박사 학위논문

Ph. D. Dissertation

객체지향 소프트웨어의 유지보수성 향상을 위한 리팩토링 식별 및 선택 방법에 대한 연구

Identification and Selection of Refactorings for Improving Maintainability of Object-Oriented Software

한 아 림 (韓 雅 琳 Han, Ah-Rim)

전산학과

Department of Computer Science

KAIST

2013

객체지향 소프트웨어의 유지보수성 향상을 위한 리팩토링 식별 및 선택 방법에 대한 연구

Identification and Selection of Refactorings
for Improving Maintainability of Object-Oriented Software

Identification and Selection of Refactorings for Improving Maintainability of Object-Oriented Software

Advisor : Professor Bae, Doo-Hwan

by

Han, Ah-Rim
Department of Computer Science
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science . The study was conducted in accordance with Code of Research Ethics[1].

2013. 5. 8.

Approved by

Professor Bae, Doo-Hwan

[Advisor]

[illegible]

[1]Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.

객체지향 소프트웨어의 유지보수성 향상을 위한 리팩토링 식별 및 선택 방법에 대한 연구

한 아 림

위 논문은 한국과학기술원 박사학위논문으로
학위논문심사위원회에서 심사 통과하였음.

2013년 5월 8일

심사위원장	배 두 환	(인)
심사위원	백 종 문	(인)
심사위원	맹 승 렬	(인)
심사위원	홍 장 의	(인)
심사위원	허 재 혁	(인)

DCS
20075196

한 아 림. Han, Ah-Rim. Identification and Selection of Refactorings for Improving Maintainability of Object-Oriented Software . 객체지향 소프트웨어의 유지보수성 향상을 위한 리팩토링 식별 및 선택 방법에 대한 연구. Department of Computer Science . 2013. 69p. Advisor Prof. Bae, Doo-Hwan. Text in English.

ABSTRACT

Object-oriented software undergoes continuous changes with various maintenance activities. Due to the changes, the design quality of the software degrades overtime. Thus, refactoring can serve to restructure the design of object-oriented software without altering its external behavior to improve maintainability, which in turn reduces maintenance costs and shortens time-to-market.

In the thesis, we provide the methods for supporting systematic refactoring identification: refactoring candidate identification and refactoring selection. For identifying refactoring candidates, we use top-down and bottom-up approaches. First, for the top-down approach—a traditional way of finding refactoring opportunities by using heuristic rules for eliminating violations of design principles in object-oriented software systems—, we establish the rules to identity the refactoring candidates with the aim of reducing dependencies of entities for methods and classes. When establishing the rules, we are motivated by the studies that dynamic information—how the system is utilized—is an important factor for estimating changes. Therefore, to perform refactorings that effectively improve maintainability, the entities are found based on how the users utilize the software (e.g., user scenario and operational profile); and within these entities, refactoring candidates are identified. Second, for the bottom-up approach—the way of finding refactoring opportunities without pre-defined patterns or rules—, we develop the method for grouping entities—methods and attributes—by using the concept of the maximal independent set in graph theory. When grouping entities, we take into account the new dependency of refactorings—refactoring effect dependency on maintainability—as well as the syntactic dependency of refactorings. The entities involved in each maximal independent set are mapped into a group of elementary refactorings and these refactorings can be applied at the same time. For selecting refactorings to be applied, we provide the method of selecting of multiple refactorings by supporting assessment and impact analysis of elementary refactorings. We develop the refactoring effect evaluation framework for assessing each elementary refactoring effect on maintainability based on matrix computation. Using the evaluation framework, we select the group of refactorings containing the multiple elementary refactorings that best improves maintainability.

We evaluate our proposed approach in three open-source projects—jEdit, Columba, and JGIT. From the experimental results, we conclude that dynamic information is helpful in identifying refactorings that efficiently improve maintainability, because dynamic information is helpful for extracting refactoring candidates in frequently changed classes. Furthermore, the refactoring identification using multiple refactorings selects refactorings that lead the software design to reach higher fitness function values (better improve maintainability) with smaller costs (i.e., smaller search space exploration cost and shorter time). In addition, the refactoring effect dependency is essential to be considered for correctly selecting a group of refactorings that most improve maintainability.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Object-oriented software undergoes continuous changes with various maintenance activities such as addition of new functionalities, correction of bugs, improvement of performance, and adaptation to new environments. Several empirical studies [62, 67, 90] have shown that the cost for maintenance and rework is biggest portion of the cost of total software life cycle. In [91], they conducted the case study by analyzing the cost of software development cost collected from three major European companies. Figure 1.1 shows the cost distribution across the software life cycle based on a projected total production time of five years. Only those application systems for which complete cost data was collected were included. For a total production time of five years, the percentage of non-recurring costs amounts on average to 21% of the total life cycle costs. Therefore, 79% of the total costs are recurring costs (i.e., maintenance and rework costs). This implies that to reduce the cost of software development, it is important to reduce the maintenance cost.

