2014 21st Asia-Pacific Software Engineering Conference

An efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation

Ah-Rim Han
Department of Computer Science
Korea University
Seoul, South Korea
arhan@korea.ac.kr

Doo-Hwan Bae
Department of Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, South Korea
bae@se.kaist.ac.kr

***Abstract*—For automating refactoring identification, previous methods for assessing the impact of a large number of refactoring candidates may be computationally expensive. In our paper, we propose an efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation, which is approximate but fast. This proposed method is evaluated on a refactoring identification approach for jEdit and Columba, two large-scale open source projects. The experiments show that the proposed method requires less time for assessing refactoring candidates and that the refactoring identification approach using our proposed method also improves maintainability.**

Keywords*—*Refactoring effect assessment method, refactoring identification process, maintainability improvement

I. Introduction

Refactoring is a widely studied problem and one that is commonly practiced in industry. In industry, refactoring is largely a manual activity that relies heavily on a software developer's expertise; however, the research community seeks to automate the refactoring process, so that refactoring can be cheaper, more effective, and applied more rigorously.

There have been two lines of studies for automating the refactoring identification process. In refactoring identification using a stepwise selection method [1], [2], the refactoring that best improves maintainability is selected in a stepwise manner among many of the extracted refactoring candidates. The entire process is repeated by re-extracting and re-assessing refactoring candidates. The stepwise approach offers the advantage of taking into account the changing system. Thus, complex dependencies among refactorings need not be considered, and newly created refactoring candidates can be taken into account. However, the stepwise approach can be inefficient by selecting only one refactoring for each iteration, even after assessing a large number of refactoring candidates. Nonetheless, selecting multiple refactorings at a time by simulating the application of possible refactoring candidates that may be available after performing n more iterations is difficult because the impact of a large number of refactoring candidates needs to be assessed. On the other hand, in search-based refactoring [3]–[7], researchers try to find an optimal sequence of refactoring applications using search techniques. During the search process, the fitness function offers some guidance toward improvements in fitness (i.e., maintainability). However, a large number of refactoring candidates needs to be assessed in order to find an optimal sequence of refactoring applications using search techniques. When the system is large, the number of refactoring candidates to be examined would be even larger.

In this context, the previous methods for assessing the effect of refactoring candidates may be computationally expensive. For instance, for calculating a fitness function for each refactoring candidate, each refactoring candidate needs to be actually or virtually applied on a source code/abstracted design model. Then a design quality evaluation model/function should be calculated, which may require calculating metric(s) of all classes existing in a system. For obtaining each metric of a class, the dependencies that entities belonging to a class have with the ones of the class itself (inner entities) or that entities of a given class have with the ones of other classes (outer entities) should be examined.

We therefore argue there is a need for an efficient (faster and cheaper) method for assessing the impact of refactoring candidates on maintainability, even if there is some loss of precision. To identify the refactorings to be applied, the fitness function offers some guidance toward improvements by estimating the effects of refactoring candidates; it does not need to be as accurate as the evaluation for the real effects of the refactorings application, although, in the state of the art approaches, there is no distinction between evaluating the effects of the applied refactorings and assessing (estimating) the impact of the applied refactoring candidates. Recently, Search Based Software Engineering (SBSE) communities has increased the need for new forms of surrogate metrics that retain some of the essence of the more computationally expensive metrics but that sacrifice some degree of precision for computational performance [8]. The methodology accepts that the surrogate can be used to cheaply assess an approximate fitness to guide a search-based approach. Likewise, in refactoring identification using a stepwise selection method, if the assessment method for assessing the impact of refactoring candidates were to be more efficient, it would be easier to select multiple refactorings at a time by simulating the application of refactoring candidates that are available after performing n more (or just more than one) iterations.

In our paper, we propose an efficient method for performing faster impact assessment of the identified refactoring

1530-1362/14 $31.00 © 2014 IEEE
DOI 10.1109/APSEC.2014.69

opportunities, which is the most time consuming part in the process. This is achieved by measuring the maintainability of the examined system using a more simplistic dependencies between methods and attributes in the system, and by performing fast matrix-based computations to assess the impact of each refactoring opportunity. The refactoring effect assessment method is developed in the form of a delta table. Based on matrix computation, the "maintainability variance" (i.e., delta of maintainability) of the application of a refactoring candidate can be calculated at once by only changing link and membership matrices (modeling configuration of the software design) and manipulating those matrices. The matrix computation is fast, because there are various scientific and numerical techniques to accelerate the speed (e.g., we use SciPy[1] libraries implemented for Python).

We applied the refactoring identification approach based on the matrix-computation refactoring effect assessment method (i.e., delta table) to two open-source projects, jEdit[2] and Columba[3]. Experiments reveal that the proposed method is significantly more efficient than an existing method that performs virtually each refactoring opportunity in a low-level design model of the examined system and computes a metric (i.e., the Entity Placement Metric (EPM) [1]) for the entire system after the virtual application of each refactoring.

