Two-Phase Assessment Approach to Improve the Efficiency of Refactoring Identification

Ah-Rim Han and Sungdeok Cha, *Member, IEEE*

Abstract—To automate the refactoring identification process, a large number of candidates need to be compared. Such an overhead can make the refactoring approach impractical if the software size is large and the computational load of a fitness function is substantial. In this paper, we propose a two-phase assessment approach to improving the efficiency of the process. For each iteration of the refactoring process, refactoring candidates are preliminarily assessed using a lightweight, fast delta assessment method called the Delta Table. Using multiple Delta Tables, candidates to be evaluated with a fitness function are selected. A refactoring can be selected either interactively by the developer or automatically by choosing the best refactoring, and the refactorings are applied one after another in a stepwise fashion. The Delta Table is the key concept enabling a two-phase assessment approach because of its ability to quickly calculate the varying amounts of maintainability provided by each refactoring candidate. Our approach has been evaluated for three large-scale open-source projects. The results convincingly show that the proposed approach is efficient because it saves a considerable time while still achieving the same amount of fitness improvement as the approach examining all possible candidates.

Index Terms—Refactoring assessment, refactoring identification, maintainability improvement

1 Introduction

Software refactoring is widely used in the industry, but still, it has been a manual task that varies greatly depending on the expertise of software engineers. To address this problem, the research community has made efforts to automate the process.

Many refactoring identification studies attempt to automatically suggest the refactoring candidates that can be safely applied for delivering improvement on maintainability; thus, developers can make final decisions on which refactorings are to be applied based on the suggested refactoring candidates. Refactoring opportunities can be identified by applying design patterns [1], [2], removing code clones [3], [4], [5], [6], [7], [8], and resolving bad smells [9] (e.g., Feature Envy [10], [11] and State Checking [12]); alternatively, when there is no information regarding which refactoring candidates may improve maintainability, we may consider all possible moves of methods to classes existing in a system.

The effects of the application for the refactoring opportunities are assessed using a fitness function chosen as the guideline for maintainability improvement. Various types of maintainability evaluation metrics or functions—for example, a coupling metric called Message Passing Coupling (MPC) [13], a cohesion metric called Connectivity [14], or the Entity Placement metric (EPM) [10]—can be used as fitness functions. However, evaluating the effects of numerous trials for the application of refactoring candidates using a fitness function incurs expensive computational costs, and these costs grow exponentially as the system becomes larger. In fact, finding the optimal sequence of refactorings by exhaustively investigating all possible sequences is a known NP-hard problem [5].

Even when locally searching refactoring candidates and selecting refactorings one after another in a stepwise manner for each iteration of the refactoring identification process, as done in our approach, the accumulated number of assessed candidates will increase drastically as the system becomes larger. For instance, in our experiment, the required time for assessing a Move Method refactoring candidate for the fitness function of Connectivity in `Apache Ant` is 0.07 seconds, which seems small and manageable. However, an average number of 1,948 candidates must be assessed for each iteration, and it takes 4.15 hours (14,930 seconds) in total after iterating 100 runs.

The complexity of fitness functions for evaluating the maintainability of a system's design is another factor to consider. A chosen fitness function may require high computational cost, for example, if it is a weighted sum of several metrics. Moreover, in the search-based software engineering (SBSE) community, the recent research trend is to solve optimization problems involving multiple conflicting objectives [15], [16], [17], [18], [19], [20], [21]. Such assessment requires an increased computational overhead; thus, the refactoring identification process needs to be efficient and scalable.

In this paper, we propose a two-phase assessment approach for improving the efficiency of computation in the refactoring identification process. In the first phase, the effects of refactoring candidates for all possible moves of

• The authors are with the Department of Computer Science and Engineering, Korea University, Anam-dong Sungbuk-gu, Seoul 02841, South Korea. E-mail: {arhan, scha}@korea.ac.kr.

*Manuscript received 13 Jan. 2016; revised 30 June 2017; accepted 19 July 2017. Date of publication 24 July 2017; date of current version 23 Oct. 2018. (Corresponding author: Ah-Rim Han.)
Recommended for acceptance by S. Kim.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2731853*

entities (i.e., methods and fields) to other classes are pre-evaluated using various types of Delta Tables. Each Delta Table is constructed of one type of dependency relation (e.g., method calls of entities and shared variables between methods) to fit the particular goal of a fitness function (e.g., MPC, EPM, and Connectivity metrics). Various types of Delta Tables make up the Multi-criteria Delta Table that can take multiple dependency relations into account. Then, the search space of refactoring candidates is reduced by selecting the candidates that are more likely to improve maintainability and highly ranked in the Multi-criteria Delta Table. In the second phase, only the chosen candidates are evaluated using a fitness function. This process is iterated to select the next refactorings, and the refactorings are applied one after another in a stepwise fashion.

The Delta Table is a core concept in the two-phase assessment approach because of its ability to rapidly calculating the values for the delta of maintainability. An element in each Delta Table represents a candidate for Move Method refactoring or Move Field refactoring and its value denotes the changed amount (Δ) of maintainability, which in turn represents the number of links across the classes after moving an entity to a target classes. The Delta Table can be quickly calculated at once by mapping the software design into membership and link matrices and multiplying those matrices. The Delta Table is computed very quickly because various scientific and numerical techniques, such as the special libraries of Eigen [22], help to accelerate the speed of the matrix computation. The benefit of using the Delta Tables exceeds the overhead of computing them, as will be shown in the experiment. Consequently, the elements in the Delta Table are approximate measures, but they are sufficient to preliminarily assess the effects of the application of refactoring candidates in an extremely short time. The Delta Tables help identify the refactoring candidates that are more likely to have better values in a fitness function and, indeed, to improve maintainability; thus, they can be used to reduce the number of refactoring candidates to be evaluated with the fitness function, in order to increase computational efficiency.

We applied our approach to three large (e.g., between 135K and 222K lines of code) open-source projects, namely `Apache Ant` [23], `JGit` [24], and `JHotDraw` [25]. The experiments revealed that our approach is efficient in that it saves a considerable time while still achieving the same amount of fitness improvement as the approach examining all possible candidates (i.e., the no-reduction approach). This time savings increase as the candidate size becomes larger and more complex fitness functions are used. Furthermore, we observed that as the restriction size of refactoring candidates in the Multi-criteria Delta Table is reduced, denoting that the number of candidates to be assessed using fitness functions has decreased, the efficiency of the rate of improvement on fitness functions with respect to time is increased. However, a too-small size can cause the early process termination; thus, it is important to determine the proper size. Finally, with respect to the performance of the Multi-criteria Delta Table as a filter for the restriction, our approach finds refactoring candidates that improve the fitness functions with a high probability.

This paper is organized as follows. Section 2 explains the definition and features of the Delta Tables. Section 3 discusses the refactoring identification process used in our paper, and Section 4 explains the two-phase assessment approach in detail. In Section 5, we present the experiment conducted to evaluate the proposed approach and consider the obtained results. Section 6 contains a discussion of related studies. Finally, we conclude and discuss future research in Section 7.

2 Delta Table 2.0: A Lightweight Assessment Method

We first explain the basic definition of the Delta Table which is a preliminary version that has been presented in previous papers [26], [27]. We formalize the equation and provide the explanation of the meaning of each element constituting it. We then present version 2.0 of the changed Delta Table that reflects new features to the following three aspects: 1) Multi-criteria Delta Table, 2) adaptive Delta Tables for extending to other types of refactorings, and 3) performance improvement. The efficiency of the assessment method is the main concern of this paper; thus, the rapid calculation of the Delta Table is extremely important, and we put extensive effort into performance improvement.

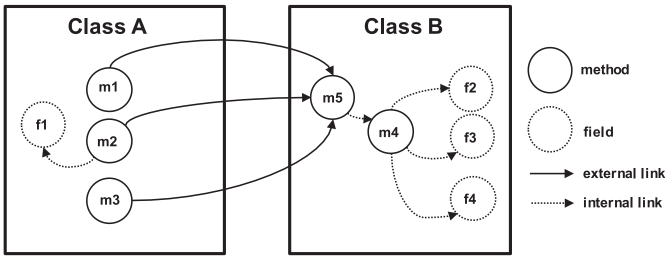
2.1 Basic Definition of the Delta Table

When two entities, such as methods and fields, have dependency relations, they are connected using a link and considered to be located within the same class for higher cohesive and less coupled arrangement of those entities. Counting the number of dependencies between classes is sufficient to estimate the code quality of structural properness. In this context, the *number of links across the classes* naturally represents the *lack* of degree of dependency among entities in the same class (lack of cohesion), and at the same time, the degree of dependency among entities of different classes (coupling). The number of links across the classes that each element of the Delta Table has should be minimized, and this number can be used as an approximate measure for assessing the maintainability of the system's design.

By parsing the object-oriented source codes, all entities (i.e., methods and fields) and classes existing in a system are captured, and the membership and link matrices are constructed. By calculating these matrices, the Delta Table matrix is obtained for all possible moving methods and fields.

Each element in the Delta Table ($\mathbf{D}$) represents a Move Method refactoring or a Move Field refactoring where the entity of a method/field (row) moves to the target class (column). The value of each element $\mathbf{D}_{ij}$ indicates the variance (Δ) of the number of links across the classes when moving method i or field i to class j. The size of the Delta Table is determined by $|E_S| \times |C_S|$, where E_S and C_S denote all entities and classes in the system, respectively. The Delta Table contains all possible moves of methods and fields in the system, and the value of each element can be used to determine the refactorings that result in greater reduction of the number of links across the classes.

An element in a membership matrix ($\mathbf{M}$) represents that an entity (row) belongs to a particular class (column). Element $\mathbf{M}_{ij}$ is 1 when entity i is placed in class j, and all the elements for entity i become 0.



(a) Example of a design model.

L_{Int}	m1	m2	m3	m4	m5	f1	f2	f3	f4
m1	0	0	0	0	0	0	0	0	0
m2	0	0	0	0	0	1	0	0	0
m3	0	0	0	0	0	0	0	0	0
m4	0	0	0	0	1	0	1	1	1
m5	0	0	0	1	0	0	0	0	0
f1	0	1	0	0	0	0	0	0	0
f2	0	0	0	1	0	0	0	0	0
f3	0	0	0	1	0	0	0	0	0
f4	0	0	0	1	0	0	0	0	0

(b) Internal Link matrix ($\mathbf{L_{Int}}$)

L_{Ext}	m1	m2	m3	m4	m5	f1	f2	f3	f4
m1	0	0	0	0	1	0	0	0	0
m2	0	0	0	0	1	0	0	0	0
m3	0	0	0	0	1	0	0	0	0
m4	0	0	0	0	0	0	0	0	0
m5	1	1	1	0	0	0	0	0	0
f1	0	0	0	0	0	0	0	0	0
f2	0	0	0	0	0	0	0	0	0
f3	0	0	0	0	0	0	0	0	0
f4	0	0	0	0	0	0	0	0	0

(c) External Link matrix ($\mathbf{L_{Ext}}$)

M	class A	class B
m1	1	0
m2	1	0
m3	1	0
m4	0	1
m5	0	1
f1	1	0
f2	0	1
f3	0	1
f4	0	1

(d) Membership matrix ($\mathbf{M}$)

Fig. 1. Modeling a design model using matrices.

An element in a link matrix ($\mathbf{L}$) denotes that an entity (row) has a dependency relation to an entity (column). Examples of dependency relations are method calls and shared fields between methods. Element $\mathbf{L}_{ij}$ is 1 when entity i has a dependency relation to entity j. The direction of the links is not distinguished, and the link matrix is symmetric. When two entities have relations to each other in a bi-directional way, then the both elements become 2.

By the multiplication of the link matrix and the membership matrix ($\mathbf{L} \times \mathbf{M}$), we can obtain the number of internal or external links that an entity has with entities located in its class or other classes ($\mathbf{L_{Int}} \times \mathbf{M}$ or $\mathbf{L_{Ext}} \times \mathbf{M}$), respectively. When entity i has dependency relations with the entities locating in class j, the number of dependency relations can be calculated as follows:

$$(\mathbf{L} \times \mathbf{M})_{ij} = \sum_{k=1}^{|E_S|} \mathbf{L}_{ik}\mathbf{M}_{kj}.$$

The inverse function, $\mathbf{Inv}$, is used to invert the values of $\mathbf{L_{Int}} \times \mathbf{M}$. Each element in $\mathbf{L_{Int}} \times \mathbf{M}$ indicates the number of internal links where an entity has dependency relations with entities located in the class itself. This means that moving the entity to other classes will potentially increase the external links in the system. Thus, to obtain the effects after moving entity i with internal link(s), let $(\mathbf{L_{Int}} \times \mathbf{M})_{ij} = n$ ($\neq 0$); the inverse function changes the elements as follows: $(\mathbf{L_{Int}} \times \mathbf{M})_{ij} \leftarrow 0$ and $(\mathbf{L_{Int}} \times \mathbf{M})_{ik} \leftarrow n$, where $k \neq j \wedge$ for all $k \in \{1, \ldots, |C_S|\}$. The element in $\mathbf{L_{Ext}} \times \mathbf{M}$ indicates the number of external links where an entity has dependency relations with entities located in other classes; thus, moving the entity to those classes reduces the number of links across the classes.

Finally, each element of the number of links across the classes in the Delta Table ($\mathbf{D}$) is calculated as given in

$$\mathbf{D} = \mathbf{Inv}(\mathbf{L_{Int}} \times \mathbf{M}) - \mathbf{L_{Ext}} \times \mathbf{M}. \quad (1)$$

The illustrative example for modeling system's design into link and membership matrices is shown in Fig. 1. The system's design in Fig. 1a is mapped to the membership matrix of Fig. 1d, the internal link matrix of Fig. 1b, and the external link matrix of Fig. 1c. In the membership matrix, the entities and classes existing in the system are listed in rows and columns. Four entities and five entities are located in class A and class B. In the internal and external matrices, five internal links and three external links are present, and those matrices are symmetric. Fig. 2 shows the calculation process using the modeling matrices. By multiplying the link matrix and the membership matrix, Figs. 2a and 2b are obtained for the internal and external types, respectively. To reflect the effects when moving internal links to other classes, the inverse function is applied to Fig. 2a, and Fig. 2c is obtained. Moving entities to the classes having external links reduces the number of links across the classes in the system; thus, the Delta Table in Fig. 2d is calculated by subtracting Fig. 2b from Fig. 2c.

