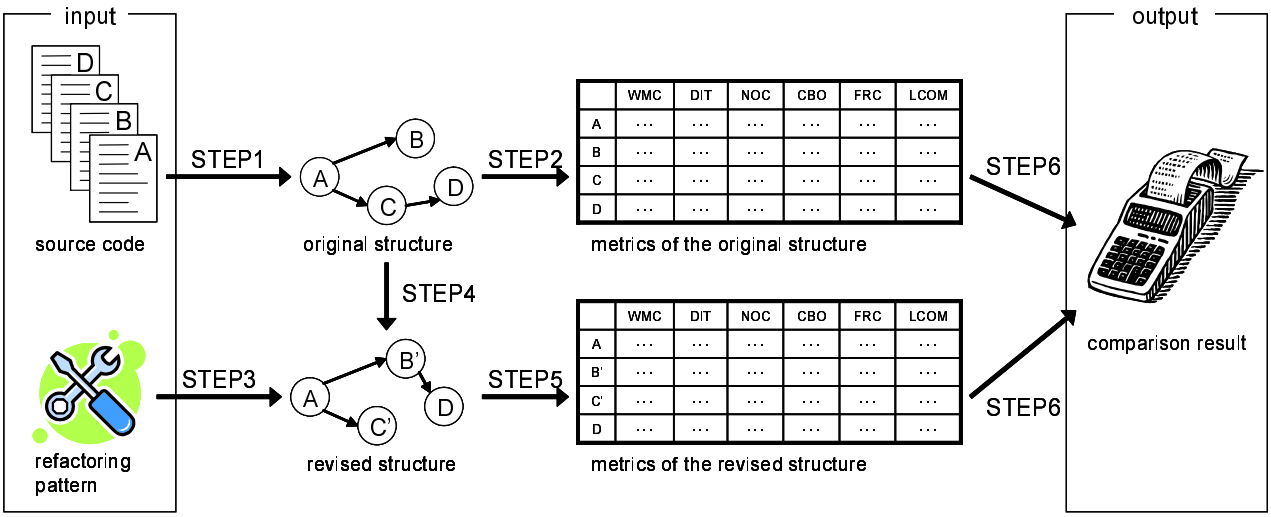
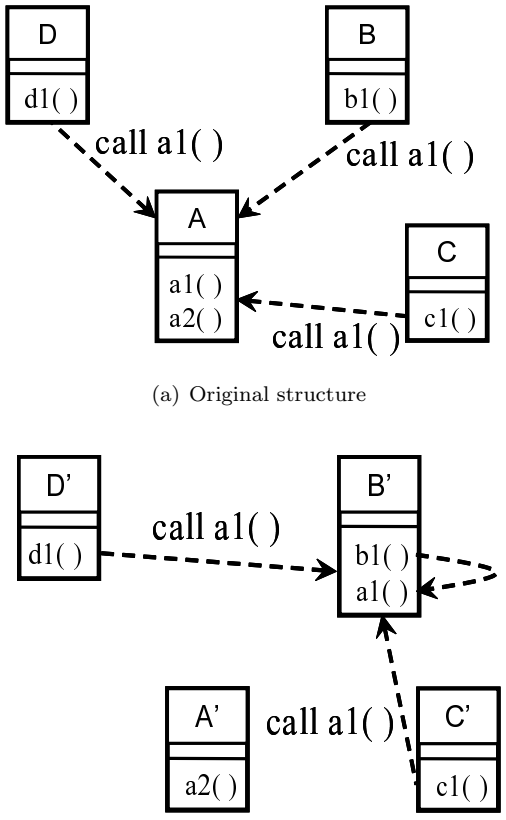


Figure 1. Overview of the proposal



(a) Original structure

(b) Revised structure

Figure 2. Refactoring example

in which *a1* is invoked are affected. Since the example refactoring moves *a1* to another class, all of the *a1* invocations have to be changed. As shown in Figure 2(a), classes *B*, *C*, and *D* are affected by the refactoring because they have *a1* invocations.