We use Move Method refactoring to illustrate our approach. Our proposed method of refactoring identification can be extended to big refactorings as well once they are broken down to elementary-level refactorings (e.g., Move Method or Move Field refactorings), and duplications (e.g., conflicts) and the dependencies among them are analyzed. In this paper, we do not consider Move Field refactoring—moving attributes (i.e., fields) from one class to another. We agree with the opinion [1] that fields are strongly conceptually bound to the classes in which they are initially placed and are less likely to change than methods once assigned to a class.

The rest of this paper is organized as follows: Section II contains a discussion of related studies. Section III explains the overall refactoring identification process and the need for an efficient method for assessing the impact of refactoring candidates. Section IV explains the detailed procedure for calculating the delta table, describing the matrix computation-based refactoring effect assessment method. In Section V, we present the experiment to evaluate the proposed approach and discuss the obtained results. Finally, we conclude and discuss future research in Section VI.

II. Related Work

For automated refactoring, several techniques and tools for supporting the activities of the refactoring identification process have been studied and proposed. Our paper is related to the studies of assessing the design quality of the refactored code.

Tahvildari and Kontogiannis [9] propose a metric-based method for detecting design flaws and analyzing the impact of the chosen meta-pattern transformations for improving maintainability. They detect design flaws based on predefined quality design heuristics using object-oriented metrics of complexity, coupling, and cohesion. However, the effects of certain given meta-pattern transformations are evaluated on object-oriented metrics as positive and negative. Since a quantitative method for assessing the effects of meta-pattern transformations is not available, the approach cannot determine the most effective refactorings in terms of the degree of maintainability improvement among the multiple candidates of meta-pattern transformations. Du Bois et al. [10] provide a table representing the analysis of the impact of certain refactorings—that redistribute responsibilities either within the class or between classes—on cohesion and coupling metrics. As the work in [9], they specify the impact of refactorings as ranges of best to worst cases as positive (i.e., improvement), negative (i.e., deterioration), and zero (i.e., neutral); this method also lacks a means of quantitative refactoring-effect assessment, which is essential for making a decision on the most effective refactorings for improving maintainability.

To quantify the impact of the degree of maintainability improvement for assessing refactoring candidates, researchers have used various types of design quality evaluation models/functions, such as the weighted sum of OO metrics [6], QMOOD [5], distance measures [11], EPM [1], and maintainability evaluation function [2]. However, to calculate design quality evaluation models/functions, the refactorings should be actually or virtually applied to the source codes or design models. The *actual* application of the suggested refactorings to source code adds a significant overhead due to disk write operations performed once for applying each refactoring and once for undoing it [1]. Even for the application of suggested refactorings *virtually*, the cost of the assessment for the large number of refactoring candidates—produced for each iteration of the refactoring identification process or that are in the search spaces required to be examined in search-based refactoring—may be computationally expensive. For instance, to know the effect of the application for each refactoring candidate in terms of maintainability improvement, a design quality evaluation model/function should be calculated after the application of the refactoring candidate. This may require calculating metric(s) of all classes existing in a system; and for obtaining each metric of a class, the dependencies between inner entities or outer entities should be examined.

III. Refactoring Identification Process

Refactoring identification refers to planning where to apply which refactorings or how to apply the refactorings for meeting the goals of refactoring, such as improving maintainability, understandability, and testability.

A. Overview

For automating the refactoring identification process, we use the systematic approach provided in our previous work [2] using a stepwise selection method. The approach is composed of three main activities.

Extraction: Refactoring candidates that are expected to improve software design quality (i.e., maintainability) are extracted from an object-oriented source code.

Assessment: The effects of the extracted refactoring candidates can be assessed using various types of evaluation

[1] http://www.scipy.org/

[2] http://www.jedit.org/

[3] http://sourceforge.net/projects/columba/

models for maintainability (e.g., maintainability evaluation function [2]). The effect of a refactoring candidate is assessed as follows: the difference (Δ) of the evaluation models for maintainability = the value of the evaluation model for maintainability *after* applying the refactoring candidate – the value of the evaluation model for maintainability *before* applying the refactoring candidate. This represents the maintainability variance for the refactoring candidate and can be used as a *fitness function* for refactoring selection criteria. When the larger (or smaller) value of the evaluation model for maintainability denotes the design having higher maintainability (i.e., more improvement in maintainability), the design should be converged to have the larger (or smaller) value of the evaluation model for maintainability (by applying a refactoring); therefore, the fitness function is designed to be maximized (or minimized).
Selection: using the results of the assessment above, we select the refactoring that most improves the maintainability among the extracted refactoring candidates.

We formulate the preconditions of refactorings [1], [12] and check them before selecting a refactoring to be applied to assure behavior preservation. The selected refactoring is applied, and extraction and assessment of refactoring candidates are reperformed to select the next refactoring. The refactoring identification process is *iterated* until there are no more improvements in fitness for the extracted refactoring candidates. When no more refactoring candidates for improving maintainability are found, the refactoring identification procedure is terminated and the sequence of logged refactorings is generated.