To put it briefly, in the Delta Table, an estimate of maintainability variance that can be expected in the application

L_{Int} X M	class A	class B
m1	0	0
m2	1	0
m3	0	0
m4	0	4
m5	0	1
f1	1	0
f2	0	1
f3	0	1
f4	0	1

(a) $\mathbf{L_{Int}} \times \mathbf{M}$

L_{Ext} X M	class A	class B
m1	0	1
m2	0	1
m3	0	1
m4	0	0
m5	3	0
f1	0	0
f2	0	0
f3	0	0
f4	0	0

(b) $\mathbf{L_{Ext}} \times \mathbf{M}$

Inv(L_{Int} X M)	class A	class B
m1	0	0
m2	0	1
m3	0	0
m4	4	0
m5	1	0
f1	0	1
f2	1	0
f3	1	0
f4	1	0

(c) *Inverse function* to 2(a)

D	class A	class B
m1	0	-1
m2	0	0
m3	0	-1
m4	4	0
m5	-2	0
f1	0	1
f2	1	0
f3	1	0
f4	1	0

(d) Delta Table ($\mathbf{D}$) = 2(c) - 2(b)

Fig. 2. Delta table calculation.

of a refactoring candidate, is calculated as $|i|$ - $|e|$, where $|i|$ and $|e|$ indicate the number of internal (e.g., calls to methods defined in the same class) and external relations (e.g., calls to methods defined in the target class), respectively. To this end, a refactoring candidate with a smaller and negative value in the Delta Table is a more promising candidate when it comes to improving maintainability. Candidates with zero or positive values are regarded as the refactoring candidates that *no longer* improve maintainability.

It is important to emphasize that by multiplying the link matrix and the membership matrix, we can obtain information about the classes to which the entities associated with each entity belong, which is the fundamental basis for estimating the potential effects of moving methods with the number of links across the classes. Thus, to simulate the application for one step of a refactoring, only the membership information the class to which an entity belongs needs to be changed; we can then easily obtain the number of the internal and external links by the matrix multiplication very quickly.

2.2 New Features

2.2.1 Multi-Criteria Delta Table

We devise the Multi-criteria Delta Table to adopt various types of Delta Tables. Given a specific fitness function, the multiple types of link matrices are selected among many kinds of dependency relations to fit the fitness function. The Delta Table is constructed of each type of dependency relation, and multiple Delta Tables make up the Multi-criteria Delta Table. This increases the flexibility of the usage of the Delta Tables. The detailed method for ranking refactoring candidates in the Multi-criteria Delta Table will be explained in Section 4.1.2.

2.2.2 Adaptive Delta Table for Extending to Other Types of Refactorings

The large-scale refactoring consists of elementary-level refactorings, such as Move Method and Move Field refactorings. This is also supported by other literature, as "*each of the classes consists of a set of methods moved from the original class, and therefore, the Move Method refactoring is a special case of the more general Extract Class refactoring*" [28]. Each element of the Delta Table is designed to have the maintainability variance after moving a field or a method; thus, it can be extended to assess other types of refactorings, such as Extract Class and Extract Method refactorings.

To extend it to assess other types of refactorings, the size of the Delta Table should be adjustable to adapt the added or deleted entities or classes. In short, classes, method, and fields can be newly created or deleted, thus the length of the rows or columns of the membership, link, and Delta Table matrices can be changed. The techniques for the internal implementation to make the efficient adaptive Delta Table are described in Section 2.2.3.

The change amounts after applying a big refactoring can be obtained by applying Move Field refactorings or Move Method refactorings, which are constituents of the big refactoring, one after another. For each move for a method or a field, the membership and link matrices are updated, and only the changed elements of the Delta Table are calculated by multiplying those matrices.

The illustrative example of applying Extract Class refactoring is presented in Fig. 3. The code before and after applying the refactoring is shown in Fig. 3a. The method `getTelephoneNumber()` is extracted from the class `Person` and moved to the newly created class `TelephoneNumber`. The fields `areaCode` and `number` are moved to class `TelephoneNumber`, and the extracted method is renamed to `getDisplayableString()`. In the old class `Person`, the new field `telephoneNumber` with the `TelephoneNumber` class type is created and a getter method, `getTelephoneNumber()`, is added to retrieve the object. As a result, after applying the Extract Class refactoring, the membership and link matrices in Fig. 3b are adapted as the matrices in Fig. 3c.

It should be noted that the automated identification of opportunities in big refactorings, such as finding logical units of data that are grouped together for Extract Class refactoring, is beyond the scope of this paper. When those types of refactoring candidates are given, we can use the Delta Tables to assess only the impact of the refactoring candidates.

2.2.3 Performance Improvement

The ability to efficiently assess a large number of refactoring candidates is the most important contribution of this paper. In the actual implementation, we applied new techniques for maximizing performance.

Merging Internal and External Link Matrices. Except for newly established relations for the added entities, the relations of the entities remain the same. By applying a refactoring, the fact that the dependency relation involves an entity referring to another entity does not change; rather, only the membership information that states which class an entity belongs to is altered. Thus, the link matrices do not necessarily need to be updated for each application of a refactoring.

We merged the separate internal and external matrices, $\mathbf{L_{Int}}$ and $\mathbf{L_{Ext}}$, into one link matrix. Updating internal and external link matrices and multiplying those link matrices with a membership matrix in every application of a refactoring, for instance, results in a high computational cost. The Delta Table calculation in Equation (1) is equivalent to Equation (2), and it can be converted to Equation (3), which consists of matrices of $\mathbf{L}$ and $\mathbf{M}$

$$\mathbf{D} = \mathbf{L_{Int}} \times \mathbf{Inv}(\mathbf{M}) - \mathbf{L_{Ext}} \times \mathbf{M}. \quad (2)$$

$$\mathbf{D} = (\overbrace{\mathbf{L} \otimes \mathbf{MM}^\top}^{\mathbf{L_{Int}}}) \times (\mathbf{1} - \mathbf{M}) - (\overbrace{\mathbf{L} \otimes (\mathbf{1} - \mathbf{MM}^\top)}^{\mathbf{L_{Ext}}}) \times \mathbf{M}. \quad (3)$$

Each element constituting the Equation (3) is explained as follows.

- $\otimes$: element-wise multiplication
- $\mathbf{1}$: all-ones matrix
- $\mathbf{M}^\top$: transposed $\mathbf{M}$
- $\mathbf{1} - \mathbf{M}$: $\mathbf{Inv}(\mathbf{M})$
- $\mathbf{MM}^\top$: multiplication of $\mathbf{M}$ and $\mathbf{M}^\top$
- $\mathbf{1} - \mathbf{MM}^\top$: converted $\mathbf{MM}^\top$

Here, $\mathbf{1}$ represents the matrix over the real numbers, where every element is equal to one. Subtracting $\mathbf{M}$ from $\mathbf{1}$

```
/* ------ Before ------ */

class Person
{
    private String name;
    private String areaCode;
    private String number;

    public String getTelephoneNumber()
    {
        return areaCode + "-" + number;
    }
}


/* ------ after ------ */

class Person
{
    private String name;
    private TelephoneNumber telephoneNumber;

    public TelephoneNumber getTelephoneNumber()
    {
        return telephoneNumber;
    }
}


class TelephoneNumber
{
    private String areaCode;
    private String number;

    public String getDisplayableString()
    {
        return areaCode + "-" + number;
    }
}
```

(a) Example code of Extract Class refactoring.

M	Person
name	1
areaCode	1
number	1
getTelephoneNumber()	1

L	name	areaCode	number	getTelephoneNumber()
name	0	0	0	0
areaCode	0	0	0	1
number	0	0	0	1
getTelephoneNumber()	0	1	1	0

(b) Membership and link matrices of $\mathbf{M}$ and $\mathbf{L}$ before refactoring.

M'	Person	TelephoneNumber
name	1	0
areaCode	0	1
number	0	1
getDisplayableString()	0	1
telephoneNumber	1	0
getTelephoneNumber()	1	0

L'	name	areaCode	number	getDisplayableString()	telephoneNumber	getTelephoneNumber()
name	0	0	0	0	0	0
areaCode	0	0	0	1	0	0
number	0	0	0	1	0	0
getDisplayableString()	0	1	1	0	0	0
telephoneNumber	0	0	0	0	0	1
getTelephoneNumber()	0	0	0	0	1	0

(c) Membership and link matrices of $\mathbf{M}'$ and $\mathbf{L}'$ after applying Extract Class refactoring.

Fig. 3. Adapting membership and link matrices for extract class refactoring.

produces the converted matrix of $\mathbf{M}$, which is equivalent to $\mathbf{Inv}(\mathbf{M})$, by changing $0 \rightarrow 1$ and $1 \rightarrow 0$.

As shown in Fig. 4, it is interesting that by multiplying $\mathbf{M}$ and $\mathbf{M}^\top$, we can obtain the matrix in which each element $(\mathbf{MM}^\top)_{ij}$ is 1 when entity i and entity j are in the same class. Consequently, the element-wise multiplication of $\mathbf{MM}^\top$ and $\mathbf{L}$ produces $\mathbf{L_{Int}}$. In contrast to this, in the converted matrix of $\mathbf{MM}^\top$, $\mathbf{1} - \mathbf{MM}^\top$, each element is 1 when entity i and entity j are located in different classes. Thus, the element-wise multiplication of $(\mathbf{1} - \mathbf{MM}^\top)$ with $\mathbf{L}$ produces $\mathbf{L_{Ext}}$.

Refactoring Impact Analysis. For each iteration of the refactoring process, a refactoring is selected and the refactoring is applied by changing the membership matrix. Thus, to reflect the changes, the Delta Table should be recalculated.

For efficient computation, we need to identify the changed elements after applying a refactoring, which is regarded as *refactoring impact analysis*. Only these elements are used to recalculate the Delta Table.

When moving a method m from a class c_1 (source class) to a class c_2 (target class), let R(m) be the entities that are affected by applying the refactoring, that is, $\mathrm{R}(m) = \{e \mid (m, e) \in \mathbf{L} \text{ and } (e \in c_1 \text{ or } e \in c_2)\}$. Then, the elements that are need to be examined for recalculating the Delta Table are as follows: $\mathbf{L}' = \{\mathbf{L}_{ij}, i \in \mathrm{R}(m) \text{ and } j \in \mathrm{R}(m)\}$ and $\mathbf{M}' = \{\mathbf{M}_{ij}, i \in \mathrm{R}(m) \text{ and } j = c_1 \text{ or } c_2\}$.

Matrix Computation Using Special Libraries. To accelerate the speed, the C++ library, Eigen [22], is used for matrix computation. Compared to the previous version, which used SciPy [29] libraries implemented for Python, Eigen significantly improves the speed of the system due to features described below.

Eigen [22] supports resizing the matrix, which is an essential function for constructing the effective adaptive Delta Table (Section 2.2.2). To assess the impact of Extract Class or Extract Method refactorings, the size of the membership and link matrices should be changed for added/deleted entities or classes. Without resizing, entire matrices would need to be copied in the memory while executing, to adapt even a small number of entities or classes, which is inefficient.

Furthermore, Eigen [22] provides the view concept, which allows read-write access to a part of the matrix (i.e., submatrix such as a column or a row of a matrix). This is a critical operation when only a few elements are changed for the Delta Table computation. Most of the link and

M	class A	class B
m1	1	0
m2	1	0
m3	1	0
m4	0	1
m5	0	1
f1	1	0
f2	0	1
f3	0	1
f4	0	1

M^T	m1	m2	m3	m4	m5	f1	f2	f3	f4
class A	1	1	1	0	0	1	0	0	0
class B	0	0	0	1	1	0	1	1	1

(a) $\mathbf{M}$ (Fig. 1(d)) and $\mathbf{M}^\top$

MxM^T	m1	m2	m3	m4	m5	f1	f2	f3	f4
m1	1	1	1	0	0	1	0	0	0
m2	1	1	1	0	0	1	0	0	0
m3	1	1	1	0	0	1	0	0	0
m4	0	0	0	1	1	0	1	1	1
m5	0	0	0	1	1	0	1	1	1
f1	1	1	1	0	0	1	0	0	0
f2	0	0	0	1	1	0	1	1	1
f3	0	0	0	1	1	0	1	1	1
f4	0	0	0	1	1	0	1	1	1

(b) $\mathbf{M} \times \mathbf{M}^\top$

Fig. 4. Multiplication of membership matrix and its transposed matrix denotes the pairs of entities locating in the same class.

membership matrices are sparse matrices; however, many of the libraries support only dense matrices, which are comparably easy to implement. Eigen supports efficient submatrix computation via the view concept for sparse matrices, which is useful after performing the refactoring impact analysis, and it consequently has a major effect on speeding up our approach.

3 Refactoring Identification Process

A simple way to find the best sequence refactorings that most improves maintainability is to generate all possible schedules and select the most beneficial one; however, such a brute-force approach is obviously time consuming [5]. Theoretically, when the number of possible refactorings is k, the impact of the design quality for the application of $k!$ (k-permutations) number of sequences of refactoring candidates should be assessed. As the number of refactorings increases, the number of possible refactoring schedules increases exponentially. Therefore, scheduling refactorings by investigating all possible sequences may become NP-hard.

The search-based refactoring studies in [5], [30], [31], [32] use global search techniques, such as a genetic algorithm (GA). To find the optimal refactoring sequence (or an appropriate refactoring schedule) with a reasonable computational cost, they first generate the sequences of refactorings using the GA and then evaluate the effects of the application on those sequences. However, if solutions are generated without considering the dependencies of the relations between entities, many infeasible solutions would be produced, which thus might require additional repair costs in the practical application of the chosen order of refactorings. This problem is also noted in [33], which demonstrates that the core operations of a traditional GA are based on random selections; thus, the results of GA scheduling may correspond to infeasible solutions. The researchers in that study also pointed out the performance degradation issue that arises when considering many constraints.

For each iteration of the refactoring identification process, our approach locally searches the refactoring candidates, and a refactoring is selected—either interactively by accepting a developer's feedback or automatically by choosing the refactoring that improves the maintainability of a fitness function to the greatest extent—and the refactoring is applied. Consequently, the refactorings selected are applied one after another in a stepwise fashion. This approach belongs to the family of stepwise refactoring recommendation techniques [10], [34], [35], [36], [37]. JDeodorant [34], [35] is one of the most influential research tools for automated refactoring identification.

The application of refactoring changes the design configuration. Thus, the effects of the application for all available refactoring candidates should be evaluated again at every step of the refactoring process in order to select the next refactorings. The whole of the refactoring identification process is performed repeatedly and is terminated when there are no more candidates improving maintainability. Meanwhile, developers can interactively select preferable refactorings among the suggested refactoring candidates for each iteration of the refactoring process, and they can stop this process when the refactoring goal is accomplished.

3.1 Overall Search Process

Fig. 5 represents the overview of the two-phase assessment approach in the refactoring identification process. In the first level, the effects of refactoring candidates of all possible moves of entities to other classes are preliminarily assessed using the Delta Tables where elements have approximate values to estimate the maintainability of the designs transformed by Move Method or Move Field refactorings; the candidates with the top $k\%$ ranked in the Multi-criteria Delta Table are chosen. In the second level, the chosen candidates are assessed using a fitness function (e.g., maintainability evaluation function or metric), which can be regarded as a rigorous assessment of maintainability.

The proposed approach is presented in relation to the search-based problem as follows.