Next, all of the identified classes are changed. The following describes *automatic change* and *interactive change* respectively. Figure 3 illustrates both changes.

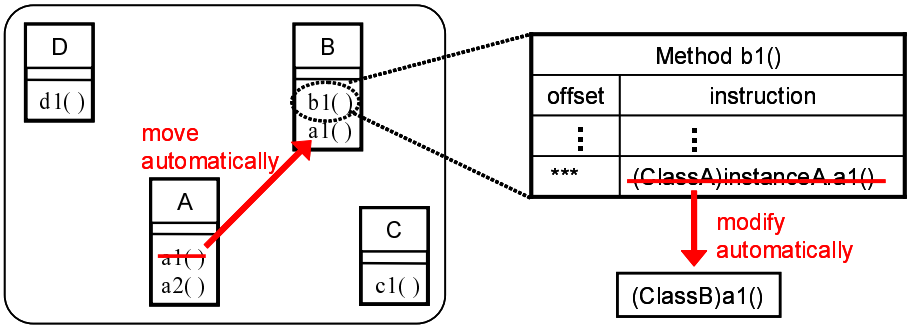
automatic change : This kind of change is a completely automatic processing, not require developer's interventions. In the example, The following two changes are classified into *automatic change*.

- Delete method *a1* from class *A*, and add it to class *B*.
- In class *B*, *a1* invocations are changed as internal method invocations.

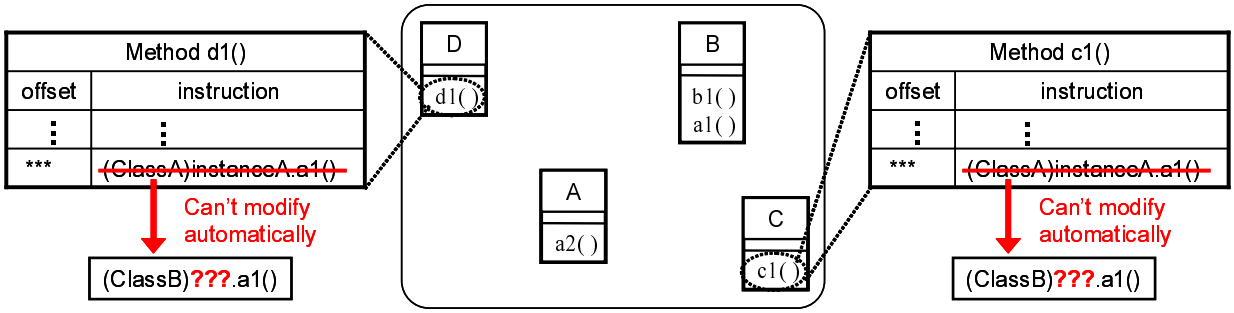
interactive change : Some operations forming a refactoring require developer's interventions, that is, they cannot be completed automatically. In the example, a developer needs to determine which instance invokes method *a1* of class *B* in class *C* and *D*. The method automatically identified such interactive change operations and asks to him/her. After receiving answer, the method changes the structure based on it.

STEP5

CK metrics are calculated from the structure changed in the previous step. These metrics represent



(a) Automatic change



(b) Interactive change

Figure 3. structure change based on refactoring pattern

the complexity of the revised program.

STEP6

In the final step, the method outputs how the complexity of the target software will change by performing the refactoring. The output is the comparison result between CK metrics of the original structure and ones of the revised structure.

The comparison is performed on each metric. We call the comparison result *change rate*. The following formula is for calculating the change rate of metric WMC of the example refactoring;

$$\frac{\sum_{y\in A',B',C',D'} WMC(y) - \sum_{x\in A,B,C,D} WMC(x)}{\sum_{x\in A,B,C,D} WMC(x)}$$

A, B, C, and D are classes in the original structure, and A', B', C', and D' are ones in the revised structure. Also, $WMC(x)$ is the value of metric WMC for class x. The change rates of other metrics are calculated by the same formula.