It is worth noting that the proposed approach does not refactor an object-oriented program in a fully automated manner but automatically identifies a set of refactoring candidates that can be safely applied for delivering the improvement on maintainability. Thus, the software developer must make the final decision on whether or not to apply the suggested refactorings; even though the recommended refactorings are beneficial from a maintainability perspective, they might be rejected due to other factors.

B. Need for an Efficient Refactoring Effect Assessment Method

One of the major technical challenges in automated refactoring identification is determining the sequence of refactorings to perform since each refactoring depends on the preceding applied refactoring. This is because the application of a refactoring changes the system structure and may make many subsequent refactorings inapplicable (or less effective). Likewise, each applied refactoring may affect the applicability of other refactoring candidates or influence their effects on design quality factors, such as maintainability. This phenomenon is known in evolutionary computation as epistasis [13].

In our previous work [2] and that of Tsantalis [1], the refactoring that best improves maintainability is selected in a stepwise way among many of the extracted refactoring candidates; the entire process is then repeated by re-extracting and re-assessing refactoring candidates. To restrict the number of refactoring candidates to be examined for each iteration, refactoring candidates are extracted using object-oriented design heuristics; and they are sorted according to, for example, the scoring function [2] or the number of assesses with the distance measure [1] (e.g., for each method, a target class—which has the largest number of assesses with the smallest distance measure—is determined as a refactoring candidate) to choose part of the refactoring candidates. The stepwise selection method takes into account the changing system, which is an advantage. Thus, complex dependencies among refactorings need not be considered, and newly created refactoring candidates can be taken into account. However, the stepwise selection could be inefficient by selecting only one refactoring for each iteration *even* after assessing a large number of refactoring candidates. Nonetheless, selecting multiple refactorings at a time by simulating the application of possible refactoring candidates that may be available after performing n more iterations is very difficult. As the number of refactoring candidates increases, the number of possible refactoring sequences increases exponentially. Therefore, scheduling refactorings (i.e., selecting multiple refactorings at a time) by investigating all possible refactoring candidates exhaustively may become impossible (NP-hard).

In the refactoring identification using a stepwise selection method, if the assessment method for assessing the impact of refactoring candidates is efficient, it would be easier to select multiple refactorings at a time (in short, it enables to defer the refactoring selection) by simulating the application of refactoring candidates that are available after performing n more (or just more than one) iterations. Likewise, in search-based refactoring [3]–[7], it would be easier to find an optimal sequence of refactoring applications using search techniques, which requires to examine a large number of refactoring candidates.

For the need stated above, we propose an efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation. By only changing link and membership matrices (modeling configuration of the software design) and manipulating those matrices, the maintainability variance for the application of a refactoring candidate on the design configuration can be easily obtained. A more detailed explanation of the refactoring effect assessment method is provided in Section IV.

IV. Matrix Computation-Based Refactoring Effect Assessment Method

Fig. 1 presents an overview of calculating the matrix computation-based refactoring effect assessment method. The refactoring effect assessment method is developed in the form of a delta table, which enables the effects of extracted refactoring candidates to be assessed. In Section IV-A, we describe how maintainability is measured in our paper and explain how the abstracted design model capturing important entities affecting maintainability is obtained from object-oriented source code. In Section IV-B and Section IV-C, the definition and calculation of the refactoring effect delta table are explained.

A. Construction of the Design Model

Our goal for refactoring is to improve maintainability for the design of the object-oriented software. We measure maintainability based on the following concept. In object-oriented software, high cohesion and low coupling have been accepted as important factors for good software design quality in terms of maintenance [14]. *Cohesion* corresponds to the degree to which entities of a class belong together, and

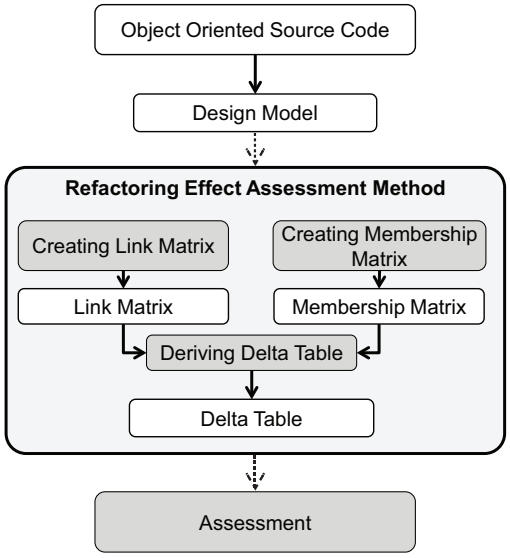
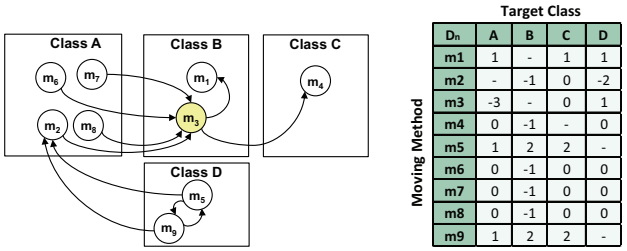


Fig. 1: Overview of calculating the matrix computation-based refactoring effect assessment method (i.e., delta table) and where this method is used in the refactoring identification process for assessing refactoring candidates.