The main factor affecting the maintenance cost is the design quality of software [63, 75, 31]. However, since the changes often take place without consideration of the software's overall structure and design rationale due to time constraints, the design quality of the software may degrade overtime. This phenomenon is known as *software aging* [72] or *software decay* [29]. Thus, refactoring can serve to restructure the design of object-oriented software without altering its external behavior [29] to improve maintainability (or to make software accommodate changes more easily), which in turn reduces maintenance costs and shortens time-to-market.

Much of the existing research on automated refactoring focuses on *refactoring application* [78, 25, 80, 51, 9], that is, applying refactorings on actual source codes. Several studies have attempted to support refactoring identification. For instance, to support each activity of the refactoring process, (1) algorithms are developed to find refactoring candidates with the opportunities of applying design patterns [55, 21, 50], removing code clones [56, 45, 93, 46], and improving code quality such as testability [42], as well as maintainability. (2) For evaluating the design of the refactored code, design quality evaluation models such as QMOOD [56] and Maintainability Index (MI) [69], or a special metric such as historical volatility [89], are used. Distance measures [83, 88] or weighted sums of metrics [82] also have been used as evaluation functions (i.e., fitness functions); pareto optimality has been used to compare different fitness functions and to combine results from different fitness functions [43]. (3) The methods for scheduling of refactoring candidates also have been studied [58, 64] to achieve the greatest effect of maintainability improvement. In an attempt to provide the method for automated refactoring, the literature has proposed methods of refactoring identification by using several search techniques. O'Keeffe et al. [68] treat object-oriented design as an optimization problem and employ several search techniques such as multiple ascent hill-climbing, simulated annealing, and genetic algorithms to automate the refactoring process. They do not provide where to apply which refactorings because extraction of refactoring candidates depends on random choices. In the other aspect of refactoring automation, Steimann et al. [85] propose a concept of a framework for