Representation of the Solution. For each iteration of the refactoring process, a refactoring can be selected either interactively by the developer or automatically by choosing the refactoring that best suits a fitness function of the maintainability; then, the refactorings are applied one after another in a stepwise fashion. The final solution is the sequence of Move Method refactorings selected in each process.

Change Operators. For every iteration of the process, the possible search space is composed of feasible moves of methods to other classes (i.e., Move Method refactorings) that can be applied from the current design and that preserve preconditions for behavior preservation and quality assurance. In similar fashion to [10], [15], [30], we consider only Move Method refactoring. Move Field refactoring is excluded because fields are strongly conceptually bound to the classes in which they are initially placed, and they are less likely to change once assigned to a class [10]. To provide suggestions for desirable refactorings in an automatic manner, the trials of refactored designs—each of which can be transformed by applying a refactoring—are investigated for obtaining the effects of their application on maintainability.

Fitness Functions. To evaluate the maintainability of the refactored designs, we have chosen the following metrics or functions as the fitness functions in the experiment:

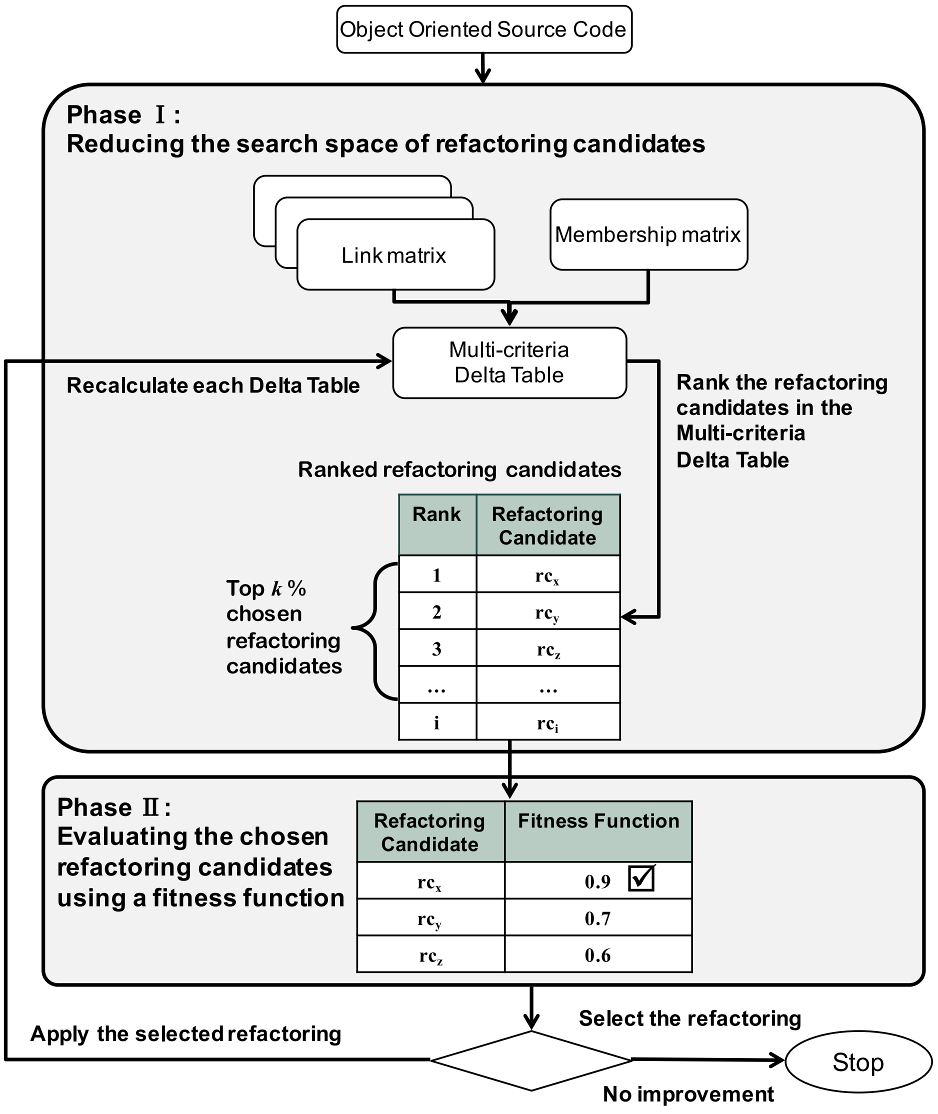

Object Oriented Source Code
Phase Ⅰ:
Reducing the search space of refactoring candidates
Link matrix
Membership matrix
Multi-criteria Delta Table
Recalculate each Delta Table
Rank the refactoring candidates in the Multi-criteria Delta Table
Ranked refactoring candidates
Rank | Refactoring Candidate
1 rc_x
2 rc_y
3 rc_z
… …
i rc_i
Top k % chosen refactoring candidates
Phase Ⅱ:
Evaluating the chosen refactoring candidates using a fitness function
Refactoring Candidate | Fitness Function
rc_x 0.9 ☑
rc_y 0.7
rc_z 0.6
Select the refactoring
Apply the selected refactoring
No improvement
Stop

Fig. 5. Overview of the approach using two-phase assessment in the refactoring identification process.

Connectivity (cohesion metric) [14], Message Passing Coupling (MPC) (coupling metric) [13], and Entity Placement metric (EPM) [10]. In addition to these three measures, the multi-objective functions for two objectives (e.g., MPC versus Connectivity, EPM versus MPC, and EPM versus Connectivity) are also used. Since calculating a fitness function for all possible Move Method refactoring candidates may be cost intensive, the candidates are preliminarily assessed using the Delta Tables, and then only the candidates that are top k% ranked based on the Multi-criteria Delta Table, which is composed of the Delta Tables, are chosen to be evaluated with the fitness function. More detailed explanations regarding each fitness function are provided in Section 4.2.

Search Algorithm and Heuristics. Steepest-ascent hill climbing is a local search algorithm in which the search examines all neighboring solutions and moves to the paths exhibiting the highest quality improvement [38], [39]. When trying to automatically select a refactoring, this search technique is used to find the best refactoring for a fitness function of maintainability.

When a multi-objective function is used as a fitness function, one of the refactoring candidates is randomly selected from the set of Pareto optimal candidates (i.e., Pareto front or Pareto set). It is important to emphasize that the multi-objective fitness function does not mean to use the multi-objective search. For fast and flexible processing, we adopt the stepwise interactive search, which locally searches refactoring candidates and then iteratively selects and applies a refactoring for each step of the refactoring process. The rationales to use this search is explained in Section 3.2.

Apart from the search technique, when choosing the candidates in the Multi-criteria Delta Table, inferior candidates (i.e., those ranked below 5 percent) are discarded and replaced with randomly chosen candidates. Randomness is useful in that it has the effect of a simulated annealing search [40] that permits a series of changes that degrade the solution to allow the search to escape from the local minimum.

We assume that there is no information regarding which refactoring candidates improve the maintainability; thus, all possible transformations become the subjects of refactoring opportunities. In our approach, we use elementary-level Move Method refactorings.

A refactoring candidate that is highly ranked using the Delta Tables is not always guaranteed to have a higher fitness value. In other words, some candidates may have the higher values in fitness even though they have not been chosen as promising candidates using the Delta Tables in the first phase of the assessment. To provide a chance to be assessed with fitness functions for those candidates, a refactoring candidate that has not been selected continuously through the several iterations is excluded from consideration. This can be implemented by increasing the counter of the selected refactoring candidate. We made three attempts (i.e., $\gamma = 3$) to find a solution. When the counted number of the refactoring candidates exceeded the specific number γ, the refactoring candidate is excluded as a candidate.

The proposed approach does not refactor an object-oriented program in a fully automated manner; rather, it automatically identifies a set of refactoring candidates that can be safely applied for delivering improvements in maintainability. Thus, for every iteration of the refactoring identification process, the software developers make the final decision on whether to apply the suggested refactorings. In short, the developers should examine whether the suggested refactorings conform to the design practice and principles of the development team. Although the recommended refactorings are beneficial from a maintainability perspective, the application of the refactorings can be rejected, if they conflict with other design principles, such as reusability, flexibility, or understandability.

3.2 Interactive Local Search Method

We provide the rationales to accommodate the interactive local search method.

Dependency-Aware Refactoring Selection. The application of each refactoring changes the design of the program, and subsequently, the effects of other refactoring candidates are also changed; some of the candidates become invalid to be applied, and new refactoring opportunities are introduced. In evolutionary computation, this phenomenon is known as epistasis [41], [42]. The effects of the refactoring candidates should be recalculated for every refactoring application. Thus, to consider refactoring dependencies, refactorings are chosen and applied, one for each refactoring process step.

Fast (Online or Real-Time) Processing by Taking the Developer's Feedback. As the research trend changes to interactive and real-time processing methods, fast processing is more highly valued and is gaining more attention. We assume that the environment in which this approach is to be used will have limited (not infinite) resources and time, and in this context, identifying refactorings in order to apply them more quickly is indeed important. We believe that finding a globally optimized sequence of refactorings that takes several hours or days is not practical. Developers would not wait if the time is too long. Furthermore, interactive selection provides a more flexible method in which developers can actively make a decision regarding the refactoring application. Our approach provides a set of refactoring candidates ranked by a fitness function for each iteration of the process; thus, either the developers can choose their preference of refactoring to apply, or they can use the automated execution mode to select the best one. Developers can also stop performing refactoring identification at any time, once the goal of the maintainability improvement reasonably has been accomplished through the application of the refactorings selected to that point in time. Note that the aim of the interactive local search approach is not to find the optimal sequence of refactorings that reach the maximum improvement in maintainability.

4 Two-Phase Refactoring Assessment

In the following sections, we provide a detailed explanation for each step of the two-phase assessment approach.

4.1 Phase I: Reducing the Search Space of Refactoring Candidates Using the Multi-Criteria Delta Table

Fig. 6 illustrates a procedure for identifying possible refactoring candidates and ranking them in the Multi-criteria Delta Table. In order to increase computational efficiency, the number of refactoring candidates to be evaluated with a fitness function is restricted to the top $k\%$ of the Multi-criteria Delta Table.

4.1.1 Identifying Possible Refactoring Candidates

A Delta Table is constructed of each type of dependency relation. All methods existing in the system are considered to be moved to other classes, and these Move Method refactoring candidates are the elements of a Delta Table. The candidates that have passed the behavior preservation preconditions become the subjects of assessment using a fitness function. Finally, multiple Delta Tables make up the Multi-criteria Delta Table, and refactoring candidates in which at least one of each Delta Table has a negative value are considered. We refer to them as the possible refactoring candidates.

The possible refactoring candidates are identified as described below.

1. Calculating Delta Tables. For each type of link matrix, a Delta Table is calculated following the method explained in Section 2.1.

In the experiment, three types of dependency relations are considered, and three Delta Tables are obtained. The specific dependency relations are established when (1) a method calls another method, (2) a method accesses a field, or (3) two methods access the same field.

It is important to emphasize that the major change in version 2.0 of the Delta Table is that we adopt a Multi-criteria Delta Table. Various types of link matrices can be used, and multiple Delta Tables are obtained, which increases the flexibility in choosing the fitness functions by which developers aim to accomplish the refactoring goals.

When applying Move Method refactoring, it is a poor idea to move a method that already exhibits strong cohesion with the original class; thus, such a cohesive refactoring candidate should be assigned a lower priority for moving. To reduce the effect of a highly cohesive method, the values for the elements in the Delta Table are adjusted by dividing them by the number of internal relations, as follows:

$$\frac{|i| - |e|}{|i|},\ where\ i \neq 0.$$

When i is zero, the value of the Delta Table is not adjusted. That is, moving a method with no internal relations will not increase the external relations of the overall system, and we do not need to assign a penalty to it.

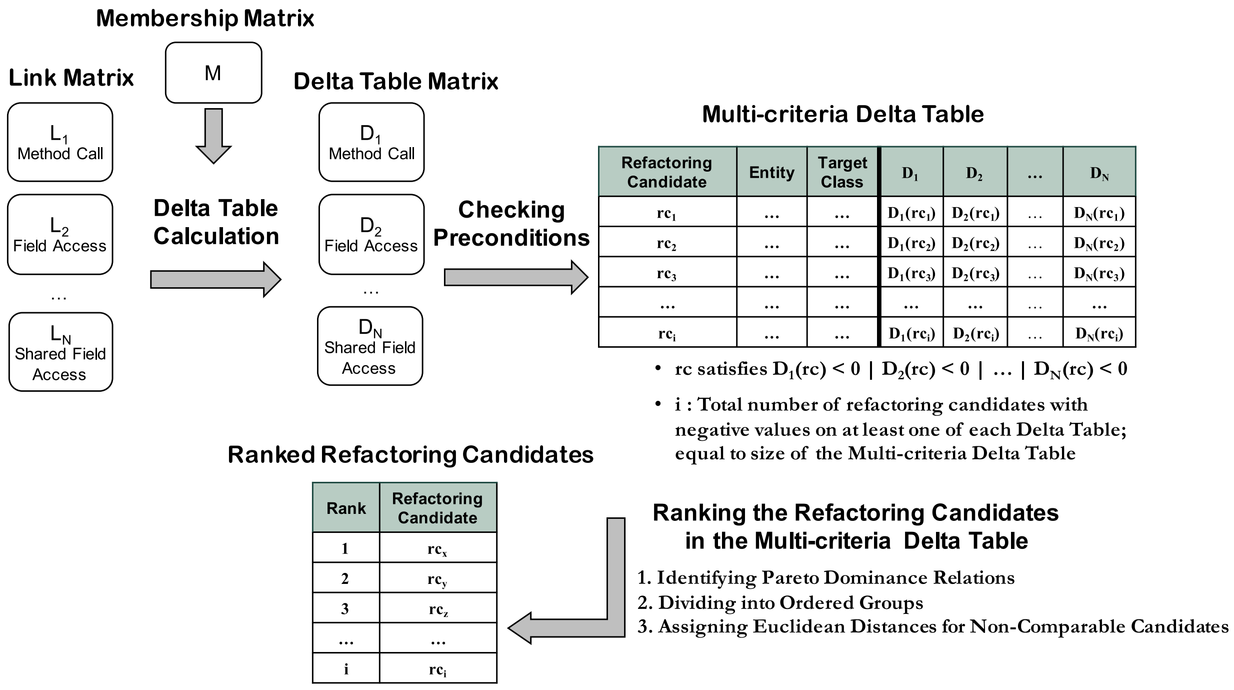

Membership Matrix
M
Link Matrix
L1 Method Call
L2 Field Access
...
LN Shared Field Access
Delta Table Calculation
Delta Table Matrix
D1 Method Call
D2 Field Access
...
DN Shared Field Access
Checking Preconditions
Multi-criteria Delta Table
Refactoring Candidate | Entity | Target Class | D1 | D2 | ... | DN
rc1 ... D1(rc1) D2(rc1) DN(rc1)
rc2 ... D1(rc2) D2(rc2) DN(rc2)
rc3 ... D1(rc3) D2(rc3) DN(rc3)
rci ... D1(rci) D2(rci) DN(rci)
• rc satisfies D1(rc) < 0 | D2(rc) < 0 | … | DN(rc) < 0
• i : Total number of refactoring candidates with negative values on at least one of each Delta Table; equal to size of the Multi-criteria Delta Table
Ranked Refactoring Candidates
Rank | Refactoring Candidate
1 rcx
2 rcy
3 rcz
... ...
i rci
Ranking the Refactoring Candidates in the Multi-criteria Delta Table
1. Identifying Pareto Dominance Relations
2. Dividing into Ordered Groups
3. Assigning Euclidean Distances for Non-Comparable Candidates

Fig. 6. A procedure for identifying possible refactoring candidates and ranking them in the Multi-criteria Delta Table.