D_n	A	B	C	D
m1	1	-	1	1
m2	-	-1	0	-2
m3	-3	-	0	1
m4	0	-1	-	0
m5	1	2	2	-
m6	0	-1	0	0
m7	0	-1	0	0
m8	0	-1	0	0
m9	1	2	2	-

(a) Example design model n. (b) Delta table D_n.

Fig. 2: Example of the design model and the refactoring effect delta table.

coupling refers to the strength of dependency established by a connection from one class to another. For this reason, the number of dependencies between inner entities should be as large as possible (high cohesion). At the same time, the number of dependencies between outer entities should be as small as possible (low coupling). To this end, maintainability is quantified as the number of dependencies of entities across classes. This number naturally represents the *lack* of degree of dependency among entities of the same class (lack of cohesion) and, at the same time, the degree of dependency among entities of different classes (coupling). As a result, by applying refactorings, we aim to *reduce* this number in order to improve maintainability. In other words, the fitness function is designed to be minimized by *reducing* this number for improving maintainability.

From the object-oriented source code, the design model captures important entities and their dependencies affecting maintainability and is in the form of a graph. The initial design model G_R = (V_R, E_R) is defined as follows:

- V_R = {methods, attributes}
- E_R = {method_calls(method m_1, method m_2), attribute_accesses$_1$(method m_1, attribute a_1), attribute_accesses$_2$(method m_1, method m_2)}.

The vertices (V_R) indicate the entities of methods and attributes. The vertex contains membership information indicating that an entity belongs to which class. The edges (E_R) indicate the dependency between entities. We assume that a dependency exists between two entities when these entities are preferably located in the same class in order to improve maintainability (in terms of low coupling and high cohesion). To this end, an edge is connected between the entities when (1) a method calls the other method (method_calls), (2) a method accesses an attribute (attribute_accesses$_1$), or (3) two methods access the same attribute (attribute_accesses$_2$).

In this paper, we focus on relations established by method-calling procedures when capturing dependencies among entities in the design model. We do not consider the methods of getter/setter, since they do not provide functionality. Note that for dependencies that may affect maintainability, other types of dependencies caused by structural relations between classes (such as association, aggregation, composition, and inheritance) can be considered.

B. Definition of the Refactoring Effect Delta Table

A refactoring effect delta table is derived to quantify the degree of maintainability improvement after the application for each elementary refactoring candidate. In this table, the row elements indicate the moving methods and attributes while the column elements indicate the target classes. Each cell in the refactoring effect delta table indicates a Move Method refactoring (row: moving method, column: target class). The value of each cell of the table indicates maintainability improvement (i.e., delta of maintainability) on the current design model, which is obtained after the application of each Move Method refactoring.

For example, in design model n of Fig. 2(a), the system consists of four classes and each class contains methods A = {m_2, m_6, m_7 m_8}, B = {m_1, m_3}, C = {m_4}, and D = {m_5 m_9}. The dependencies are represented with directed edges. For this design model, the refactoring effect delta table can be obtained as in Fig. 2(b). Let MM(method m, class c) denote each cell of the Move Method refactoring—moving method m to target class c—and let D_n[MM(method m, class c)] denote the delta of maintainability of the refactoring (i.e., moving method m to class c) for design model n. Following this, by using these notations, the value for delta of maintainability for the refactoring, for example, MM[(m_3, A)], referring to the refactoring effect delta table is as follows: D_n[MM(m_3, A)] = -3. This means the refactoring carried out in moving method m_3 (located in class B) to target class A reduces the systems dependencies (i.e., improves maintainability) by as much as -3. This change in the value of the systems maintainability is calculated by adding the number of decreasing dependencies (-4) and the number of potentially increasing dependencies (+1) across class A and class B. The decreasing dependencies are (m_7, m_3), (m_6, m_3), (m_8, m_3), and (m_2, m_3), whereas

the increasing dependency is (m_3, m_1). The algorithm for calculating the delta table is explained in Section IV-C.