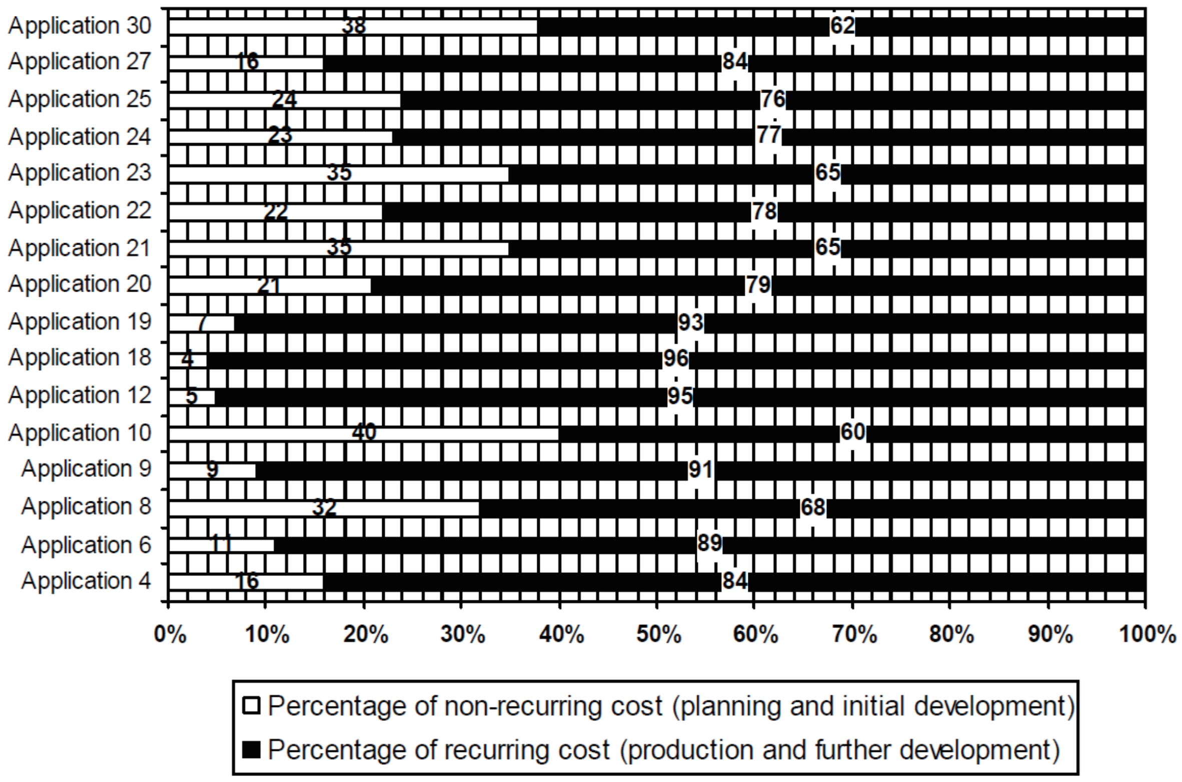


Figure 1.1: Cost distribution in software lifecycle [91].

specifying refactorings in an ad-hoc fashion. They argue that in practice concrete refactoring needs may deviate from what has been distilled as a named refactoring, and mapping these needs to a series of such refactorings is worth to be developed. However, we still lack systematic approaches, clear guides, and automated tool support for identifying where to apply which refactorings and in what order.

Goal and Approach

In the thesis, we provide the methods for supporting systematic refactoring identification: identification of refactoring candidates and selection of refactorings to be applied. For each iteration of the refactoring identification process, multiple elementary refactorings that most improve maintainability are produced. The procedure of refactoring identification process is iterated until no more group of refactoring candidates for improving maintainability are found.

For identification of refactoring candidates, we attempt top-down and bottom-up approaches. First, for the top-down approach—traditional way of finding refactoring opportunities by using heuristic rules for eliminating violations of design principles (e.g., removing bad smells) in object-oriented software systems—we establish the rules to extract the refactoring candidates with the aim of reducing dependencies of entities of methods and classes. When establishing the rules, we are motivated by our previous studies [39] (see in Appendix A.) that have shown that the data capturing how the system is utilized is an important factor for estimating changes; (1) program usage data recorded from Integrated Development Environments (IDEs) significantly improves the accuracy of change prediction approaches [77, 76] and (2) dynamic coupling measures [3] and behavioral dependency measures [39]—that are obtainable during run-time execution and pinpoint the systems' parts that are often used—are good indicators for predicting change-prone classes. As a result, we have come to argue that if changes are *more* prone to occur in the pieces of codes that users *more* often utilize, then investing efforts on the refactorings involving such codes may effectively improve maintainability. Therefore, entities are identified based on how the users utilize the software (e.g., user scenario and operational profile [66]); and within these entities, refactoring candidates are identified. Second, for the bottom-up approach—identification of refactoring opportunities without humans' insights [85]—we develop the method for grouping entities (i.e., methods and attributes) into maximal independent sets (MISs). Different from the previous approaches for refactoring candidate identification, we attempt to grouping elementary refactorings (i.e., Move Method refactorings) without pre-defined patterns. The methods involved in each MIS are transformed into a group of elementary refactorings—in this thesis, Move Method refactorings. Each of the group has elementary refactorings that can be applied at the same time. The concept of the MIS is from graph theory. We select the group of refactorings containing the multiple elementary refactorings that best improves maintainability. When calculating MISs, we take into account the new dependency of refactorings—called refactoring effect dependency (RED). The RED is essential to be considered when selecting refactorings even though refactorings are *not syntactically* dependent—syntactic dependency of refactorings indicates that the application of one refactoring changes or deletes elements necessary for the other refactorings thus it disables those refactorings.

For selecting refactorings to be applied, we provide the method of selecting refactorings by supporting assessment and impact analysis of *elementary* refactorings. We have developed the refactoring effect evaluation framework (i.e., delta table. Each cell of the delta table indicates delta of maintainability after the application of each elementary refactoring on the current design configuration. This delta table is used for refactoring selection criteria. The matrix computation is used for calculating each of elementary refactoring's effect on maintainability. The matrix computation is fast, thus it provides efficient computation for deriving the delta table. It is important that the method of elementary-level refactoring computation enables selecting multiple refactorings. Note that the previous studies of exhaustive and greedy way of refactoring selection suffer from the need of too much space exploration cost due to many possible sequences to be evaluated and the inefficiency in selecting just one best refactoring for the iteration of refactoring identification process, respectively. Furthermore, this method supports to extend considering refactorings to other various type of refactorings; because the action of big refactoring (e.g., Collapse Hierarchy Class refactoring) comprises of elementary refactorings (e.g., Move Method refactorings). The procedure of refactoring selection consists of the several activities: (1) calculation for each of elementary refactoring's effect on maintainability, (2) checking whether there are duplicated elementary refactorings (i.e., syntactic dependencies) or the RED among refactorings—that are identified as refactoring candidates from top-down and bottom-up approaches—, (3) finding multiple (elementary) refactorings—that can be applied at a same time—containing refactorings that most improve maintainability, and (4) identification of the impacted refactorings after applying selected refactorings and recalculation of the changed values for those impacted refactorings.