For example, let us assume that there are two methods a and b whose ($|i|$, $|e|$) values are a = (1, 2) and b = (10, 11), respectively. While the values assigned to the Delta Table are the same (e.g., -1) for both methods, method b is apparently highly cohesive to the original class; thus, it should not be selected as a candidate for refactoring. Therefore, the impact of method b should be minimized. The values for methods a and b are adjusted into $\frac{1-2}{1} = -1$ and $\frac{10-11}{10} = -0.1$, respectively. It is better to select to move method a than method b as a refactoring candidate.

2. Checking preconditions. Improving the practical applicability of this approach requires stringent preconditions for behavior preservation and quality assurance. The preconditions defined in [10], [35] are used.

For every iteration of the process, before choosing the refactoring candidates that should undergo the assessment using a fitness function, the set of preconditions are checked if the available refactoring candidates can be applied without changing the systems behavior. When candidates do not meet the preconditions, then they are not considered for further assessment.

The Delta Table is extended to adapt other types of refactorings, and rigorous and strict preconditions are introduced for quality assurance, which is essential for considering practical applicability. Examples of the preconditions include the condition that the method to be moved should not contain the assignment of a source class field, or that the method to be moved should have the one-to-one relationship with the target class, and so on.

3. Identifying the refactoring candidates in the Multi-criteria Delta Table. Finally, possible Move Method refactoring candidates are identified. As explained in Section 2.1, a refactoring candidate with a negative value in each Delta Table is considered to improve maintainability. Thus, in the Multi-criteria Delta Table, refactoring candidates in which at least one of each Delta Table has a negative value are left.

4.1.2 Ranking Refactoring Candidates Using the Multi-Criteria Delta Table

The refactoring candidates are ranked in the Multi-criteria Delta Table as described below.

1. Identifying Pareto dominance relations between refactoring candidates. In the Multi-criteria Delta Table, to identify refactoring candidates that are more likely to improve maintainability, the concept of Pareto dominance [43] in multi-objective optimization problems is used. Pareto dominance is a binary relation between two solutions. Solution A is said to dominate solution B if all components of A are at least as good as those of B with at least one strictly better component [21]. Meanwhile, solution A is non-dominated if it is not dominated by any solution; then, solution A is the Pareto optimal and the set of Pareto optimal solutions is the Pareto front or Pareto set.

From this concept, we can derive the dominance relation between two refactoring candidates rc_x and rc_y as follows. Refactoring candidate rc_x is considered better than refactoring candidate rc_y, when at least one element of the Delta Table for refactoring candidate rc_x is smaller than that for refactoring candidate rc_y, and rc_x is smaller than or equal to rc_y in elements of all other Delta Tables. Note that refactoring candidates with smaller values in Delta Tables are regarded to improve maintainability more; thus, the objectives are converged to minimize the values in each Delta Table. It is said that rc_x dominates rc_y, and this dominance relation is denoted as the partial order $rc_x \preceq rc_y$.

Pareto dominance relation of two refactoring candidates can be summarized and formalized as follows. Refactoring candidate rc_x dominates refactoring candidate rc_y, $rc_x \preceq rc_y$, if and only if there is m s.t. $\mathbf{D}_m(rc_x) < \mathbf{D}_m(rc_y)$ and $\mathbf{D}_n(rc_x) \leq \mathbf{D}_n(rc_y)$ for all $n \in \{1, \ldots, N\}$, where N is the number of the Delta Table matrices. In short, rc_x is at least as good as rc_y for all Delta Table values and it is better than rc_y for at least one Delta Table value.

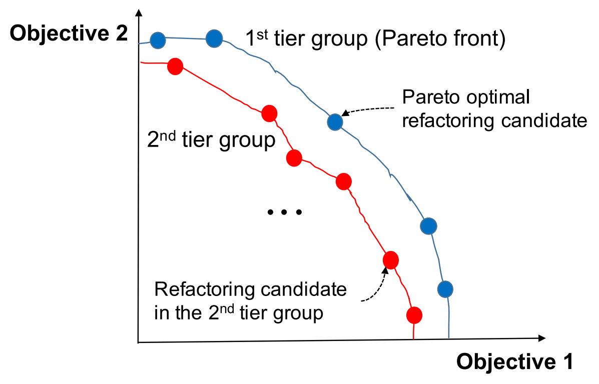


Fig. 7. Distribution of refactoring candidates in the Pareto concept for the multi-objective optimization problem. This example shows bi-objectives.

2. Partitioning into the ordered groups of refactoring candidates based on Pareto dominance relations. The Pareto dominance relation is a partial order; thus, the refactoring candidates are partitioned into several ordered groups.

Fig. 7 presents an illustrative example of the distribution of refactoring candidates in the Pareto concept of the multi-objective optimization problem. The first-tier group can be regarded as the Pareto front. Thus, to find the best solution, one of the Pareto optimal solutions from the Pareto front can be chosen according to user preferences in the various types of Delta Tables.

However, finding the best solution is not the main concern in our approach. We investigate the partitioned groups in an order of highly ranked tiers to identify the refactoring candidates that are more likely to have better values in fitness functions and improve maintainability. Thus, we need to quantify each refactoring candidate belonging to the same tier group to rank these refactoring candidates. The distribution of the candidates for each group forms the convex curve; thus, the euclidean distance [44] is used to obtain the approximate values of the candidates.

3. Assigning approximate values based on euclidean distances for non-comparable refactoring candidates. To calculate the euclidean distance, values of the Delta Table should be adjusted. First, components laid for each dimension of the objective should be normalized. Thus, each value of the Delta Table contained in the same group is normalized as $(\mathbf{D}(rc) - min)/(max - min)$, where min and max indicate the minimum and maximum Delta Table values in the group, respectively. Second, positive values in Delta Tables are replaced by zero. Only the negative values in each Delta Table are considered to disregard the effects of the Delta Tables that do not improve maintainability.

The euclidean distance for refactoring candidate rc_i is calculated as follows:

$$\sqrt{(\mathbf{D}_1(rc_i))^2 + \cdots + (\mathbf{D}_N(rc_i))^2},\ where\ \mathbf{D}(rc_i) \leq 0.$$

To this end, the refactoring candidate with the greater value in the euclidean distance is highly ranked in the Multi-criteria Delta Table.

We briefly provide an illustrative example for ranking the refactoring candidates in the Multi-criteria Delta Table. When two Pareto dominance relations are identified, for example, $rc_x \preceq rc_y$ and $rc_x \preceq rc_z$, two partitioned, ordered groups are established as follows: the first tier as (rc_x) and the second tier as $(rc_y,\ rc_z)$. Since the refactoring candidates of rc_y and rc_z belong to the same group and are not comparable, euclidean distance values are assigned to each candidate to rank them. If rc_y is determined to have a greater euclidean distance value than rc_z, then the final rank order is rc_x, rc_y, and rc_z.

4.2 Phase II: Evaluating the Chosen Refactoring Candidates Using a Fitness Function

Real assessments for the refactoring candidates chosen by the Delta Table are performed using a fitness function. There are many kinds of metrics or functions for evaluating maintainability; these can be used as fitness functions for guiding the search for refactorings.

It is important to explain the fitness functions used in the experiment. In general, lower coupling and higher cohesion are most representative properties for measuring maintainability. We chose Connectivity [14] and MPC [13] for the cohesion and coupling metrics, respectively, because they are used as evaluation measures in other Move Method identification studies [10], [37]. Coarse-grained metrics, such as Coupling Between Objects (CBO) [45] and Coupling Factor (CF) [46], are not appropriate as fitness functions in our approach, as their values do not change when a Move Method refactoring is applied.

To investigate the effect of a measure taking into account both cohesion and coupling aspects, the EPM [10] is used. The EPM captures both cohesion and coupling into a single formula. For a given class, the numerator represents the distances of the entities belonging to a class from the class itself (i.e., the cohesion of a class), while the denominator represents the distances of the entities not belonging to a class from this class (i.e., coupling of this class to the rest of the system). Since the QMOOD [47] is ambiguous in some cases [48] and the weights of the model are not fixed, we exclude the QMOOD.

Multi-objective functions each of which uses two objectives (e.g., MPC versus Connectivity, EPM versus MPC, or EPM versus Connectivity) are also used. The bi-objective function helps to identify refactorings that best compromise each objective (e.g., MPC, Connectivity, or EPM).

We provide a more detailed explanation of how to calculate each fitness function below.

Entity Placement Metric. EPM [10] measures how many entities have correctly been placed and uses distances instead of similarities in its computation. Entities with smaller distances from a target class should be moved and placed together to improve maintainability. Therefore, a candidate with a lower EPM metric is considered a promising candidate.

EPM for a class C is given by

$$EPM_C = \frac{\frac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\frac{\sum_{e_j \notin C} distance(e_j, C)}{|entities \notin C|}},$$

where e denotes an entity (e.g., method or field) of the system. The similarity between an entity e and a class C can be calculated as the cardinality of intersection over union for the two entity sets of entity e and class C. The distance (e, C) is obtained by subtracting the similarity value from 1.

TABLE 1
Characteristics of Each Subject

Project	Description	LOC	♯Classes	♯Entities (♯Methods + ♯Fields)	♯Methods	♯Fields
`Apache Ant` 1.9.6	Java application build tool	222,256	1,186	16,268	10,546	5,722
`JGit` 4.1.0	Distributed source version control system	166,415	946	11,686	7,543	4,143
`JHotDraw` 7.0.6	Java GUI framework	135,233	751	10,313	7,314	2,999

To obtain a system-level metric, EPM for each class is weighted by the ratio of the number of entities in the class ($|E_C|$) to the number of entities in the system ($|E_S|$), and all weighted EPMs for all classes in the system ($|C_S|$) are added as follows:

$$EPM_{System} = \sum_{i=1}^{|C_S|} \frac{|E_{C_i}|}{|E_S|} EPM_{C_i}.$$

Message Passing Coupling. MPC [13] for a class C is defined as the number of invocations of methods not implemented in class C but by the methods of class C. The MPC evaluates coupling by employing the total number of method invocations, while the other metrics (e.g., Request for a Class (RFC) [49]) measure the number of distinct methods invoked. MPC for a class C is given by

$$MPC_C = |me_C|,$$

where me_C denotes the calls from the class C to the methods defined in external classes. To obtain a system-level metric, we use the average MPC for a class.

Connectivity. Connectivity [14] for a class C is defined as the number of the common method pairs (let it be cmp in which one method invokes the other method or both access a common attribute of the class) over the total number of method pairs of the class. It is different from the other cohesion metrics because it considers two methods, m_1 and m_2, to be cohesive both when they access a common attribute and when m_1 invokes m_2 or vice versa. Connectivity for a class C is given by

$$Connectivity_C = \frac{|cmp_C|}{\frac{|M_C|!}{2(|M_C|-2)!}},$$

where cmp_C and M_C denote the sets of common method pairs and methods in class C, respectively, while $\frac{|M_C|!}{2(|M_C|-2)!}$ represents the possible number of method pairs in the class. To obtain a system-level metric, we use the average Connectivity for a class.

It is important to note that the direction of convergence, wherein refactoring candidates are considered to improve maintainability, depends on the fitness function. In the fitness functions above, refactoring candidates are suggested to have smaller values in MPC and EPM and larger values in Connectivity; thus, the directions of convergence are negative ($-$) for MPC and EPM and positive ($+$) for Connectivity. In other words, refactoring candidates that are evaluated as having values in those convergence directions are regarded as having positive effects on improving maintainability.

5 Evaluation

We designed our evaluations to address the following research questions:

RQ1 Is the two-phase assessment approach efficient? In other words, does the method of reducing the search space by choosing refactoring candidates using the Multi-criteria Delta Table and examining only those candidates based on a fitness function yield a good solution in less time?

RQ2 What is the effect of the number of the chosen refactoring candidates (top k%)?

RQ3 How well do the Delta Tables correctly identify refactoring candidates that have actual higher (better) fitness function values?

To consider projects with various characteristics, we have chosen the following three open source projects: `Apache Ant` [23], `JGit` [24], and `JHotDraw` [25]. They have been widely used as experimental subjects in other literature. Table 1 summarizes the characteristics of each project. `JHotDraw` and `Apache Ant` are both stand-alone programs; `JHotDraw` is a GUI program, while `Apache Ant` is a non-GUI program. `JGit` is a Java library implementing the Git version control system. These projects are written in Java and contain a relatively large number of classes.

Multiple runs of the refactoring identification process using the two-phase assessment (i.e., 3 projects $\times$ 6 fitness functions $\times$ 5 varied size of the Multi-criteria Delta Table $=$ 90 times in total) are performed in an automated manner. Refactoring candidates for which one or more of each Delta Table has negative values constitute the Multi-criteria Delta Table; thus, the total number of these candidates determines the size of the Multi-criteria Delta Table. In addition to considering all possible refactoring candidates, we performed the approaches for selecting various number of the candidates to the top k% (e.g., 5, 10, 20, 50 percent) of the Multi-criteria Delta Table and evaluating them by each fitness function, including MPC, Connectivity, EPM, and the multi-objective functions for two objectives (e.g., MPC versus Connectivity, EPM versus MPC, and EPM versus Connectivity). The maximum number of iterations for the refactoring process is set to 100 in this experiment.