C. Calculation of the Refactoring Effect Delta Table

The delta table is calculated as follows. First, the design model is mapped to the Link matrix (L) and the Membership matrix (M). The links and memberships can be directly mapped from the edges and vertices of the design model G_R. The L denotes the link information where an entity (row) has a dependency (a connection) to an entity (column). Let the cell of this link be L(row entity, column entity). The cell value of L denotes the strength of the relation. When there is a dependency from an entity to an entity, then the value added to the cell of L is 1; otherwise, when there is no dependency between two entities, the cell of L is 0. Note that in L, the direction of the edges (dependencies) is not distinguished and the strength of the edges (dependencies) is considered. When two entities, entity a and entity b, have a dependency, the values of cells $L(a, b)$ and $L(b, a)$ both become 1. In short, since the direction of a dependency (edge) is not differentiated, L is symmetric. If those two entities have dependencies with each other, then the values of cells $L(a, b)$ and $L(b, a)$ both become 2. Since there is no dependency between fields, the cell of L is 0. M denotes the membership information where an entity (row) belongs to a particular class (column). The cell of M is 1 when an entity (row) is placed in a class (column); the cell of M is 0 when the entity is not located in the class. Note that even though we do not consider Move Field refactorings, the entities of attributes need to be considered in the delta table. The membership and link information related to attributes—e.g., where an attribute belongs to a particular class (membership information) and a method assesses an attribute or two methods assess the same attribute (link information)—affects the calculation of the delta of maintainability.

The delta table is calculated as follows. The projection matrix (P) is produced by multiplying the two matrices L and M. P represents the link information from an entity (row) to a class (column). We compute two types of P as follows.

$$P_{Int} = L_{Int} \times M, \quad P_{Ext} = L_{Ext} \times M.$$

P_{Int} (internal projection matrix) and P_{Ext} (external projection matrix), each of which is computed by the multiplication of M with L_{Int} (matrix denoting internal links) and L_{Ext} (matrix denoting external links), respectively. L_{Int} represents the internal links that are associated between entities in the same class, while L_{Ext} represents the external links that are associated between entities across classes.

By using the formulation below, we derive the delta table (D) in which each cell is the delta of the maintainability value after the application of each Move Method refactoring on the design.

$$D = Inv(P_{Int}) - P_{Ext}.$$

The cell of P_{Int} is $k \geq 1$ when the internal link exists from the entity (row) to the class itself (column). This means that moving the entity to other classes (other than the class itself) will *potentially* increase the external link(s) in the system. We use the $Inv()$ function for P_{Int} because, as stated above, moving an entity to other class will increase the external link(s). The $Inv()$ function inverts the cell of P_{Int}(entity, class itself) as $k \rightarrow 0$ and P_{Int}(entity, other classes) as $0 \rightarrow k$. On the other hand, the cell having $k \geq 1$ in P_{Ext} means that the external link exists from an entity (row) to a class (column). Moving the entity to the class will decrease the external link(s) in the system.

By following the procedure explained above, Fig. 3 illustrates how to obtain the delta table (Fig. 2(b)) for the corresponding design (Fig. 2(a)).

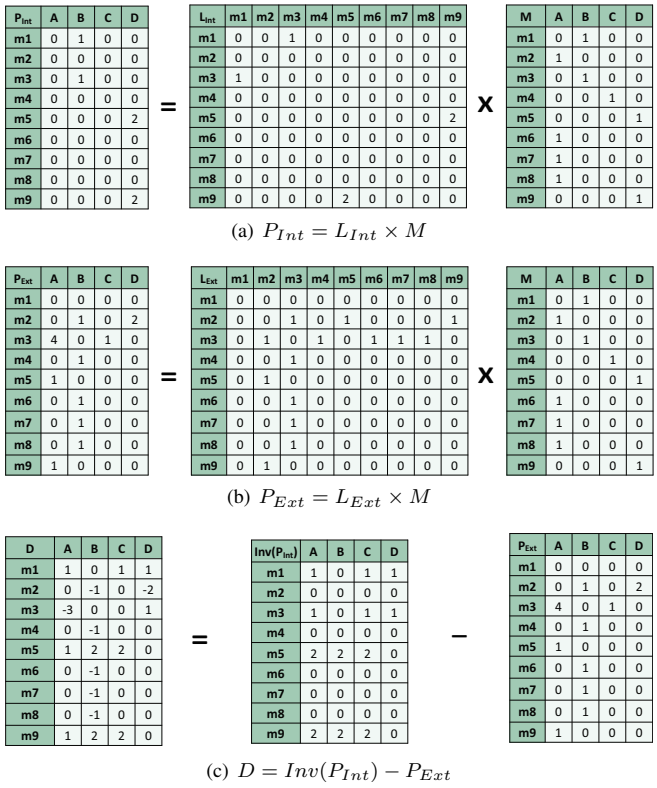


Fig. 3: Example of calculating the delta table (D) of Fig. 2(b) for the design of Fig. 2(a).

V. Evaluation

We framed the following research questions for our experiment.

RQ1. By how much is the matrix computation-based refactoring effect assessment method efficient for assessing the impact of refactoring candidates?

RQ2. Does the refactoring identification approach based on our method help improve maintainability?

Two projects are chosen as experimental subjects: jEdit and Columba. They contain a relatively large number of classes and have been widely used as experimental subjects. Table I summarizes characteristics of each subject.

A. Experimental Design

For automating refactoring identification, we use the approach of selecting one refactoring for each iteration, as

TABLE I: Characteristics for each subject.