We evaluate our proposed approach in three open-source projects—jEdit [49], Columba [20], and JGIT [52]. From the experimental results, we conclude that dynamic information is helpful in identifying refactorings that efficiently improve maintainability; because dynamic information is helpful for extracting refactoring candidates in frequently changed classes. Therefore, considering dynamic information in addition to static information provides more opportunities in identifying refactorings that efficiently improve maintainability because of the refactoring candidates that are uniquely identified by the approach using dynamic information only. Furthermore, the experimental results show that the refactoring identification using multiple refactorings selects refactorings that lead the software design to reach higher fitness function values (better improve maintainability) with smaller costs (i.e., smaller search space exploration cost and shorter time). In addition, the RED should be considered when selecting multiple refactorings.

Organization

The thesis is organized as follows. Chapter 2 contains a discussion of related studies. Chapter 3 explains the framework for systematic refactoring identification and the overview of our proposed approach for refactoring identification. Chapter 4 explains the methods of identification of refactoring candidates in terms of the top-down and bottom-up approaches. Then, Chapter 5 explains the detailed procedure and the method of selection of refactorings to be applied. Chapter 6 covers the implemented tool for applying our proposed approach. In Chapter 7 and Chapter 8, we present the experiments performed to evaluate the proposed approach and discuss the obtained results, respectively. Finally, we conclude and discuss future research in Chapter 9.

Chapter 2. Related work

In the following sections, we present representative studies of refactoring identification; note that we divide the categories of related work in the aspect of the comparison with our approach.

2.1 Refactoring Candidate Identification Using Static Metric Based Heuristic Rules

Several studies have attempted to support for identifying refactorings using metrics as a means of detecting refactoring candidates or evaluating refactoring effects [24, 83, 54, 86, 92]. Tahvildari et al. [86] propose a metric-based method for detecting design flaws and analyzing the impact of the chosen meta-pattern transformations for improving maintainability. They detect design flaws based on pre-defined quality design heuristics using object-oriented metrics of complexity, coupling, and cohesion metrics. However, the authors do not provide a systematic approach for applying given meta-pattern transformations; they offer neither clear rules for detecting design flaws nor a method of how to apply meta-pattern transformations. This process still requires much human interpretation and judgment. Moreover, the effect of certain given meta-pattern transformations are evaluated on object-oriented metrics as positive and negative. Since a quantitative method for evaluating the effect of meta-pattern transformations is not available, the approach cannot determine a sequence to be applied first among the multiple potential meta-pattern transformations. Bart Du Bois et al. [24] provide a table representing the analysis of the impact of certain refactorings, which redistribute responsibilities either within the class or between classes, on cohesion and coupling metrics. In the manner of Tahvildari's work [86], the authors specify the impact of refactorings as ranges of best to worst cases as positive (i.e., improvement), negative (i.e., deterioration), and zero (i.e., neutral); it also lacks a means of quantitative refactoring-effect evaluation, which is essential for making a decision on which refactorings should be applied first. Simon et al. [83] provide a software-visualization approach using a distance-based cohesion metric to support developers for choosing appropriate refactorings; the parts with lower distances are cohesive whereas parts with higher distances are less cohesive. However, decisions for which refactorings should be performed and how to apply those refactorings are still heavily dependent on developers, as the authors admit that they presume that the developer is the last authority in identifying and applying refactorings. In the above-mentioned studies, the metrics are obtained using statically profiled information from source codes, in other words, without executing a program, which might suggest refactorings on parts of software that is not really in use. Furthermore, as pointed out above, they provide neither exact algorithms guiding where to apply which refactorings nor a quantitative evaluation method, which are essential for selecting better refactoring.

Research has looked at providing a tool support and systematic methodology to assist developers in making decisions as to where to apply which refactoring. Tsantalis et al. [88] propose a methodology and constructed a tool for the identification of Move Method refactoring opportunities that solve *Feature Envy* bad smells. They extract a list of behavior-preserving refactorings based on a distance-based measure that employs the notion of distance between system entities (i.e., methods and attributes) and classes. This concept of distance for measuring lack-of-cohesion is also used in [83]. The authors also defined an *Entity Placement* metric, also based on the concept of distance and used as a means of quantitative refactoring-effect evaluation. However, in their experiment, they show the performance of refactoring opportunities by measuring the effect of refactored designs only on coupling and cohesion metrics and some qualitative analysis.