In this step, only the classes affected by the refactoring are used for calculating *change rate*. For example, if there were class E which is not affected by the refactoring, E wouldn't be used for calculating *change rate*.

4 Implementation

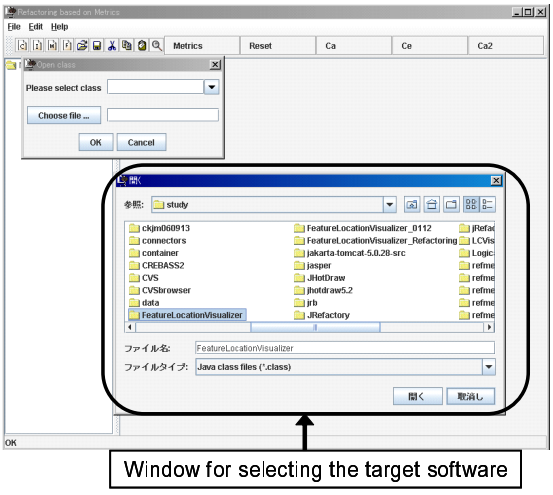
We have implemented a software tool based on the proposed method. At the present time the tool can process only the software systems written in Java language.

The tool uses java bytecode as the structure representing the program. It may be more natural to use abstract syntax tree (AST) or program dependency graph (PDG). However, there are useful and practical tools/libraries to handle bytecode, and we thought that using them allowed us to develop a software tool at low cost.

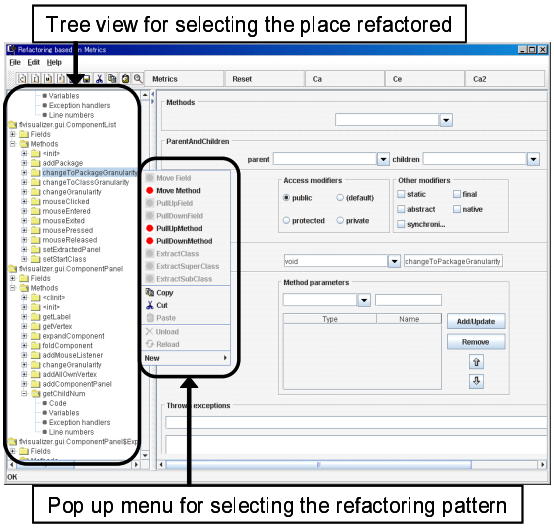
The software tool that we developed consists of the following components.

- Input and output component,
- Bytecode change component,
- Metrics measurement component.

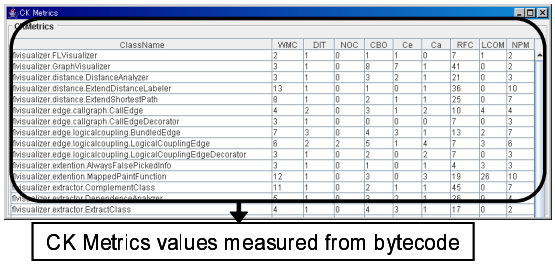
The remainder of this section describes each component closely.



(a) Input the target software



(b) Input how the software is refactored



(c) Outout the CK-metrics

Figure 4. Snapshot of input and output component

Input and output component

This component is the user interface of the software tool. Developers input the target software system, and specify where they are going to perform a refactoring (see Figure 4(a)) and which refactoring pattern they are going to apply (see Figure 4(b)). The tool provides the CK metrics values measured from the bytecode, and also provides the change rate through this component (see Figure 4(c)). This component plays a role of STEP1 and STEP3 in Figure 1.

Bytecode change component

This component change the structure based on the refactoring specified by developers. This component uses *Class Construction Kit*[1] to process bytecode. Based on the refactoring that developers are going to perform, *Class Construction Kit* changes the bytecode. Before changing the bytecode, this component identifies all the classes affected by the refactoring. If the changes cannot be processed automatically, this component requires developers to input how to change the bytecode (See STEP3 of the proposed method described in Section 3.). This component plays a role of STEP4 in Figure 1.