5.1 RQ1. Efficiency of the Computation of the Two-Phase Assessment Approach

Experimental Design. To investigate the efficiency of the computation of the proposed two-phase assessment approach, we measured the total elapsed time for performing the entire process. The obvious measure denoting the efficiency of the computation is time. In addition to time, the accumulated number of assessed refactoring candidates, which is proportional to time, is collected. Regarding the efficiency

TABLE 2
Overview of the Results: Total Time, Speed Up, the Number of Assessed Refactoring Candidates, and Final Accomplishments of the Fitness Functions After 100 Iterations

Project	Approach		Computational Cost			Achievement in Fitness Function
	Fitness Function	Reduction	Total Time (sec)	Speed Up	Total Candidate (#)	Final Value [Δ amounts]
`Apache Ant`	MPC	Delta top 20%	321.52	5.2	17,500	13.9098 [−0.4359]
		No-reduction	1,660.08		194,761	13.9089 [−0.4368]
	Connectivity	**Delta top 20%**†	1,108.03	13.5	17,500	0.312 [+0.0174]
		No-reduction	14,930.60		194,854	0.312 [+0.0174]
	EPM	**Delta top 20%**†	1,628.22	9.8	17,500	0.9031 [−0.0013]
		No-reduction	15,954.59		195,430	0.9031 [−0.0013]
`JGit`	MPC	Delta top 20%	249.39	5.7	20,200	13.2801 [−0.4525]
		No-reduction	1,421.43		233,554	13.2780 [−0.4546]
	Connectivity	**Delta top 20%**†	535.71	8.7	20,200	0.3666 [+0.0382]
		No-reduction	4,671.66		235,286	0.3666 [+0.0382]
	EPM	**Delta top 20%**†	1,076.57	10.3	20,200	0.896 [−0.0031]
		No-reduction	11,077.33		236,005	0.896 [−0.0031]
`JHotDraw`	MPC	**Delta top 20%**†	137.35	2.6	5,400	16.9055 [−0.2569]
		No-reduction	354.80		54,162	16.9055 [−0.2569]
	Connectivity	**Delta top 20%**†	295.17	6.3	5,400	0.2901 [+0.0189]
		No-reduction	1,864.17		53,778	0.2901 [+0.0189]
	EPM	**Delta top 20%**†	221.32	7.2	4,266	0.9204 [−0.0005]
		No-reduction	1,603.48		47,153	0.9204 [−0.0005]

• *For improving maintainability, fitness functions should be increased or decreased: MPC (−), Connectivity (+), and EPM (−).*
• *Speed Up x means that time of Delta top 20 percent approach (our approach) is x times as fast as time of no-reduction approach, and it is calculated as follows:* Speed Up = *time for no-reduction/time for Delta top 20 percent.*
• † *is appended to the cases of* Delta top 20 *percent approach that can achieve the same amount of improvement as the no-reduction.*

indicator, a more efficient approach takes less time to achieve the same amount of maintainability improvement, or from another perspective, it contributes greater maintainability improvement in same time duration.

The elapsed time for each iteration includes calculating the Delta Tables, checking preconditions, ranking refactoring candidates in the Multi-criteria Delta Table, and calculating fitness functions on the refactored designs that are the trials (simulations) of the application of refactoring candidates. The experiment is performed under the following conditions: processor 2.7 GhHz Intel Core i5, memory 32 G 1333 MHz DDR3, and operating system OS X 10.10.5.

For a comparative study, we perform the refactoring identification process without restricting the candidates. In short, in this approach, all possible Move Method refactoring candidates that have passed the preconditions are assessed through fitness functions; this approach is referred to as the "*no-reduction*" approach.

Results for Time Saving. Table 2 gives an overview of the results total time, speed up, the number of assessed refactoring candidates, and final accomplishments of the fitness functions. To capture subtle changes, the values of the fitness functions are represented in four-digits decimal numbers because a Move Method refactoring is an elementary-level move and it does not change the values of fitness functions in great amounts. In this table, the two-phase assessment approach restricting refactoring candidates to the top 20 percent in the Multi-criteria Delta Table, which we call "**Delta top 20** percent," is shown as representative of our approach because this restriction size is sufficient to find candidates as much as the no-reduction approach. Three types of fitness functions (e.g., MPC, Connectivity, and EPM) and the three types of multi-objective functions for two objectives (metrics) from those fitness functions (e.g., MPC versus Connectivity, EPM versus MPC, and EPM versus Connectivity) are used for three projects (e.g., `Apache Ant`, `JGit`, and `JHotDraw`); thus, 18 cases are compared to our approach and the no-reduction approach. It should be noted that to improve maintainability, fitness functions should be increased or decreased as MPC (−), Connectivity (+), and EPM (−).

The results in Table 2 show that by using our approach, we are able to save a considerable time while achieving the same amount of fitness improvement as the no-reduction approach. For instance, our approach is 2.6 (*min*) to 13.5 (*max*) times faster than the no-reduction approach is. This speed up is calculated by the time of no-reduction approach over the time of our approach. Note that when calculating the speed up, the time for accomplishing the same amount of fitness improvement needs to be compared for a fair analysis. Thus, when final achievement of Delta top 20 percent approach is less than that of the no-reduction approach, the former approach is set to the baseline and the time to accomplish this baseline is compared.

When excluding cases using the multi-objective fitness function, seven out of nine cases can achieve the same amount of improvement as the no-reduction approach by only assessing refactoring candidates ranked to the top 20 percent in the Multi-criteria Delta Table. These corresponding seven cases of Delta top 20 percent approach are denoted by † in Table 2. When assessing the candidates of the top 50 percent, we could achieve the same final fitness values as the no-reduction approach for all cases (see Fig. 10). The effects of the restriction size of refactoring

TABLE 3
Overview of the Results When Using *Multi-Objective Fitness Functions*

Project	Approach		Computational Cost			Achievement in Fitness Function
	Multi-objective Fitness Function **(Objective 1, Objective 2)**	Reduction	Total Time (sec)	Speed Up	Total Candidate (#)	Final Value [Δ amounts]
Apache Ant	(MPC, Connectivity)	Delta top 20%	1,235.46	**n/a**	17,500	(14.0396 [−0.3061], **0.3081**‡ [+0.0135])
		No-reduction	16,424.57		194,512	(**14.0245**‡ [−0.3212], 0.3072 [+0.0126])
	(EPM, MPC)	Delta top 20%	1,789.74	9.6	17,500	(0.9035 [−0.0009], 14.0531 [−0.2926])
		No-reduction	17,225.71		195,376	(0.9035 [−0.0009], 14.0531 [−0.2926])
	(EPM, Connectivity)	Delta top 20%	2,613.81	10.5	17,500	(0.9033 [−0.0011], 0.3097 [+0.0151])
		No-reduction	27,521.06		195,410	(0.9033 [−0.0011], 0.3097 [+0.0151])
JGit	(MPC, Connectivity)	Delta top 20%	672.32	**n/a**	20,200	(**13.4123**‡ [−0.3203], 0.3567 [+0.0283])
		No-reduction	5,920.03		234,242	(13.4175 [−0.3151], **0.3575**‡ [+0.0291])
	(EPM, MPC)	Delta top 20%	1,272.89	9.8	20,200	(0.8969 [−0.0022], 13.4398 [−0.2928])
		No-reduction	12,429.60		235,115	(0.8969 [−0.0022], 13.4398 [−0.2928])
	(EPM, Connectivity)	Delta top 20%	1,568.10	11	20,200	(0.8964 [−0.0027], 0.3641 [+0.0357])
		No-reduction	17,223.65		236,216	(0.8964 [−0.0027], 0.3641 [+0.0357])
JHotDraw	(MPC, Connectivity)	Delta top 20%	328.10	**n/a**	5,400	(**16.9387**‡ [−0.2237], 0.2846 [+0.0134])
		No-reduction	2,150.99		54,091	(16.9547 [−0.2077], **0.2860**‡ [+0.0148])
	(EPM, MPC)	D‘elta top 20%	270.08	**n/a**	4,320	(**0.9203**‡ [−0.0006], 16.9947 [−0.1678])
		No-reduction	2,232.21		54,494	(0.9205 [−0.0004], **16.9760**‡ [−0.1864])
	(EPM, Connectivity)	Delta top 20%	463.71	8.2	4,914	(0.9203 [−0.0006], 0.2831 [+0.0119])
		No-reduction	3,802.73		54,303	(0.9203 [−0.0006], 0.2831 [+0.0119])

• *‡ is appended to the final fitness value which is greater than the one of the opponent's approach for each objective; Speed Up cannot be calculated (denoted as **n/a**) when neither approach could be identified as superior because no solution dominated both objectives; nonetheless, we could still observe that the time taken in our approach is greatly reduced (see Fig. 8).*

candidates in the Multi-criteria Delta Table with the efficiency of computation and final achievement in fitness functions are discussed in detail in Section 5.2.

Multi-Objective Fitness Function. Table 3 represents the results of the two-phase assessment approach when using the multi-objective functions (e.g., MPC versus Connectivity, EPM versus MPC, and EPM versus Connectivity) as a fitness function. When comparing the final values of each objective between our approach and the no-reduction approach and considering the two objectives to have the same relative importance, in four out of nine cases, neither approach could be identified as superior, because no solution (at final achievement) dominated both objectives. For example, for the multi-objective function of MPC and Connectivity, in `Apache Ant`, the no-reduction approach has a better MPC final fitness value, while our approach has a better Connectivity final fitness value. Likewise, in `JGit` and `JHotDraw`, our approach has better MPC final fitness values, while the no-reduction approach has better Connectivity final fitness values, respectively.

Nonetheless, we could still observe that the time taken in our approach is greatly reduced compared to the time required by the no-reduction approach to accomplish the same solutions. The graphs in Fig. 8 show the results of selected refactorings from the two approaches (Delta top 20 percent versus no-reduction) using the multi-objective functions as fitness functions. Note that each refactoring is randomly selected from the Pareto front at each iteration of the identification process, as explained in Section 3.1. Each axis represents one objective (e.g., MPC, Connectivity, or EPM), and the time and the speed up measure are annotated to the points where the two approaches arrive at the same solutions (reaching equal values for both objectives). From these points of the solutions, our approach is 6.7 (*min*) to 13.2 (*max*) times faster than the no-reduction approach. For this reason, when using the multi-objective fitness functions, we conclude that our approach can also be efficiently computed.

Computational Cost for Each Iteration. Table 4 shows the computational cost of the number of assessed candidates and the time for each iteration; the number of assessed candidates and the time required for performing our approach are less than those of the no-reduction approach. The time difference for each iteration is small; thus, it may be seen manageable in a small-scale system or when the load for computing fitness functions is low. However, as the system becomes larger and the fitness functions become more complex, the *accumulated* time differences will be increased drastically. It should be noted that the time it takes to calculate the Delta Table is extremely low [*max*: 1.54 (sec), *min*: 0.67 (sec)], and this is affordable. This overhead enables improvement of the efficiency of our approach by restricting the number of candidates to be assessed using the fitness functions and then achieving the same amount of fitness improvement as the no-reduction approach.

Our approach is efficient in that it saves a considerable amount of time while still achieving the same amount of fitness improvement as the no-reduction approach. Our approach is 2.6 (*min*) to 13.5 (*max*) times faster than the no-reduction approach.

We observed significant results on time savings related to the system size and fitness function complexity. Fig. 9 shows the graphs for the time savings of each subject. The X-axis is the percent of achievement on fitness functions, while the Y-axis is time saving. This *time saving* denotes the *absolute difference of time* between the Delta top 20 percent

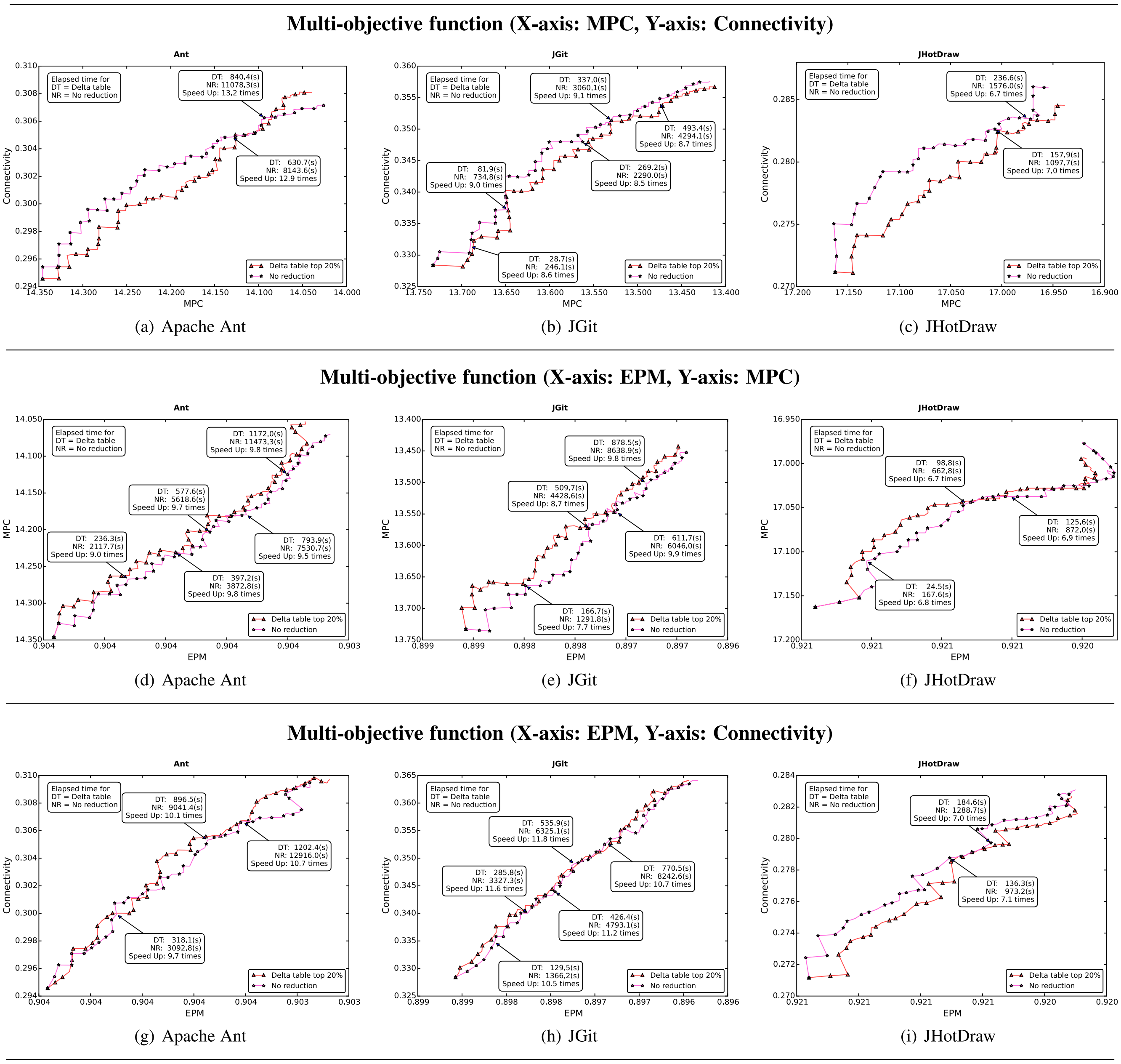


Fig. 8. Graphs of the *multi-objective fitness functions* (for instance, bi-objectives); Each objective at each axis; The time and the speed up (time for no-reduction/time for Delta top 20 percent) are annotated, where the two approaches (Delta top 20 percent versus no-reduction) arrive at the same solutions for three subjects.

and the no-reduction approaches. The baseline of the total achievement on fitness functions is set to *min*{Delta top 20%, no-reduction}, which is the same method used when calculating the speed up in Table 2. The time saving for every iteration is accumulated, and the time savings are represented as the refactoring identification process proceeds. The results of the graphs in Fig. 9 can be summarized below.