2.2 Determining Refactoring Sequences to be Applied

Selection Strategy: Exhaustive or Greedy

It is difficult to schedule refactorings by considering the issues of refactoring dependencies or newly created refactoring candidates. Theoretically, when the number of available refactoring candidates is m and the number of the selecting refactorings is n and assuming that there are no repetitions of refactoring candidates, the number of the refactoring schedules that need to be examined is n-permutations of m that can be formulated by $m!/(m-n)!$. As the number of refactoring candidates increases, the number of possible refactoring schedules increases exponentially. Therefore, scheduling refactorings by investigating all possible orders may become NP-hard.

For selecting refactoring candidates, in the studies [88, 28, 38], they the single refactoring that best improves the current design of software is selected in a *stepwise* (i.e., *greedy*) way after extracting and assessing refactoring candidates. Then, the selected refactoring is applied, and the refactoring identification process is iterated until no more refactorings that can improve maintainability are found. Finally, the sequence of refactorings is generated by logging the results of the selected refactoring for each refactoring identification process. This selection method offers the advantage of taking the change of system; thus, newly created refactoring candidates can be considered. In addition, by extracting and assessing refactoring candidates again after applying the selected refactoring, refactoring dependencies do not need to be considered. However, it is inefficient to select just the single best refactoring for each refactoring identification process. The costs (e.g., search space exploration cost and computation cost) of extracting and assessing refactoring candidates are high.

To address the limitation, we provide an automated method for selecting multiple refactorings that can be applied at the same time.

Search Space Reduction

The methods of narrowing the refactoring sequences to those that are semantically sound and avoiding sequences leading to the same results have been also studied. Piveta et al. [73] propose an approach to narrow the number of refactoring sequences by discarding those that semantically does not make sense and avoiding those that lead to the same results. They also provide a detailed example of the approach considering sequences for method manipulation, showing how the number of sequences can be significantly reduced.

Refactoring Scheduling: Finding an Optimal Sequence of Refactoring Applications using Search Techniques

In studies of search-based refactoring, they try to find an optimal sequence of refactoring applications using search techniques. Lee et al. [56] and Seng et al. [82] use genetic algorithms to produce a sequence of refactorings to apply to reach an optimal system in terms of the employed fitness function. However, [82] do not take into account that the application of a refactoring may create new refactoring candidates not originally present in the initial system. Moreover, some of the produced sequences of refactorings using the search-based approach may not be feasible to be applied because of dependencies among refactoring candidates; applying one refactoring may conflict with the application of other refactorings. The scheduling approaches using the Genetic Algorithm (GA) do not perform feasibility checking of generated sequence of refactorings and they would require additional cost of repairing. Lee et al. [56] try to resolve the refactoring-conflict problem by repairing infeasible sequences of refactorings, but it seems time-consuming to reorder the randomly generated sequence of refactorings without considering refactoring conflict. As a consequences, for considering newly generated, deleted and changed refactoring candidates, the studies of refactoring scheduling using search techniques would require much of search exploration cost due to too many possible sequences of refactorings to be evaluated. Therefore, in our approach, we attempt to select a group of refactorings that can be applied at a same time.

2.3 Analysis of Dependencies/Conflicts between Refactoring Candidates

When selecting refactorings, as mentioned briefly in the previous section, syntactic dependency (i.e., conflicts) of refactorings should be considered. The syntactic dependency of refactorings indicates that if the application of one refactoring changes or deletes elements necessary for the other refactorings, thus disable those refactorings. Therefore, many research efforts have been invested in resolving conflicts of the extracted refactorings and applying as many refactorings as possible for improving maintainability of software.

Tom Mens et al. [64] represent refactorings as graph transformations; and they propose the techniques of critical pair analysis and sequential dependency analysis to detect the dependencies between refactorings. Using the results of this analysis can help the developer to make an informed decision of which refactoring is most suitable in a given context and why. In the very similar manner, Fawad Qayum et al. [74] represent the system by a graph model and refactoring steps as graph transformation rules. Then, dependency information (which is derived from the analysis of graph transformation) is used for expressing the problem as an instance of the optimisation problem. Zibran et al. [93] notice the importance of considering the syntactic dependency of refactorings. They argue that the application of a subset of refactoring from a set of applicable refactoring activities may result in distinguishable impact on the overall code quality; moreover, there may be sequential dependencies and conflicts among the refactoring activities. Hence, they insist that it is necessary that, from all refactoring candidates a subset of non-conflicting refactoring activities be selected and ordered (for application) such that the quality of the code base is maximized while the required effort is minimized.