Name (Version)	jEdit (jEdit-4.3)	Columba (Columba-1.4)
Type	Text editor	Email client
Class ♯	952	1506
Method ♯	6487	8745
Attribute ♯	3523	3967

explained in Section III-A. We compare 1) the approach based on the delta table and 2) the same approach substituting the delta table with the EPM [1], which is a method for assessing the impact of refactoring candidates on maintainability based on distance measures.

For each iteration in the approach of refactoring identification based on the delta table, the assessed refactoring candidates are all the Move Method refactorings available in the system. In the approach of refactoring identification based on the EPM, the number of refactoring candidates to be examined are restricted. At the first level, for each pair of classes (source class and target class existing in a system), the top 10 refactoring candidates highly ranked with the number of entities that method m (belonging to a source class) accesses from each target class are chosen. Then, at the second level, all the chosen refactoring candidates are sorted in ascending order according to the distance measure [1] and the top 50 refactoring candidates are assessed for each iteration. Under the computing resources used in our experiment, we could not accept all the extracted Move Method refactoring candidates for all methods in a system. For each iteration, we select the Move Method refactoring that most improves maintainability. Therefore, among refactoring candidates, the refactoring that most reduces the dependencies in terms of the value in the delta table and that has the lowest value for the EPM is selected. The *lowest* EPM should be selected because it uses distances instead of similarities in its computation. The refactorings are identified by repeating the refactoring identification process until no more refactorings that improve maintainability (i.e., no improvement in the values of the delta table or the EPM) are found.

To investigate that by how much the matrix computation-based refactoring effect assessment method is efficient for assessing the impact of refactoring candidates (RQ1), we compare the two approaches on the elapsed time required for performing each iteration and the total elapsed time. The elapsed time for each iteration includes extracting refactoring candidates, checking preconditions, updating the design model, and recalculating the refactoring effect delta table, as well as assessing the refactoring candidates. The elapsed time (sec) is measured under the following conditions: processor 2.7GhHz Intel Core i5, Memory 8G 1607 MHz DDR3, and Software OS X 10.9.3. The refactoring identification process is repeated until it reaches the final solution (where no more refactorings that improve maintainability are found), and the total elapsed time is obtained by accumulating the elapsed time of entire iterations.

To investigate that the refactoring identification approach based on our method help improve maintainability (RQ2),

TABLE II: Required time for performing the refactoring identification approach (explained in Section III-A) based on the delta table and the EPM [1].

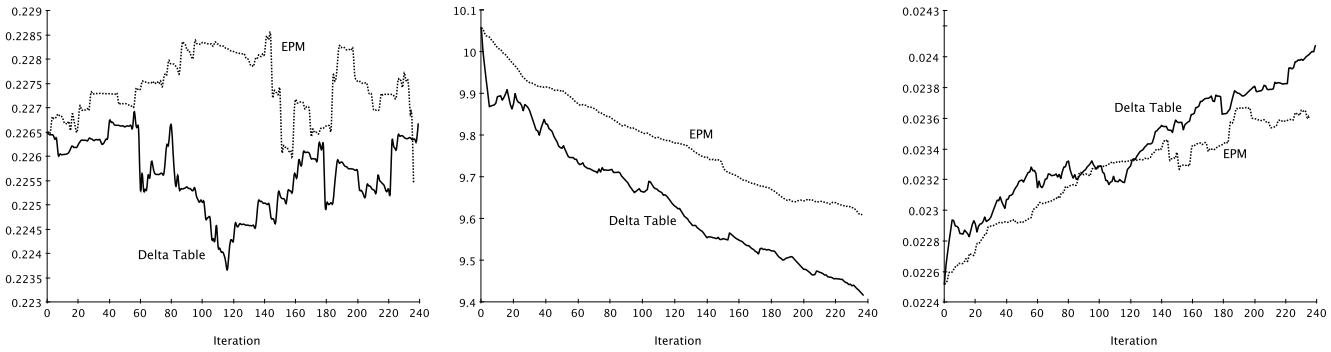
Time (sec)	jEdit (Total: 236 iterations)		Columba (Total: 90 iterations)	
	Delta Table	EPM	Delta Table	EPM
Avg. time per iteration	1.66	317.99	2.33	602.69
Max. time per iteration	5.95	346.11	5.18	665.59
Min. time per iteration	1.49	315.21	2.16	596.81
Total time	391.81	75,046.46	209.29	54,241.87

we show that the refactored design that applies refactorings identified using our approach contributes to improving the maintainability of the system. The *maintainability* of the refactored design of the code is evaluated using several maintainability indices, such as Method Similarity Cohesion (MSC) [15] (cohesion metric), Message Passing Coupling (MPC) [16] (coupling metric), and the maintainability evaluation function [2] (which was used to assess the contribution to improving maintainability of extracted refactoring candidates in our previous paper [2]). In MSC, the similarity among all pairs of methods is integrated and normalized to measure the cohesiveness of the class. MSC is different from other cohesion metrics in that it considers the degree of similarity between a pair of methods in a class. MPC is a commonly used metric for representing coupling and is appropriate for capturing small code changes, such as moving a method to a class. MPC for a class C indicates the number of static method calls for all invoked imported methods. The maintainability evaluation function is designed as $\frac{cohesion}{coupling}$ to produce larger values as the system becomes more maintainable (with higher cohesion and lower coupling).