At present, the method supports some of refactoring patterns involving structural change [9], which are remarked below.

- *Move Field*, *Pull Up/Down Field*,
- *Move Method*, *Pull Up/Down Method*,
- *Extract Class*, *Extract Super/SubClass*,

Metrics measurement component

This component measures CK metrics from the bytecode of the original program and the revised one, and also computes change rates of them. This component uses *ckjm*[2] to measure CK metrics from bytecode. The proposed method uses *Cyclomatic number* for calculating metric WMC. The metrics values and the change rates are provided to developers through the

[1] *Class Construction Kit* is a tool for visualize and change bytecode, it is implemented with BCEL and Swing [15]. If developers want to know the structure of a software system without the source code, this tool is very useful. Also bytecode change with this tool allows them to downsize the bytecode or process something that is difficult with the source code.

[2] *ckjm* is a metrics measurement tool. This tool measures CK metrics from java bytecode, and outputs the result in several formats like CSV [5]. It is implemented with BCEL. *ckjm* also measures three metrics Ce, Ca, and NPM besides CK metrics as you can see in Figure 4(c). But, in our method threse three metrics are not used. If you are interested in the metrics, please refer to [5].

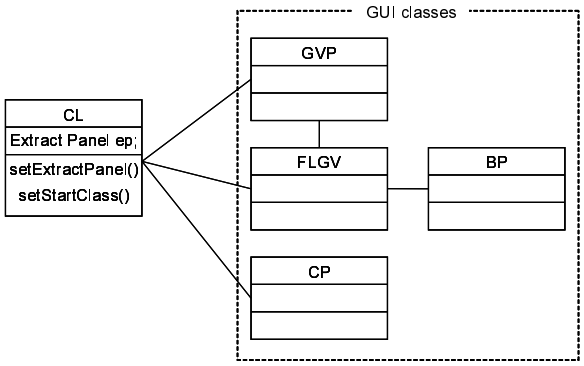


Figure 5. Problem of existing code

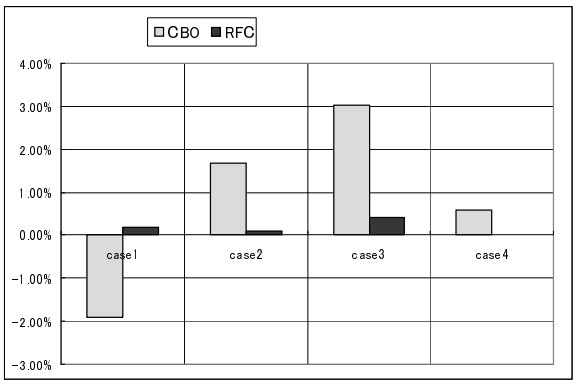


Figure 6. Comparison of 4 candidates

input and output component. This component plays a role of STEP2, STEP5, and STEP6 in Figure 1.

5 Case Study

5.1 Outline

We conducted a case study to evaluate the usefulness of the proposed method. The target of this case study is a program developed by a master student of our lab, and it has been maintained for a year. The program is written in Java language, the number of classes is 37, and the LOC is 4,815.

We manually identified which modules of the target software had undesirable conditions with the master student. After the identification, we thought out 4 refactoring candidates to improve the modules. The below describes the problem and the refactoring candidates.

Problem

Class ComponentList (CL) was designed not related to GUI originally. However, after 1-year maintenance, CL has a function related to GUI: There is a field whose type is ExtractPanel, which is one of GUI parts, and also there are two methods setExtractPanel and setStartClass referring to the variable. Figure 5 illustrates the coupling between class CL and GUI classes. Each connector in this figure means that there is a coupling between the classes being in its both ends. We can see that class CL has couplings with GUI classes GVP, FLGV, and CP.

Refactoring Candidates

We thought that the variable and the methods should be moved to another class related to GUI. The following 4 classes are the candidates that they are moved to.