Chapter 3. Overview of Our Approach

Framework for Systematic Refactoring Identification

According to [65], the refactoring process consists of the following distinct activities.

1. Identify places where the software should be refactored.
2. Determine which refactoring(s) should be applied.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring on quality characteristics of the software.
6. Maintain the consistency between the refactored program code and other software artifacts such as documentation, design documents, and test cases, etc.

Based on this process, we categorize the activities of the refactoring process into three phases (Table 3.1).

Table 3.1: Identified three phases by referencing refactoring process in [65].

Phase	Description
Refactoring Identification	Determination where to apply which refactorings in what order
Refactoring Application	Actual modification on source code
Refactoring Maintenance	Testing the refactored code, consistency checking with other software artifacts, and change management

”Refactoring-identification phase” refers to planning to determine where to apply which refactorings or how to apply the refactorings for meeting the goal of refactoring, such as improvement of maintainability, understandability, and testability. ”Refactoring-application phase” refers to the task of applying planned refactorings on actual source codes. ”Refactoring-maintenance phase” refers to three activities: testing the refactored code, checking consistency with other software artifacts such as requirement documents or Unified Modeling Language (UML) models, and change management. The refactoring is one kind of code change; therefore, in the change management activity, tasks for recording change logs and change owners—who are responsible for making those changes—for applying each refactoring are needed.

In the thesis, we focus on refactoring identification. To enable automated refactoring, we propose a framework for systematic refactoring identification—the activities of refactoring candidate identification, refactoring candidate assessment, and refactoring selection—(as in Fig. 3.1). In Chapter 4 and Chapter 5, we will explain the

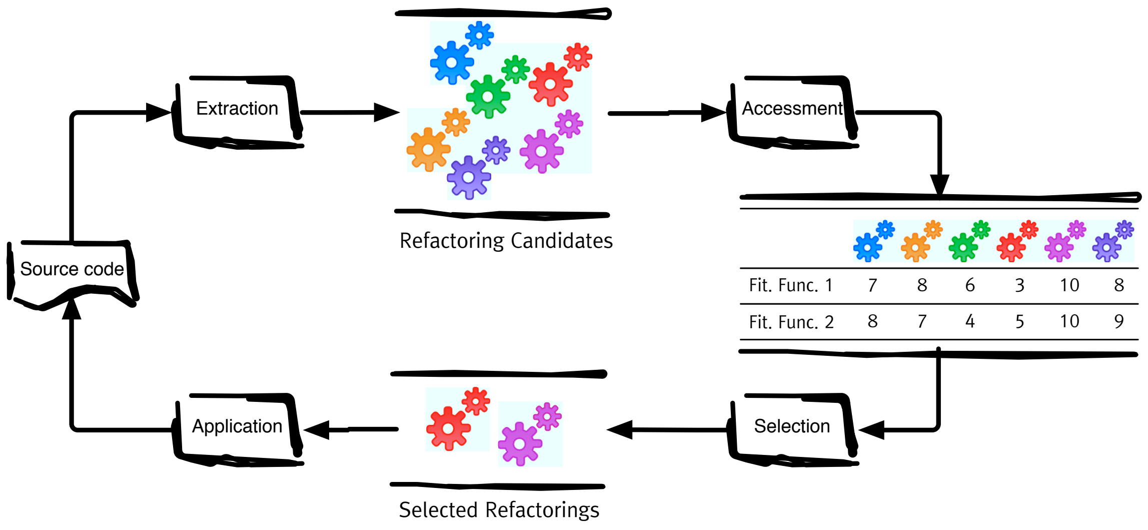


Figure 3.1: A proposed framework for systematic refactoring identification.

detailed methods of the proposed approach for refactoring identification.

Overall Procedure of Our Approach

Fig. 3.2 shows an overview of our proposed approach for refactoring identification. The following briefly describes a procedure of the proposed refactoring identification method. The input of refactoring identification is the source code of program. Refactoring identification consists of two main activities: refactoring-candidate identification and refactoring selection. In the refactoring candidate identification activity, refactoring candidates are (1) extracted using the dynamic information based rules, and (2) grouped into the MISs considering the RED. In the refactoring selection activity, we have developed the refactoring effect evaluation framework (i.e., delta table) for assessing each elementary refactoring effect on maintainability based on matrix computation. Each group of refactorings are sorted in the order of expected degree of improvement on maintainability by using the delta table. The group of refactorings containing the multiple refactorings that best improves maintainability is selected and applied; and the delta table is recalculated. The refactoring selection process (i.e., the procedure of selection for multiple refactorings) is iterated until no more refactorings that can improve maintainability are found. The output is the groups of elementary refactorings, which are the logged results obtained from each refactoring selection process. The next section will explain the detailed procedure and methods of the refactoring-candidate identification and refactoring selection.