B. Results

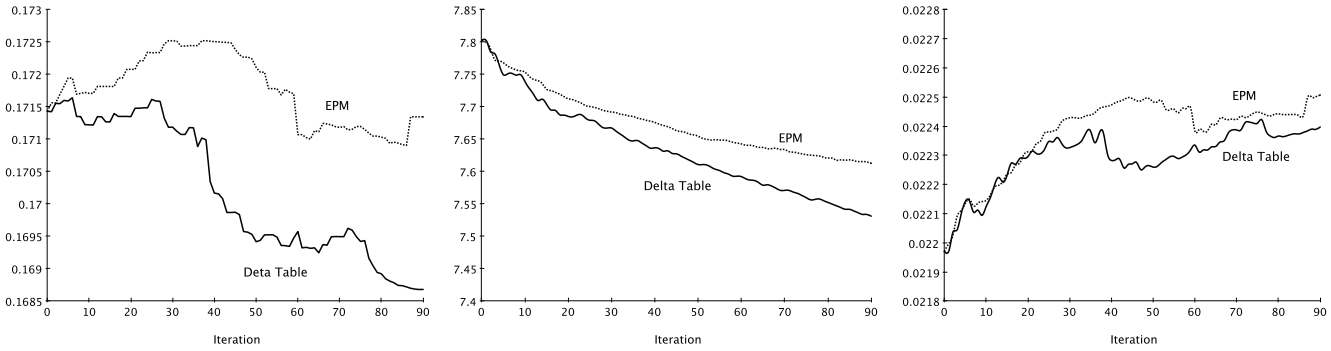
Table II represents the elapsed time of each iteration and the total elapsed time for performing the refactoring identification approach for jEdit and Columba, respectively. The baseline is set to the approach that is terminated within smaller iterations. The number of the total iterations for jEdit and for Columba are 236 iterations and 90 iterations, respectively. From this table, we can observe the following:

- The total time required to perform the approach with the delta table is much less than the approach with the EPM.
- In the approach with the delta table, the maximum time per iteration is the time taken for performing the first iteration. In short, constructing the design model and calculating the link and membership matrices need more time compared to the rest of the iterations (e.g., jEdit: max. 5.95 [sec] $>$ avg. 1.66 [sec], Columba: max. 5.18 [sec] $>$ avg. 2.33 [sec]).
- As the systems become larger (jEdit: 952 classes, Columba: 1506 classes), the number of refactoring candidates that need to be assessed for each iteration is increased; therefore, the computation time is increased as well. For instance, the average time required for each iteration is increased from jEdit to Columba as



(a) Method Similarity Cohesion (MSC) [15] (b) Message Passing Coupling (MPC) [16] (c) Maintainability evaluation function [2]

Fig. 4: Maintainability improvement for jEdit.



(a) Method Similarity Cohesion (MSC) [15] (b) Message Passing Coupling (MPC) [16] (c) Maintainability evaluation function [2]

Fig. 5: Maintainability improvement for Columba.

follows: avg. 1.66 [sec] → avg. 2.33 [sec] in the approach with the delta table while avg. 317.99 [sec] → avg. 602.69 [sec] in the approach with the EPM.

- It is also notable that the rate of increased computation time with respect to the system size is much less in the approach with the delta table than the approach with the EPM. For instance, for computing *90 iterations*, the total time is increased from jEdit to Columba as follows: 154.63 [sec] → 209.29 [sec] for the approach with the delta table while 28,622 [sec] → 52,241 [sec] for the approach with the EPM. Note that the accumulated time to perform the same total number of iterations should be compared, thus the time for 90 iterations (smaller total number iterations of Columba) is chosen, even in jEdit the process is iterated 236 times in total time of 391.81 [sec] (delta table) and 75,046 [sec] (EPM).

The graphs of maintainability improvement are presented in Fig. 4 and Fig. 5 for jEdit and Columba, respectively. The refactoring selected for each iteration is applied, and the maintainability of the refactored design is shown using maintainability indices. For the approach with the delta table in both jEdit and Columba, the values of the maintainability evaluation function increase (in Fig. 4(c) and Fig. 5(c)) as the selected refactorings are applied.

The trend in some maintainability indices is not monotonous (especially for MSC), because the methods of evaluating for the real effects of the applied refactorings and assessing (estimating) the impact of refactoring candidates are different. In other words, the former method represents the maintainability indices (i.e., MSC, MPC, and the maintainability evaluation function), while the latter method indicates the values of maintainability in the delta table. Note that our method is motivated by the observation that the method for estimating the impact of refactoring opportunities does not need to be as accurate as the evaluation method for the real effects of the applied refactorings.