CASE1 FeatureLocationGraphViewer (FLGV)

CASE2 BirdPanel (BP)

CASE3 ComponentPanel (CP)

CASE4 GraphViewPanel (GVP)

The proposed method estimates the refactoring effectiveness of each candidate. On the other hand, the master student judged a candidate to be the best by his subjectivity.

5.2 Master Student's Decision

The master student selected CASE3 because he thought that class CP had a similar function to the methods moved. His decision was based on his instinct rather than objective basis like bug information or software metrics or design patterns.

5.3 Proposed Method's Estimation

Table 1 and Figure 6 illustrate the estimation of the proposed method. They represent the change rates of

Table 1. Change rates of each refactoring

	CASE1	CASE2	CASE3	CASE4
WMC	0.00	0.00	0.00	0.00
DIT	0.00	0.00	0.00	0.00
NOC	0.00	0.00	0.00	0.00
CBO	-1.90	1.67	3.03	0.60
RFC	0.18	0.09	0.40	0.00
LCOM	0.00	0.00	0.00	0.00

metrics values between the original program and the revised one. Figure 6 represents only the metrics whose change rates are not zero due to limitations of space. They tell us the following things.

- All refactorings will not change the sum of metrics WMC, DIT, NOC, and LCOM at all.
- All refactorings will have a little bit of change on the sum of metric RFC.
- CASE1 will be able to greatly reduce the sum of metric CBO while all of the other refactorings will increase.

We predicted that change rates of DIT and NOC would be zero because all refactorings don't change the inheritance hierarchy. This refactoring consists of *Move Field* and *Move Method*, and doesn't reconstruct the insides of the methods, which is the reason why the sum of metric WMC was zero: we used *Cyclomatic number* for calculating metric WMC. We did not predicted that the sum of LCOM doesn't change at all, it seems to be a coincidence.

After the estimation, we actually performed all of the four refactorings on the source code, and measured CK metrics from the revised source code and computed the change rates. All of the metrics values and the change rates were the same as ones measured form the revised bytecode, which means that the estimation by using bytecode is precise.

6 Discussion

6.1 Validity of the Method

Unfortunately, no research has revealed obvious foundation that good refactorings lead to lower CK measures. There are various reasons for performing refactorings, so some good refactorings should lead to higher CK measures. For example, applying design patterns to the source code for getting higher expandability for the future often increases coupling between the specific classes. The method should not match such refactorings as applying degisn patterns.

When a maintainer performs a refactoring, there is a obvious goal of it. For example,

- Class C is too big and complicated, it should be divided into some small classes,
- Method M locates in an inappropriate class, it should be moved to another class.

By using the method, the maintainer can know whether the goal can be accomplished with the refactoring. Moreover, side-effects of the refactoring can be represented by the ratio of CK metrics, so the maintainer can avoid regret the refactoring after acturelly performing it.

6.2 About the Case Study

In this case study, the proposed method estimated the refactoring effects of 4 refactoring candidates. All of them are conceived by us and the master student who is the developer of the target program.

The master student considered CASE3 as the best in the four refactoring candidates while the proposed method estimated that CASE3 is not an effective refactoring because of the increase of CBO. We reported the estimation result of the proposed method to the master student after his determination. He recognized that the estimation was better than his determination, and adopted the refactoring recommended by the proposed method.

From the result of this case study, we can say the followings.

- The master student subjectively determined which refactoring seems to be more effective than other refactorings.
- The refactoring that he selected was not effective because of increase of complexity.

From the above results, we can conclude it difficult for developers to figure out the complexity of the software system. And it is also more difficult not to increase the complexity unnecessarily in performing refactorings. Therefore, the method proposed in this paper can appropriately navigate developers to effective refactorings, and it is useful in the refactoring process.

This case study has the following limitations.

The examinee is a single master student

In order to get more calculable evidences that the proposed method practically helps developers to perform effective refactorings, we need to conduct experiments involving more examinees. Also, we should conduct experiments for different level programmers: in this case study, examinee is a single master student. If the examinee was a real software engineer, he might be able to select the refactoring regarded the best by the proposed method. The experience of the person doing the refactoring is a factor in case study outcome.