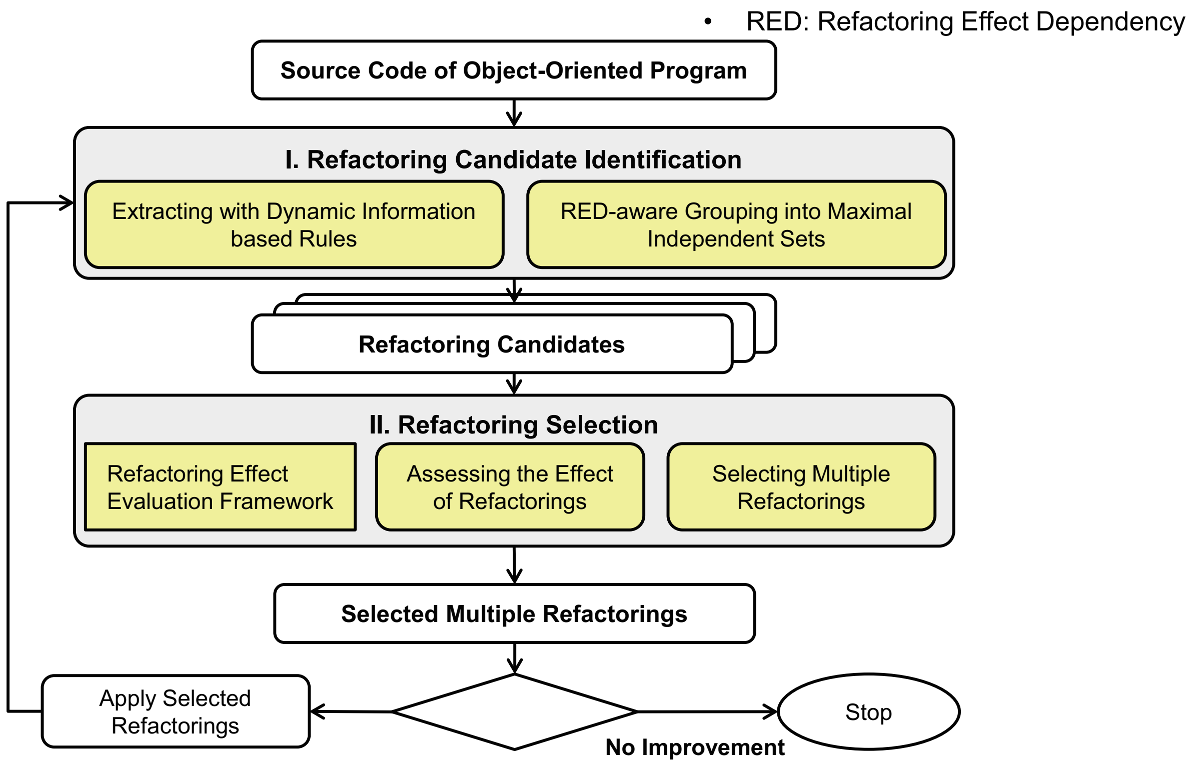

• RED: Refactoring Effect Dependency
Source Code of Object-Oriented Program
I. Refactoring Candidate Identification
Extracting with Dynamic Information based Rules
RED-aware Grouping into Maximal Independent Sets
Refactoring Candidates
II. Refactoring Selection
Refactoring Effect Evaluation Framework
Assessing the Effect of Refactorings
Selecting Multiple Refactorings
Selected Multiple Refactorings
Apply Selected Refactorings
No Improvement
Stop

Figure 3.2: Overall procedure of our approach.