Result of the Effect of System Size. The efficiency of the computation increases as the size becomes larger. The system size relates to the number of classes and the number of entities (methods and fields) in the system. The size of the subjects is larger in the following order: `Apache Ant`, `JGit`, and `JHotDraw` (see Table 1). The elapsed time for performing the entire process using each fitness function increased according to this order. The time savings for all used fitness functions in `Apache Ant` are much larger than the time savings in `JHotDraw`.

Result of the Effect of the Fitness Function Complexity. In each subject, we observe that the efficiency of the computation is increased as a used fitness function becomes more complex. For instance, MPC is a rather simple fitness function, while EPM and Connectivity fitness functions are complex fitness functions. In each subject, the time savings for EPM and Connectivity fitness functions are much larger than the time savings for MPC.

The complexity of fitness functions denotes aspects of both design and computation. Design complexity indicates that many types of metrics constitute a fitness function (e.g., the weighted sum of several metrics) or a multi-objective fitness function. Computation complexity signifies that the computational load of fitness functions is high. For instance,

TABLE 4
Computational Cost for Each *Iteration*

Reduction	Cost Per Iteration		`Apache`	`JGit`	`JHotDraw`
No-reduction	Candidate (#)		1,954	2,360	548
	Time (sec)	MPC	16.60	14.21	3.55
		Connectivity	149.31	46.72	18.64
		EPM	159.55	110.77	18.65
Delta top 20%	Candidate (#)		175	202	54
	Time (sec)	MPC	3.22	2.49	1.37
		⊢ Delta Table	1.52	0.98	**0.67***
		⊢ Others	1.70	1.51	0.70
		Connectivity	11.08	5.36	2.95
		⊢ Delta Table	**1.54****	0.99	**0.67***
		⊢ Others	9.54	4.37	2.28
		EPM	16.28	10.77	2.80
		⊢ Delta Table	1.53	0.97	**0.67***
		⊢ Others	14.75	9.8	2.13

• *Candidate (#): Average number of assessed refactoring candidates for each iteration.*
• *Time for Delta Table (sec): Average time for calculating the Multi-criteria Delta Table for each iteration; annotated for* min *time and* max *time with* $*$ *and* $**$.
• *Time for Others (sec): Average time required for other than the Multi-criteria Delta Table (mostly for calculating each fitness function) for each iteration.*

to obtain a system-level metric, metrics for all classes existing in a system may need to be calculated. If a metric for each class requires an examination of all relations surrounding that class, it is hard to reuse the caching objects, which increases the computational overhead.

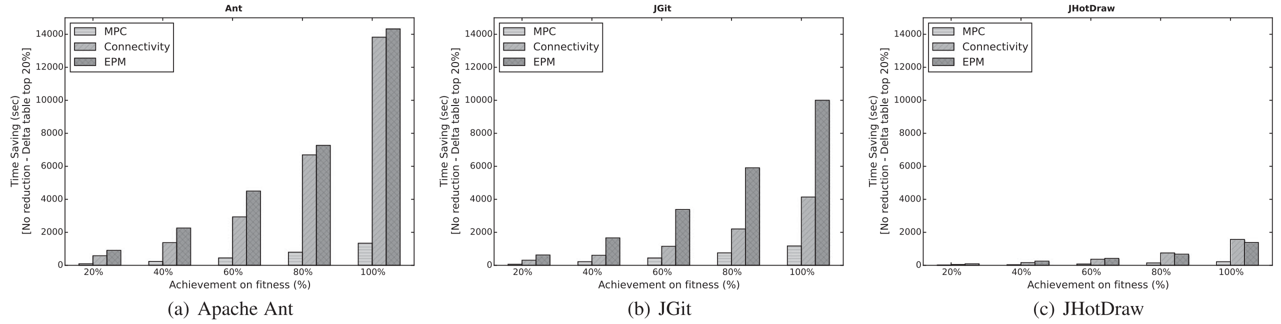
In this context, loads of the computation in EPM and Connectivity are comparably high. As shown in the formulation of fitness functions in Section 4.2, the EPM investigates the relations to the entities in a class and the entities outside the class for every entity in a system; the aim of this is to consider cohesion and coupling aspects. In Connectivity, the number of entities in each class is important, as the metric considers all possible method pairs to investigate their relations if they have method calls or shared fields. Thus, for both EPM and Connectivity, the computation time is increased *linearly* to the number of classes or entities in a system. However, for Connectivity, the computation time is increased *exponentially* to the *number methods in a class*, as Connectivity examines all possible pairs of methods in a class, and therefore the number of entities for each class is critical in determining the computational cost. Contrary to the metrics, MPC is not essentially affected by the size of the system; it simply counts the number the method invocations. In summary, the computational cost, such as the amount of calculation of fitness functions for the application of refactoring candidates, is directly affected by the complexity of the fitness functions used and the system size of the next elements: the number of classes or entities in a system and the number of entities in a class (inner entities). The number of entities in a system is proportional to the number of entities outside a class (outer entities).

From the results using multi-objective fitness functions, we could find the evidence that our two-phase assessment approach becomes more efficient when more complex fitness functions are used. For instance, the time savings between two approaches, no-reduction approach and our approach, when using two objectives for a fitness function (in Table 3) are much larger than the ones compared to the cases using the single objective (in Table 2). In addition, the more complex each objective in a multi-objective fitness function, the more time it takes to perform the entire refactoring process; thus, it is more efficient to use the two-phase assessment. For example, in `Apache Ant`, the time savings are 24,907 (sec) and 15,436 (sec) for the cases of EPM versus Connectivity and EPM versus MPC, respectively. In those multi-objective functions, EPM is commonly used, and the other objectives are MPC and Connectivity, respectively; Connectivity is a more complex fitness function than MPC, which produces the difference in the greater amount of time savings. Thus, we conclude that our approach is more efficient when the more complex fitness function is used.

The amount of time savings is increased as the size becomes larger and as more complex fitness functions are used. The elapsed time for performing the entire process using each fitness function is increased in the following order: `Apache Ant`, `JGit`, and `JHotDraw`, which indicates a larger system size order. Furthermore, more time is required in EPM and Connectivity than MPC, which conforms to the order of higher computational loads of fitness functions.

5.2 RQ2. Restriction Size Effect: Multi-Criteria Delta Table with Top k

To investigate the effect of restricting the size of refactoring candidates in the Multi-criteria Delta Table on computational efficiency, we compared the achievements of fitness functions (e.g., Connectivity, MPC, and EPM) over time for the various sizes (e.g., top $k\%$ [5%, 10%, 20%, 50%] of the Multi-criteria Delta Table).



(a) Apache Ant (b) JGit (c) JHotDraw

Fig. 9. Time savings for each subject: The *X*-axis is the percent of achievement on fitness functions (i.e., maintainability improvement), while the *Y*-axis is time saving.

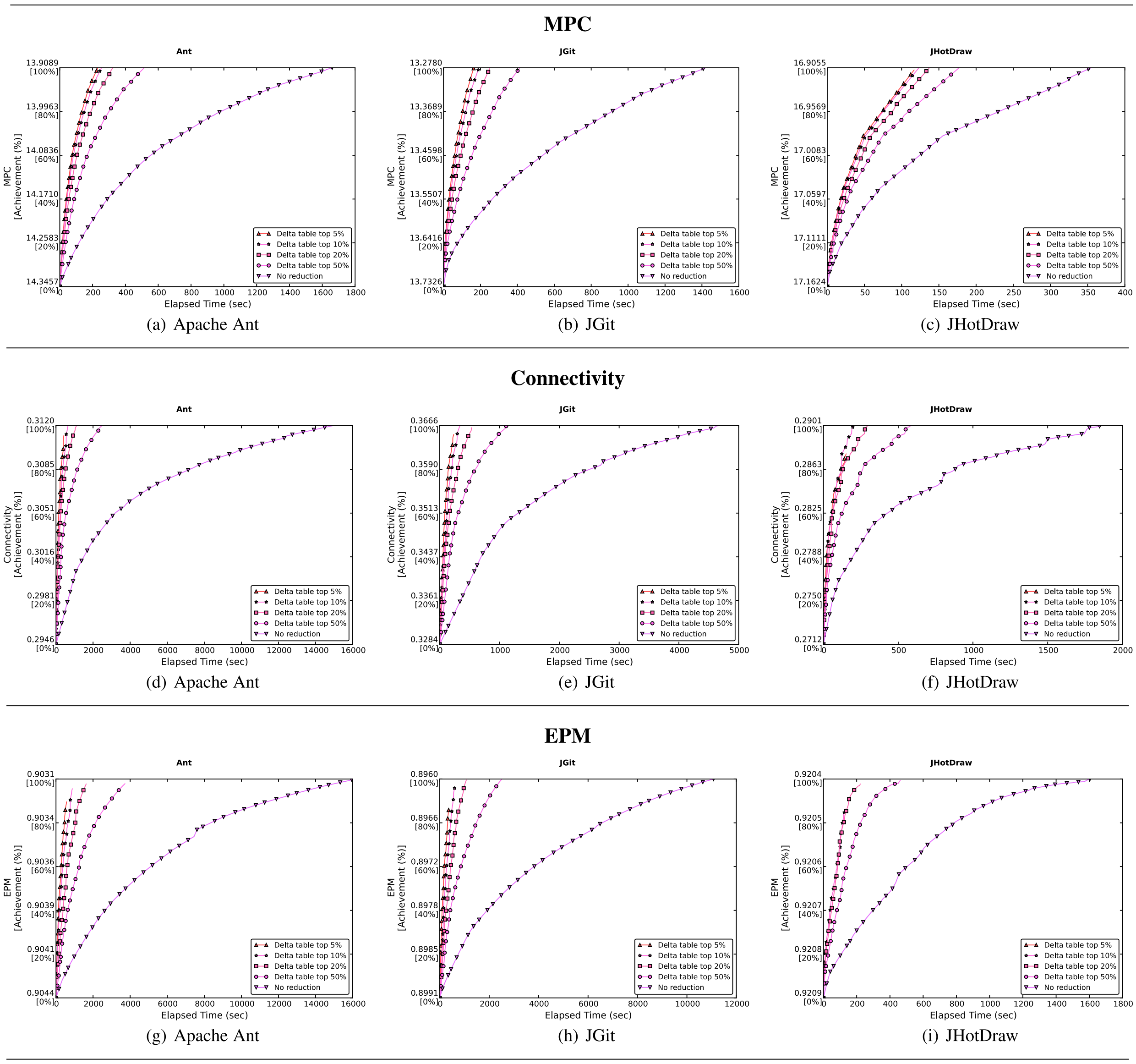


Fig. 10. Effect of restriction size of refactoring candidates in the Multi-criteria Delta Table (top k% [5%, 10%, 20%, 50%]) on efficiency: Achievements of fitness functions (*Y*-axis) over *time* (*X*-axis).

Fig. 10 represents the graphs showing the effects of the size of refactoring candidates in the Multi-criteria Delta Table on efficiency, measured by the achievements of fitness functions over time. The trends of these graphs apparently show that as the restriction size of candidates in the Multi-criteria Delta Table becomes smaller, the maintainability tends to improve more rapidly with respect to time because a smaller number of candidates is assessed using each fitness function. For instance, when restricting refactoring candidates from all to the top 20 percent in the Multi-criteria Delta Table, the time it took is at least 1.7 and up to 6.3 times faster to accomplish the same results for the final achievement in the fitness functions.

As the restriction size of candidates in the Multi-criteria Delta Table becomes smaller, the maintainability tends to improve more rapidly with respect to time. The time it takes to assess the top 20 percent candidates in the Multi-criteria Delta Table is up to 6.3 times faster than assessing all candidates.

The size in the Multi-criteria Delta Table affects the efficiency of computation and final maintainability achievement. If the Delta Table is extremely small (e.g., Delta Table top $k \leq 5\%$), the maintainability improvement achieved could be too small or the process may be terminated early; thus, it may not achieve sufficient improvement compared to the intended goal for fitness functions. In contrast, when the restriction size is larger and more candidates are chosen in the Multi-criteria Delta Table, the chance of a better final achievement relating to fitness functions is increased; however, the efficiency tends to be lower because much more time is required to assess the candidates. Thus, the maintainability of the fitness functions does not improve relative to time invested.

For this reason, it is important to determine the proper size in the Multi-criteria Delta Table by considering both the efficiency and the final achievement. Determining the size of the Multi-criteria Delta Table is worthwhile and critical, especially in situations when time or computing resources are limited.

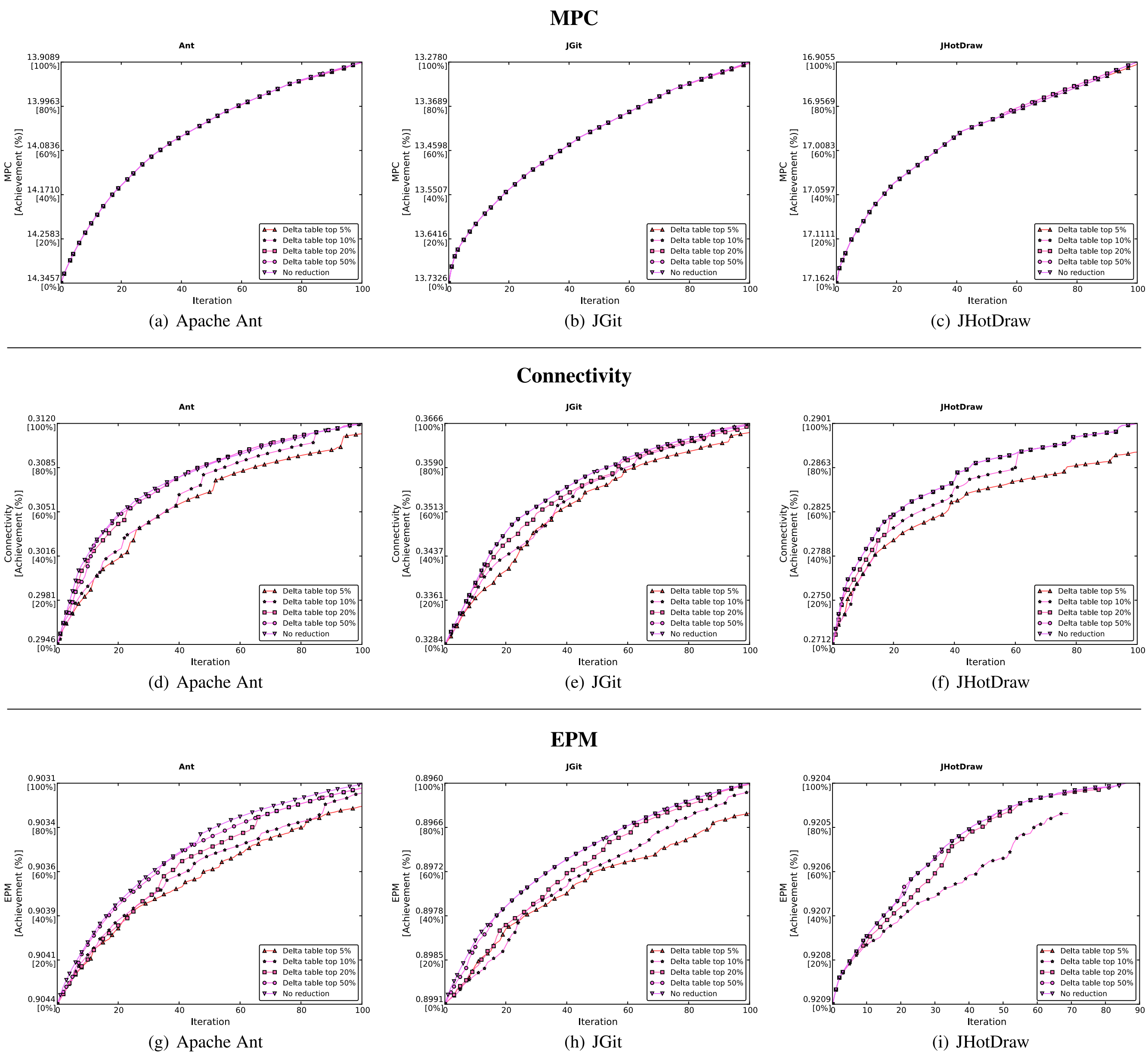


Fig. 11. Effect of restriction size of refactoring candidates in the Multi-criteria Delta Table (top k% [5%, 10%, 20%, 50%]) on efficiency: Achievements of fitness functions (*Y*-axis) over the *number of iterations* (*X*-axis).