The target program size is not practical

The target program of the case study was developed by a single master student. Therefore its size is not large: the number of classes is 37, and the LOC is 4,817. We should conduct more experiments on more practical-size software systems. But, this case study revealed that it is difficult to perform effective refactorings on even a small-size program. Therefore, we can consider that it is much more difficult on practical-size software systems.

Only the change of complexity metrics values was considered as the effect of refactorings

In this case study, Only the complexity change of classes was considered as the effect of refactorings. However, in reality, completing a refactoring requires the source code modification cost and the regression test cost and might further costs for something. In order to estimate refactoring effectiveness more precisely, we have to consider those costs.

7 Related Work

Leitch et al. has proposed a method for assessing maintainability benefits of refactoring [16]. Their goal is to develop a well-defined way for estimating the costs and benefits of refactoring. In their context, the cost of a refactoring is operations for regression tests to guarantee that the refactoring doesn't change the observable behavior of the software system, and the benefit is the maintenance saving by the refactoring. Both the cost and the benefit are calculated based on the COCOMO II model, which was proposed by Boehm et al. [2]. In the method, control and data dependency graphs are required to compute maintenance costs of the original system and the revised one. However, those graphs are constructed by manual code inspection, which means it difficult to apply the method to middle-scale or large-scale software systems. On the other hand, our proposed method almost automatically [3] calculates metrics of both the original system and the revised one. Therefore, we can say that our proposed method is more scalable.

Counsell et al. has divided seventy two refactoring patterns, which are described in Fowler's book [9], into categories from the viewpoint of regression test [6]. Some refactorings change an interface of the classes of the software system, and others do not. If the interface is not changed by a refactoring, the set of existing tests can be adapted to the revised software system. That requires a little cost to guarantee the external behavior preservation. However, if the interface was changed by a refactoring, new test suite has to be created for confirming the correctness of the refactoring, which requires much more costs. The categories were originally made by Deursen et al. [8], and were extended by Counsell et al. A refactoring pattern sometimes includes other refactoring patterns: for example, *Extract Class* pattern includes *Move Method* pattern and *Move Field* patterns. The original refactoring pattern categorization doesn't consider dependency between refactoring patterns while the extended categorization does consider. This categorization allows us to estimate the cost of regression cost, and may be a indicator whether a refactoring should be performed or not.

Kataoka et al. suggested a method measuring the effect of refactorings [14]. They use coupling between methods as the indicator of refactorings effects. The coupling is calculated from three kinds of program elements, *return-value*, *arguments*, and *shared-variables*. The couplings of the original system and the revised one are compared for evaluating the refactoring. This method is applied to the system after the source code is changed. That is, this is not a estimation method of refactoring but an evaluation of them. However, we think it is possible to calculate this metric from the structure representing the source code as well as CK metrics. This metric should be able to estimate the effect of refactorings.

Maruyama has proposed an undo mechanism for refactorings [18]. Existing editors and IDEs like Eclipse support undo for a single source file while the mechanism can handle undo involving multiple source files. The mechanism should become a great help of the refactoring process from the viewpoint of the following points.

- Before performing a refactoring, developers cannot estimate the impact of it completely. Therefore they sometimes have to cancel the refactoring performed because of the unexpected result of it. Undo mechanism allows them to readily cancel refactorings performed.

- Many refactorings tend to rewrite multiple source files rather than a single source file. Therefore, undo functions implemented in existing editors or IDEs are not sufficient to cancel refactorings performed.

The possible of easy cancel of refactorings performed should be one factor that developers decide whether they perform refactorings or not.

[3] Our method requires user decisions when constructing revised structure. The detail is described in Section 3.

Several methods for automated refactoring have been proposed [4, 13, 19, 20]. The methods automatically detect parts that refactorings should be applied to. The parts are identified by using software metrics or heuristics based on author's experiences. Refactoring patterns and design patterns are the ways that those method automatically applied refactorings to the parts. The source code change can be done full-automatically, which means that the cost of rewriting files will drastically decrease. However, the automatic transformation may tend to unexpected results rather than manual source code rewriting. We consider that the undo mechanism as described in the above will be a complementary technique to the automatic refactorings.