It is also noteworthy that the maintainability evaluation function can be increased, even though MPC (coupling metric) or MSC (cohesion metric) is deteriorated. This happens when the decreasing degree of MPC (or the increasing degree of MSC) is greater than the decreasing degree of MSC (or the increasing degree of MPC). In such case, the positive impact outperforms the negative impact on improving maintainability. Generally, the cohesion metric should be increased while the coupling metric should be decreased in order to improve maintainability.

For jEdit, the improvement of MPC in the approach with the delta table surpasses the improvement of MSC in the approach with the EPM. The values of maintainability in the delta table are measured based on the number of dependencies

across the classes, thus they are similar with the MPC. As a result, in Fig. 4(c), the degree of improvement for the maintainability evaluation function in the approach with the delta table is larger than the one in the approach with the EPM. To investigate the validity of the provided method, more experiments need to be performed on various projects with more conditions (e.g., scalability tests, correlation analysis with the existing metrics). We plan to perform more extensive experiments in future work.

It is important to emphasize that, with the experiment for RQ2, we aim to show that the refactoring identification approach using our method improves maintainability. The goal of our work is *not* to enhance the capability of improving maintainability of the refactoring identification approach. In other words, we do not compete with the approaches of identifying refactoring opportunities in the perspective of the capability for maintainability improvement.

From the results above, we conclude that our refactoring effect assessment method (i.e., delta table) is more efficient than the EPM for estimating the impact of refactoring candidates; and the refactoring identification approach using our method also contributes to identify refactoring candidates that improve maintainability.

C. Threats to Validity

The internal restrictions or optimization techniques used for measuring the EPM might not be fully considered. Therefore, the results for evaluating maintainability shown in Fig. 4 and Fig. 5 might be affected. Furthermore, in the approach of refactoring identification based on the EPM, more than 50 refactoring candidates need to be assessed for each iteration.

When measuring elapsed time, other factors may affect time. We regard this as causing a small variation, thus we use the elapsed time for the comparison.

VI. Conclusion and Future Work

In our paper, we developed a matrix computation-based refactoring effect assessment method in the form of a delta table and used this method for assessing the impact of refactoring candidates for an automated refactoring identification approach. Compared to the fitness functions used in previous studies, the maintainability variance for the application of a refactoring candidate on the design configuration can be easily obtained at once by only changing link and membership matrices and manipulating those matrices. The refactoring effect assessment method approximately measures the maintainability of the software design using a more simplistic dependencies between methods and attributes in the system, but calculates the values of fitness functions for refactoring candidates faster (based on matrix computation), which provides efficient computation for assessing the impact of a large number of refactoring candidates. In the experiments, we showed that the proposed method requires less time for assessing refactoring candidates and the refactoring identification approach using our method improves maintainability as well.

For future work, we plan to perform more extensive experiments to show the value of the fast refactoring effect assessment method. Specifically, we plan to perform a trade-off analysis between precision and speed (computation time), and their relationship in reaching the solution. For analyzing the capability of assessing a large number of refactoring candidates, we also plan to perform scalability tests.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2013R1A6A3A01062920). This research was supported by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (NIPA-2014-H0301-14-1023) supervised by the NIPA(National IT Industry Promotion Agency).

References

[1] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 347 – 367, 2009.

[2] A.-R. Han and D.-H. Bae, "Dynamic profiling-based approach to identifying cost-effective refactorings," *Information and Software Technology*, vol. 55, no. 6, pp. 966–985, 2013.

[3] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring for software maintenance," *The Journal of Systems & Software*, vol. 81, no. 4, pp. 502–516, 2008.

[4] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 220–235, 2012.

[5] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent GA," *Softw., Pract. Exper.*, vol. 41, no. 5, pp. 521–550, 2011.

[6] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, p. 1916, 2006.

[7] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of prioritised code clone refactoring," *IET Software*, vol. 7, no. 3, 2013.

[8] M. Harman, J. Clark, and M. Cinnédie, "Dynamic adaptive search based software engineering needs fast approximate metrics," in *Proceedings of the 4th International Workshop on Emerging Trends in Software Metrics, San Francisco, USA*, 2013.

[9] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," *Proc. European Conf. Software Maintenance and Reeng*, pp. 183–192, 2003.

[10] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 144–151.

[11] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*. IEEE, 2001, pp. 30–38.

[12] P. V. Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent uml refactorings," *Lecture Notes in Computer Science*, pp. 144–158, 2003.

[13] A. E. Eiben, P.-E. Raué, and Z. Ruttkay, "Solving constraint satisfaction problems using genetic algorithms," in *International Conference on Evolutionary Computation*, 1994, pp. 542–547.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[15] C. Bonja and E. Kidanmariam, "Metrics for class cohesion and similarity between methods," *Proceedings of the 44th annual Southeast regional conference*, pp. 91–95, 2006.

[16] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.