A too-small size will cause early process termination; thus, it is important to determine the proper size.

It is important to note that time and number of iterations can be a trade-off for the achievements of fitness functions; thus, when the restriction size of refactoring candidates is reduced, they trade less time for the larger number of iterations. However, the achievement of fitness functions with a smaller number of iterations can sometimes be highly preferred, even if they require more time. Developers may want a smaller number of iterations because they *do not* want to perform the iterative process again and again. If they attempt to perform the refactoring identification process only until the goal of the fitness improvement is achieved, they would prefer to apply a few refactoring steps. Furthermore, the number of iterations relates to the application cost; thus, a smaller number of refactoring applications can be more valued. The number of iterations denotes the number of applied refactorings because a refactoring is selected and applied for each iteration of the process. Basically, if we can ensure that applying a refactoring on actual source codes is fully automated by a tool, then the refactoring cost can be regarded as zero; however, in practice, the application of refactorings may involve additional application costs, such as the effort of relocating codes, especially when the refactorings are complex [50].

Fig. 11 represents the graphs that show the effects of the size in the Multi-criteria Delta Table for the improvement of fitness functions for each of the selected refactoring application. The number of applied refactorings is denoted by the number of iterations. For MPC, a simple metric, the improvements of fitness functions over 100 applied refactorings (i.e., 100 iterations) are identical for all various size in the Multi-criteria Delta Table. In this case, the smallest size of the table

TABLE 5
Prediction Results

Subject	Fitness Function	Precision			Recall		
		Min	Max	Mean	Min	Max	Mean
Apache Ant	MPC	0.47	0.48	0.47	**1**	**1**	**1**
	Connectivity	0.38	0.52	0.47	0.74	0.96	0.85
	EPM	0.23	0.24	0.23	0.79	0.83	0.81
JGit	MPC	0.37	0.37	0.37	**1**	**1**	**1**
	Connectivity	0.46	0.5	0.48	0.93	0.97	0.95
	EPM	0.34	0.35	0.34	0.74	0.76	0.74
JHotDraw	MPC	0.47	0.5	0.48	**1**	**1**	**1**
	Connectivity	0.4	0.43	0.42	0.97	0.99	0.99
	EPM	0.21	0.24	0.23	0.89	0.94	0.91

• Recall 1 *indicates that the Multi-criteria Delta Table predicts 100 percent of all refactoring candidates that have positive effects on a fitness function.*
• *In the experiment, we used the generic type of the Multi-criteria Delta Table, which results in the low precision. The precision can be higher when the Delta Tables are customized to fit into each fitness function.*

is the best choice for time efficiency. In contrast, for Connectivity and EPM, which have higher computation complexity, as the restriction size of the Multi-criteria Delta Table becomes larger, the amounts of improvement on fitness functions tends to become higher in accordance with the number of iterations. Finally, when assessing the top 50 percent of refactoring candidates, the rate of the improvement with respect to the number of iterations becomes similar to that in the no-reduction approach. Thus, the size does not necessarily need to be increased above this. For approaches using fitness functions of Connectivity and EPM, if a higher rate of the improvement on fitness functions over the iterations is preferred over a smaller amount of time (i.e., even if the required time will increase), then the larger size of the Multi-criteria Delta Table should be used. In our approach, we put greater emphasis on using less time.

Less iterations and more required time can be a trade-off, and developers can choose which criterion should be more highly valued according to their preference.

5.3 RQ3. Restriction Performance: Multi-Criteria Delta Table as a Filter

To test the performance of the Multi-criteria Delta Table as a filter for the restriction, we investigated its ability to identify refactoring candidates that actually have higher (better) fitness function values; these represent the real assessment methods for evaluating refactored designs on maintainability. For this, for each iteration of the refactoring identification process, refactoring candidates with negative values on at least one of Delta Tables, which are expected to have the possibilities to improve maintainability, are compared against all possible candidates with positive effects, which are regarded to improve maintainability. The comparison of the possible candidates is obtained after completing the precondition verification.

Two criteria are measured: precision and recall. Let the set of refactoring candidates in the Multi-criteria Delta Table be D and the set of refactoring candidates with positive effects on each fitness function among all the possible candidates be E. Then, the precision and the recall can be defined as follows:

$$\textbf{Precision} = |D \cap E|/|D|, \ \textbf{Recall} = |D \cap E|/|E|.$$

The precision indicates the number of the candidates commonly identified from two set over the number of the ones identified by the Multi-criteria Delta Table. The recall represents the number of commonly identified candidates over the whole set. A higher recall means that the Multi-criteria Delta Table can find more refactoring candidates that actually improve fitness; this means that the Multi-criteria Delta Table plays the role properly in limiting the search space of refactor candidates and increasing the efficiency. The value of 1 in the recall means that the refactoring candidates found by the Multi-criteria Delta Table include all candidates with positive effects on a fitness function.

Table 5 shows the results of the precision and the recall for all fitness functions of three subjects. The results are shown in the *min*, *max*, and *mean* values that are obtained for every iteration of the refactoring identification process. The values of the precision are in the range of 0.21 to 0.52. The reason for the low precision is that we used the same types of Delta Tables for all fitness functions. In contrast, the values of the recall are in the range of 0.74 to 1. In particular, for MPC, the values of the recall are 1 for all subjects, which indicates that the Multi-criteria Delta Table perfectly identified all refactoring candidates that improved MPC. The MPC is the number of invocations by the methods in a class, which is similar to the elements of the Delta Table that considers the dependency relation where one method calls another method. The Multi-criteria Delta Table also produced good results of the recall for other fitness functions, such as Connectivity and EPM. For instance, the *min* values of the recall in Connectivity and EPM are both 0.74; this value represents that more than 74 percent of candidates that have positive effects on Connectivity and EPM can be identified by the Multi-criteria Delta Table.

In the experiment, we used the generic type of the Multi-criteria Delta Table, which results in the low precision. For comparison purposes, the same types of Delta Tables are used for all fitness functions, thus many false-positive refactoring candidates may have been retrieved. In practical use, Delta Tables can be customized to fit into each fitness function, thereby reflecting the goal of the refactorings; then, the precision can be increased. Nevertheless, through the results of the recall, we could observe that the Multi-criteria Delta Table is able to find the refactoring candidates that actually improve fitness with a high probability, which is enough to show the prediction capability of the Delta Tables.

The Multi-criteria Delta Table perfectly identified all refactoring candidates that improved MPC. For Connectivity and EPM, more than 74 percent of candidates that have positive effects on each fitness function can be identified by the Multi-criteria Delta Table.

5.4 Threats to Validity

Scalarization for the Multi-Criteria Delta Table. We adapted the Pareto concept and devised the Multi-criteria Delta Table, which enables consideration of many types of relations. The solutions of refactoring candidates belonging to the Pareto fronts can be regarded as having the same priority. The candidates belonging to the same tier of the group are equally ranked as well (see Fig. 7). Thus, the maintainer can choose a solution from this group depending on his or her preferences for various types of Delta Tables. To automate the process and choosing the top $k\%$ of refactoring candidates based on the Delta Tables, we needed to investigate the partitioned groups in an order of highly ranked tiers. To rank the candidates within the same group, we needed to quantify each refactoring candidate. Thus, the euclidean distance [44] is used to obtain the approximated values of those candidates.

Limiting the Maximum Number of Iterations. The maximum number of iterations for the refactoring process is set to 100 in this experiment. The number of iterations is limited due to computing constraints; however, the number is adequate to observe the trends of the variations in fitness. Moreover, it is impractical to apply over 100 refactorings because of the application cost.

Time Savings on Connectivity. The approach using the Connectivity metric greatly improved the efficiency of the computation (compared to its computation complexity). This can be explained as follows. First, `JHotDraw` and `Apache Ant` tend to have a larger number of entities in each class on average. In addition, the same Delta Tables that are not designed for each fitness function are highly correlated with the Connectivity metric.

Subject Variation. Our evaluation used only three subject programs. To show the validity of the provided method, more experiments need to be performed on various projects (e.g., varying in size and project characteristics).

Comparison of Refactoring Candidates for Different Approaches. To investigate the ability of the Delta Tables to identify refactoring candidates that actually have better fitness function values (for the RQ3), we needed to compare the refactoring candidates produced from the two different approaches, namely our approach and no-reduction approach. Although the initial designs are identical in the two approaches, applying different sequences of refactorings resulted in different refactored designs; thus, the refactoring opportunities cannot be compared directly. For comparison purposes, in each iteration of the refactoring identification process, we followed one of the comparison approaches (i.e., the no-reduction approach) and identified the refactoring candidates using our approach from the same design. Then, these two set of identified refactoring opportunities are compared.

Comparison Approach. In this paper, we do not directly compare our method to those of other Move Method identification studies, such as Methodbook [37] and the EPM [10], since their primary goal is to identify the refactoring candidates by defining a fitness function that serves as a guide to find good solutions (e.g., candidates that improve maintainability). The main contribution of this paper is to improve the computational efficiency via a two-phase assessment in the refactoring identification process. The EPM is used as one of the fitness functions in the second-phase assessment.

Validity of Comparison Criteria. When measuring elapsed time, many factors may have an effect. We regard this as a negligible issue because it causes small variation. Thus, we used the elapsed time for the comparison.

5.5 Discussion

Multi-Objective Fitness Function. Using a multi-objective fitness function does not mean that we use the multi-objective search. In the experiment, various types of fitness functions have been used to assess the impact of the application of refactoring candidates in terms of maintainability; and a multi-objective function (e.g., MPC versus Connectivity) is used as one of the fitness functions. The experimental conditions of the various fitness functions are to show that the more complex fitness functions are used, the more the two-phase assessment approach increases computational efficiency.

It is worth to mention that we do not aim to provide the solutions for the multi-objective optimization. To solve the multi-objective optimization problem, the promising refactoring candidates of the search space (i.e., representative set of Pareto optimal solutions) should be sampled by multiple runs, and the trade-offs in satisfying the different objectives should be analyzed or quantified. However, finding the optimal refactoring sequences is not our main concerns. Our approach accommodates the stepwise interactive search (explained in Section 3.2). To this end, all the candidates in the set of Pareto optimal (Pareto front) are considered equally good, thus it is enough to provide them for each iteration of the process. We believe that the searching method, whether it is the multi-objective or single-objective, does not really affect the main contribution that the proposed method restricting the candidates using the Delta Tables helps to improve the efficiency of the refactoring identification process.

Random Selection for Restricting Refactoring Candidates. To restrict refactoring candidates, random selection can be used. However, the random selection is unstable, especially when the number of selected refactoring candidates is so small. This may generate different solutions for every execution and rely on finding a good solution by chance. Moreover, random selection seems to require no computational cost, but searching all found refactoring candidates to eliminate the redundant ones actually incurs high computational costs. Nonetheless, random selection is useful to escape from the local minimum. Thus, we also considered randomly selected candidates when choosing the top $k\%$ ranked candidates in the Multi-criteria Delta Table (see Section 3.1).

Importance of Using Techniques for Internal Implementation. When developing the proposed methods into source codes, the techniques and decisions are critical and have a great impact on performance, however, this issue is rarely discussed in software engineering research. In the implementation, we applied new techniques, such as Eigen C++ libraries, for efficient matrix computation (see Section 2.2.3), and this significantly improved the performance. Furthermore, since checking the numerous preconditions requires heavy calculation, the preconditions are checked only once when starting the refactoring process. The preconditions are checked for all possible Move Method refactoring candidates, and the mask matrix (row: entity, column: class) is established. The element in the mask matrix becomes 1 when the corresponding refactoring candidate has passed the preconditions successfully; otherwise, the element is set to 0. By element-wise multiplication of the mask matrix with the calculated Delta Tables, only the refactorings that do not violate the conditions remain.

Extension to Other Types of Refactorings. Each element in the Delta Tables has the values for the delta of maintainability for moving methods or fields. As addressed in Section 2.2.2, the large-scale refactoring consists of elementary-level refactorings such as Move Method and Move Field refactorings, thus the Delta Tables can be extended to assess other types of refactorings (e.g., Extract Method or Extract Class refactoring).

Suggestions of Refactoring Candidates to Active Open Source Project. We applied the identified refactorings to the active open source project, `JGit`. The most recent version is 4.7.1. We performed our approach of refactoring identification process automatically by choosing the refactorings that improve the maintainability of a fitness function to the greatest extent. Among the suggested refactorings, we selected to submit two of the refactorings found when using the fitness function of EPM: Move Method `clean` from class `WindowCache` to class `Entry` and Move Method `clean` from class `DfsBlockCache` to class `HashEntry`. The class `Entry` and the class `HashEntry` are the inner classes. The method `clean` tends to access the methods and attributes in each inner class (i.e., Feature Envy design problems); thus, it is better to move the methods to those inner classes where those methods are actually used.

Using the projects used in the experiments, however, it is difficult to find many meaningful suggestions for the following reasons. First, the projects are mature. As the systems evolve, the subjects where the emphasis has been placed on improving the design quality do not have as many candidates to be refactored. It is hard to show the usefulness of the suggested approach using already well-designed versions of software programs. Second, projects with characteristics such as `JGit` may not require refactoring opportunities in terms of design quality. `JGit` is the Git version control system, including repository access routines, network protocols, and core version control algorithms; and it is the algorithm-intensive program, which is procedural, not object-oriented. Therefore, some of the suggested refactorings are inadequate to apply because they can undermine the intent of the original designer and reduce the efficiency of the algorithm. Furthermore, in order to use many libraries and frameworks, the program structure is intentionally designed to reduce the maintainability. In these cases, the suggested refactorings should not be applied.