Weissgerber et al. has proposed a method identifying refactoring performed in the past [21]. The method is a hybrid approach of source code analysis and code clone detection, and in the experiments of the paper it could identify past refactorings with high recall and high precision. Demeyer et al. also has proposed a refactoring identification method [7]. The method identifies past refactorings based on the change of metrics, and identification accuracy is little lower than Weisserber's hybrid approach. We think it important to identify and evaluate refactorings performed in the past. If we could know what kinds of past refactorings were effective in the development or maintenance process of the software system, we can predict that the same kinds of present refactorings would be also effective.

8 Conclusion

In this paper, a method for refactoring effect estimation was proposed. The method measures CK metrics from the original program and the revised one without actually performing the refactoring, and compares the metrics values. The comparison result represents how the complexity of the program will change by perform the refactoring. Also, a tool was developed based on the proposed method and a case study was conducted to evaluate the usefulness of the proposed method.

In the case study, the proposed method recommended a refactoring candidate because of the reduction of metric CBO while the examinee, who is a master student, couldn't select the refactoring candidate although he developed the program. From this example, we conclude that the proposed method is useful for navigating developers/maintainers to perform effective refactorings.

Of course, the proposed method remains several points to be improved, which are described as follows.

- Extend to be able to handle other refactoring patterns like *Inline Class*.
- Consider other cost required to complete refactorings like source code modification and regression test.
- Evaluate other refactoring patterns handled in the proposed method.

Acknowledgement

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology.

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[2] B. W. Boehm, C. Abis, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Streece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 2000.

[3] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.

[4] M. O. Cinne'ide. Automated refactoring to introduce design patterns. In *Proc. of the 22th International Conference on Software Engineering (Doctral Workshop)*, pages 722–724, May 2000.

[5] ckjm. http://www.spinellis.gr/sw/ckjm/.

[6] S. Counsell, R. M. Hierons, R. Najjar, G. Loizou, and Y. Hassoun. The effectiveness of refactoring, based on compatibility testing taxonomy and a dependency graph. In *Proc. of the Testing: Academic and Industrial Conference on Practice and Research Techniques*, pages 181–192, Oct 2006.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactoring via change metrics. In *Proc. of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–177, Oct 2000.

[8] A. V. Deursen and L. Moonen. The video store revisited - thoughts on refactoring and testing. In *Proc. of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76, May 2002.

[9] M. Fowlor. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[10] M. H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[11] B. Henderson-Sellors. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[12] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proc. of International Symposium on Applied Corporate Computing*, pages 78–84, Oct 1995.

[13] S.-U. Jeon, J.-S. Lee, and D.-H. Bae. An automated refactoring approach to design pattern-based program gransformation in java programs. In *Proc of 9th Asia-Pacific Software Engineering Conference*, pages 337–345, Dec 2002.

[14] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. of the 18th IEEE International Conference on Software Maintenance*, pages 576–585, Oct 2002.

[15] C. C. Kit. http://bcel.sourceforge.net/cck.html.

[16] R. Leitch and E. Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proc. of the 9th International Symposium on Software Metrics*, pages 309–322, Sep 2003.

[17] T. Macabe. A complexity measure. *IEEE Transaction on Software Engineering*, 2(4):308–320, Dec 1976.

[18] K. Maruyama. An accurate and convenient undo mechanism for refactorings. In *Proc. of the 13th Asia-Pacific Software Engineering Conference*, pages 309–316, Dec 2006.

[19] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: A metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(4-5):331–361, Jul 2004.

[20] A. Trifu and U. Reupke. Towards automated restructuring of object oriented systems. In *Proc. of the 11th European Conference on Software Maintenance and Reengineering*, pages 39–48, Mar 2007.

[21] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of the 21st IEEE International Conference on Automated Software Engineering*, pages 231–240, Sep 2006.