6 RELATED WORK

To quantify the impact of a refactoring candidate, the refactoring should be applied either actually or virtually on source codes or design models, and then the design quality of maintainability can be measured on for both designs from before and after applying a refactoring candidate. The difference can be used to exhibit the fitness of a refactoring candidate, and this can be employed to guide searching in an automated refactoring identification approach. A larger difference in a positive direction denotes that the application of the refactoring candidate improves maintainability more.

The most time-consuming part of the refactoring identification process is assessing refactoring candidates. Assessing the designs for all trials of the application of refactoring candidates may require high computational costs for calculating the fitness functions. When the size of the software becomes large and the more complex fitness functions are used, the load for computing fitness functions can be extremely high.

To improve the computational efficiency, we used a two-phase assessment approach by restricting refactoring candidates using the Delta Tables. Each element in the Delta Table denotes the effect of the application of refactoring candidates, and approximations of these effects can be calculated rapidly and at once by computing the values for the delta of maintainability based on matrix computation.

We categorize and present the studies related to this theme below.

6.1 Refactoring Candidate Assessment

The maintainability of software design can be measured using several types of design quality metrics or maintainability evaluation functions. These have been used as fitness functions in other refactoring identification research presented below.

Simple Metrics. A single metric, such as a coupling metric (e.g., MPC [13], RFC [45], CBO [45], and CF [46]) or cohesion metric (e.g., Connectivity [14], Method Similarity Cohesion (MSC) [51], and Lack of Cohesion in Methods (LCOM) [45]), is often used [52], [53]. Such metrics are easy to calculate but consider only one aspect of maintainability.

Multiple Objectives into a Single Optimization. To consider multiple aspects simultaneously, multiple metrics are combined using weights into a single objective fitness function, and this type of fitness function is used in many refactoring studies. Examples include weighted sum of OO metrics [30] and QMOOD [5], [54]. While such a method is useful, problems may occur when weight coefficients are inappropriately chosen or component metrics have dependencies among them [15]. Several metrics (multiple objectives) must be calculated for a fitness function; thus, as the system becomes larger, the computational cost becomes more substantial as well.

Multi-Objective Optimization. A multi-objective optimization is a process of finding a solution from a pool of candidate solutions in which several non-dominant solutions involve different trade-offs. The set of all points in the objective space that are not dominated by any other points is called the Pareto front, and Pareto optimization techniques, such as NSGA-II and SPEA2, have been suggested [15], [16], [17], [18], [20].

Harman and Tratt [15] first suggest using Pareto optimality combining the two metrics, namely CBO and Standard Deviation of Methods Per Class (SDMPC), in search-based refactoring. Multiple runs of their search-based refactoring system lead to the production of a Pareto front, the values of which represent the Pareto optimal sequence of refactorings. Users can choose a value on the front that represents the trade-off between metrics most appropriate to them.

Mkaouer et al. [17] used NSGA-II to find an optimal solution among three objectives, minimizing the number of code smells and improving QMOOD quality metrics, minimizing the size of the refactoring solutions, and maximizing the preservation of the semantic coherence of the design. However, as they admitted, it is difficult to determine the best solution from a set of non-dominated solutions when adapting multiple objectives to software engineering problems.

Our Method. In this paper, we used the same kinds of fitness functions as explained above. The lightweight and rapid delta assessment method, Delta Tables, is used to preliminarily assess the effects of the application of refactoring candidates to filter out the candidates that are unlikely or rarely improve maintainability. Each element in the Delta Table denotes the value for the delta of maintainability, and the elements can be calculated quickly and at once based on matrix computation. It should be noted that other studies utilizing the matrix for refactoring identification use the matrix to store the values (e.g., the likelihood that two methods should be in the same class [55]). In contrast, in our approach, the design of source codes is encoded into matrices, and the matrix multiplication is used to directly calculating the value for the delta of maintainability, an approximate measure used for assessing each refactoring candidate. It is worth to note that using a multi-objective fitness function does not mean to use the multi-objective search. Our approach accommodates the stepwise interactive search, and the searching method is different from the one adopted in the studies [15], [17].

6.2 Efficiency Improvement for the Refactoring Identification Process

Automating the refactoring identification process has been actively studied in search-based engineering research [56], [57], [58]. However, few studies have focused on the efficiency aspects of the refactoring identification process for identifying refactorings that improve maintainability.

Search Space Reduction. Piveta et al. [59] used Deterministic Finite Automata (DFA) to represent refactoring sequences and a set of simplification rules to reduce the search space. They tried to eliminate redundant and meaningless sequences on the generated refactoring sequences by reducing the refactoring sequences to those that are semantically sound while avoiding sequences leading to the same results. To apply their technique, refactoring sequences must first be generated which is usually a time-consuming process. Our technique, however, provides a proactive and efficient approach in that search-space reduction is achieved while searching for refactoring candidates.

Approximation. There have been attempts to use approximate metrics to cheaply assess a fitness to guide a search-based approach.

Harman et al. [60] addressed the need for new forms of approximate metrics for adaptive search-based software engineering problems. They claimed that those metrics can act as surrogates for more computationally expensive measurements and should retain some of the essence of the more computationally expensive metric but sacrifice some degree of precision for computational performance. The surrogate can thus be used to cheaply assess an approximate fitness to guide a search-based approach for dynamic adaptivity.

In [26], we used the preliminary version of the Delta Table as a fitness function. We focused on the ability of the Delta Table to rapidly calculating the effects of the application of refactoring candidates, thereby helping to identify refactorings faster than the approach of using complex fitness functions.

Our Method. In this paper, for the two-phase assessment approach, we used the new version of the Delta Tables, the Multi-criteria Delta Table, which provides ways to consider several types of dependencies, as well as to adapt various types of refactorings other than Move Method refactorings. This serves to reduce the number of refactoring candidates to be evaluated with a fitness function to increase the computational efficiency.

7 Conclusion and Future Work

To automating the refactoring identification process, an efficient method for assessing the impact of the application of the numerous refactoring candidates is essential. To improve the efficiency of the process, we propose a two-phase assessment approach to reduce search spaces of refactoring candidates that are the subjects of evaluation using fitness functions. Refactoring candidates are preliminarily assessed using Delta Tables, and using the Multi-criteria Delta Table, only the candidates that are more likely to improve maintainability are chosen to be evaluated with a fitness function. A refactoring is selected—either interactively by accepting a developers feedback or automatically by choosing the refactoring that improves the maintainability of a fitness function to the greatest extent—from each iteration of the refactoring identification process and the refactorings are applied one after another in a stepwise fashion. The Delta Table plays the most important role in the two-phase assessment approach. Using this method, the delta value of the maintainability provided by each refactoring candidate can be quickly calculated at once by mapping the software design to a matrix and multiplying the matrices. Our approach is evaluated using three large open-source projects. The experiments revealed that our approach is significantly efficient because it saves a considerable amount of time while achieving the same amount of fitness improvement as the no-reduction approach. These time savings increase as the size becomes larger and more complex fitness functions are used. The benefit of saving the computational cost outweighs the overhead of computing the Delta Table.

For future work, we plan to take into account several kinds dependencies and different factors, such as the change history and textual information, for the Multi-criteria Delta Table. We believe that this would improve the applicability of our approach for practical use. Furthermore, we plan to apply global search techniques (e.g., GA) to the automated identification and perform comparative research with the local search techniques in perspectives of maintainability improvement, defect reduction, time taken, and so on. We also have a plan to perform empirical studies on the efficiency in search-based refactoring identification using several approximate evaluation methods. Since the elements in the Delta Tables are approximate but good enough to preliminarily assess the impacts of the application of refactoring candidates so that they can help significantly increase computing efficiency in various domains.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054098). We also gratefully acknowledge Yoo Shin and Ingwon Song for the valuable comments when revising this paper.

References

[1] J. Kerievsky, *Refactoring to Patterns*. Reading, MA, USA: Addison Wesley Pearson Education, 2005.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. San Mateo, CA, USA: Morgan Kaufmann, 2002.

[3] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A novel approach to optimize clone refactoring activity," in *Proc. 8th Annu. Conf. Genetic Evol, Comput.*, 2006, pp. 1885–1892.

[4] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 20, no. 6, pp. 435–461, 2008.

[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. Practice Exper.*, vol. 41, no. 5, pp. 521–550, 2011.

[6] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, 2011, pp. 266–269.

[7] G. P. Krishnan and N. Tsantalis, "Refactoring clones: An optimization problem," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 360–363.

[8] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, Nov. 2015.

[9] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[10] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May/Jun. 2009.

[11] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell (NIER track)," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 820–823.

[12] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *J. Syst. Softw.*, vol. 83, no. 3, pp. 391–404, 2010.

[13] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, 1993.

[14] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Softw. Eng.*, vol. 3, no. 1, pp. 65–117, 1998.

[15] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1106–1113.

[16] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 347–356.

[17] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 331–336.

[18] M. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: A many-objective search-based software engineering approach," *Empirical Softw. Eng.*, vol. 21, no. 6, pp. 2503–2545, 2016.

[19] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: A case study in software product lines," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 492–501.

[20] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: Finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III," in *Proc. Conf. Genetic Evol. Comput.*, 2014, pp. 1263–1270.

[21] T. Aittokoski and K. Miettinen, "Efficient evolutionary method to approximate the Pareto optimal set in multiobjective optimization," in *Proc. Int. Conf. Eng. Optimization*, 2008, pp. 841–858.

[22] Eigen, "Eigen," 2016. [Online]. Available: http://eigen.tuxfamily.org/

[23] Apache Ant, "Apache Ant," 2015. [Online]. Available: http://ant.apache.org/

[24] JGit, "JGit," 2015. [Online]. Available: http://eclipse.org/jgit/

[25] JHotDraw, "JHotDraw," 2011. [Online]. Available: http://www.jhotdraw.org/

[26] A. Han and D. Bae, "An efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation," in *Proc. 21st Asia-Pacific Softw. Eng. Conf.*, 2014, pp. 430–437.

[27] A. Han, D. Bae, and S. Cha, "An efficient approach to identify multiple and independent move method refactoring candidates," *Inf. Softw. Technol.*, vol. 59, pp. 53–66, 2015.

[28] J. Al Dallal and L. C. Briand, "A precise method-method interaction-based cohesion metric for object-oriented classes," *ACM Trans. Softw. Eng. Methodology*, vol. 21, no. 2, 2012, Art. no. 8.

[29] SciPy, "SciPy," 2015. [Online]. Available: http://scipy.org/

[30] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. 8th Annu. Conf. Genetic Evol. Comput.*, 2006, Art. no. 1916.

[31] H. Liu, G. Li, Z. Ma, and W. Shao, "Conflict-aware schedule of software refactorings," *IET Softw.*, vol. 2, no. 5, pp. 446–460, Oct. 2008.

[32] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 220–235, Jan./Feb. 2012.

[33] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of prioritised code clone refactoring," *IET Softw.*, vol. 7, no. 3, pp. 167–186, Jun. 2013.

[34] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of type-checking bad smells," in *Proc. 12th Eur. Conf. Softw. Maintenance Reengineering*, 2008, pp. 329–331.

[35] N. Tsantalis, et al., "JDeodorant," 2014. [Online]. Available: https://users.encs.concordia.ca/ nikolaos/jdeodorant/

[36] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, pp. 2241–2260, 2012.

[37] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, Jul. 2014.

[38] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring for software maintenance," *J. Syst. Softw.*, vol. 81, no. 4, pp. 502–516, 2008.

[39] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: An empirical study," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[40] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[41] Y. Davidor, "Epistasis variance: A viewpoint on GA-hardness," *Found. Genetic Algorithms*, vol. 1, pp. 23–35, 1991.

[42] A. Eiben, P.-E. Raué, and Z. Ruttkay, "Solving constraint satisfaction problems using genetic algorithms," in *Proc. 1st IEEE Conf. Evol. Comput. IEEE World Congr. Comput. Intell.*, 1994, pp. 542–547.

[43] A. Santiago, et al., "A survey of decomposition methods for multi-objective optimization," in *Recent Advances on Hybrid Approaches for Designing Intelligent Systems*. Berlin, Germany: Springer, 2014, pp. 453–465.

[44] M. L. Rizzo, "New goodness-of-fit tests for Pareto distributions," *ASTIN Bulletin-Actuarial Studies Non Life Insurance*, vol. 39, no. 2, 2009, Art. no. 691.

[45] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[46] F. Abreu, "The MOOD metric set," in *Proc. Eur. Conf. Object Oriented Program. Workshop Metrics*, 1995, Art. no. 267.

[47] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[48] M. O'Keeffe and M. O. Cinnéide, "Automated design improvement by example," *Frontiers Artif. Intell. Appl.*, vol. 161, 2007, Art. no. 315.

[49] S. Chidamber, C. Kemerer, and C. MIT, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[50] A. Han and D. Bae, "Dynamic profiling-based approach to identifying cost-effective refactorings," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 966–985, 2013.

[51] C. Bonja and E. Kidanmariam, "Metrics for class cohesion and similarity between methods," in *Proc. 44th Annu. Southeast Regional Conf.*, 2006, pp. 91–95.

[52] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," in *Proc. Eur. Conf. Softw. Maintenance Reengineering*, 2003, pp. 183–192.

[53] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *Proc. 11th Working Conf. Reverse Eng.*, 2004, pp. 144–151.

[54] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. 11th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2011, pp. 105–114.

[55] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer, 2014, pp. 387–419.

[56] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 155–164.

[57] K. C. Cheng and R. H. Yap, "Search space reduction for constraint optimization problems," in *Principles and Practice of Constraint Programming*. Berlin, Germany: Springer, 2008, pp. 635–639.

[58] A. Sakti, Y.-G. Guéhéneuc, and G. Pesant, "Boosting search based testing by using constraint based testing," in *Search Based Software Engineering*. Berlin, Germany: Springer, 2012, pp. 213–227.

[59] E. Piveta, J. Araújo, M. Pimenta, A. Moreira, P. Guerreiro, and R. Price, "Searching for opportunities of refactoring sequences: Reducing the search space," in *Proc. 32nd Annu. IEEE Int. Comput. Softw. Appl*, 2008, pp. 319–326.

[60] M. Harman, J. Clark, and M. Ó. Cinnéide, "Dynamic adaptive search based software engineering needs fast approximate metrics (keynote)," in *Proc. 4th Int. Workshop Emerging Trends Softw. Metrics*, 2013, pp. 1–6.

Ah-Rim Han received the PhD degree from the Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), in 2013. She is a research professor in the Department of Computer Science and Engineering, Korea University. Her research interests include software design quality improvement and software refactoring process automation.

Sungdeok Cha received the PhD degree in information and computer science from the University of California, Irvine. He is a professor in the Department of Computer Science and Engineering, Korea University. His research interests include software safety and computer security. He is a member of the IEEE.