Chapter 4. Identification of Refactoring Candidates

4.1 Top-Down Approach: Extracted with Heuristic Rules

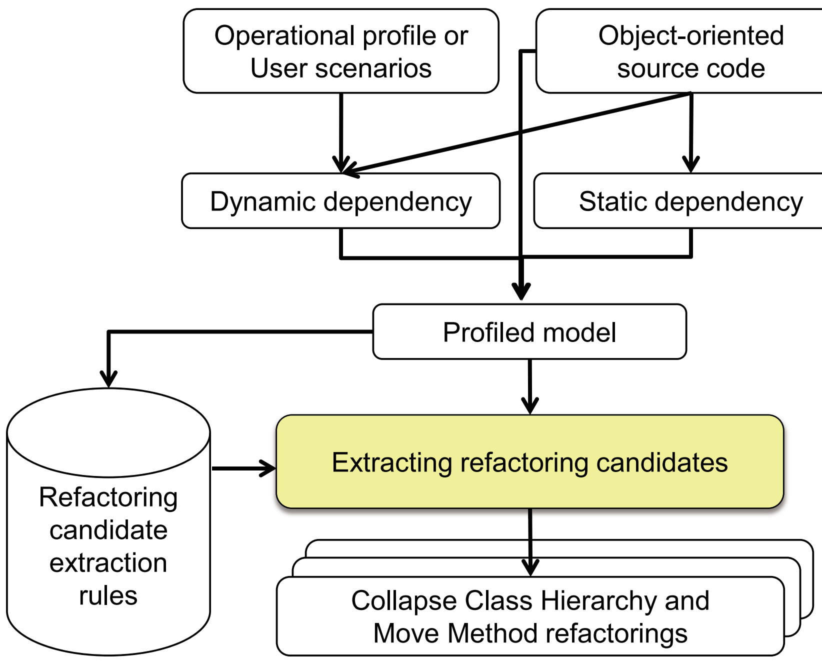

Operational profile or User scenarios
Object-oriented source code
Dynamic dependency
Static dependency
Profiled model
Refactoring candidate extraction rules
Extracting refactoring candidates
Collapse Class Hierarchy and Move Method refactorings

Figure 4.1: Overall approach of the top-down approach: extracted with the dynamic information based heuristic rules.

The Fig. 4.1 illustrates the procedure for the approach of the top-down approach—extracted with the dynamic information based heuristic rules.

4.1.1 Dynamic Information-based Identification of Refactoring Candidates

Refactoring candidates are extracted with the aim of reducing dependencies of entities of methods and classes, since the goal of the refactoring in our approach is to make software accommodate changes more easily. By motivated by the studies—the data capturing how the system is utilized is an important factor for estimating changes—, we use dynamic information of how the users utilize the software for identifying the entities involved in given user scenarios and operational profile [66]; and within these entities, refactoring candidates are extracted. Regarding dynamic information, we use dynamic profiling technique to obtain the dynamic dependencies of entities—Dynamic Method Calls (DMCs)—based on dynamic method calls by executing programs based on user scenarios or operational profiles; and we specifically designed the profiled model, Abstract Object Model (AOM), for saving the dynamically and statically profiled information.

Change Preventing Related Design Problems and Resolving Refactorings

Dependency refers to a relationship where the structure or behavior of an entity is dependent on another entity [2]. UML defines dependency as a relationship where a change to the influent modeling element may affect the dependent modeling element [70]. Object-oriented software involves structural and behavioral aspects of dependencies [32]. The relationships in classes such as association, aggregation, composition, and inheritance represent structural dependencies. *Behavioral dependency* occurs when a method calls another method (i.e., when a method requires a service from another method to execute its own behavior); in this case, the methods or the owner classes of those methods have behavioral dependencies. In our approach, we consider a behavioral dependency that occurs due to a method call. Note that structural and behavioral dependencies are not mutually exclusive; an entity can have both structural and behavioral dependencies on another entity [32]. For example, a ”use” type of association relationship for structural dependency entails behavioral dependency. High dependency between entities makes change-sensitive software in that many classes are modified when making a single change to a system (e.g., Shotgun Surgery [29]), or a single class is modified by many different types of changes (e.g., Divergent Change [29]). This makes software difficult to maintain, and, thereby, lowers the overall maintainability level. The kinds of situations mentioned above should be resolved. Therefore, refactorings should be applied in a way that reduces dependencies of entities (i.e., methods and classes), resulting in software accommodate changes more easily.

Fowler [29] suggests considerable refactorings for resolving the change preventing related bad smells—Divergent Change and Shotgun Surgery—as follows: Inline Class (i.e., merging class; in our approach, Collapse Class Hierarchy), Move Method, Move Field, and Extract Class, etc. Among the mentioned refactorings, we currently support two refactorings: Collapse Class Hierarchy and Move Method. In a Collapse Class Hierarchy refactoring, all methods and fields contained in a class are moved into another class; subsequently, the moved class is deleted. In a Move Method refactoring, a method is moved into a target class. We do not consider Move Field refactoring—moving attributes (i.e., fields) from one class to another class—, because fields have stronger conceptual binding to the classes in which they are initially placed since they are less likely than methods to change once assigned to a class [88]. For Extract Class refactoring, our rule-based approach has difficulty in determining specific code blocks to be split in an automated way; therefore, we leave this refactoring for future work.

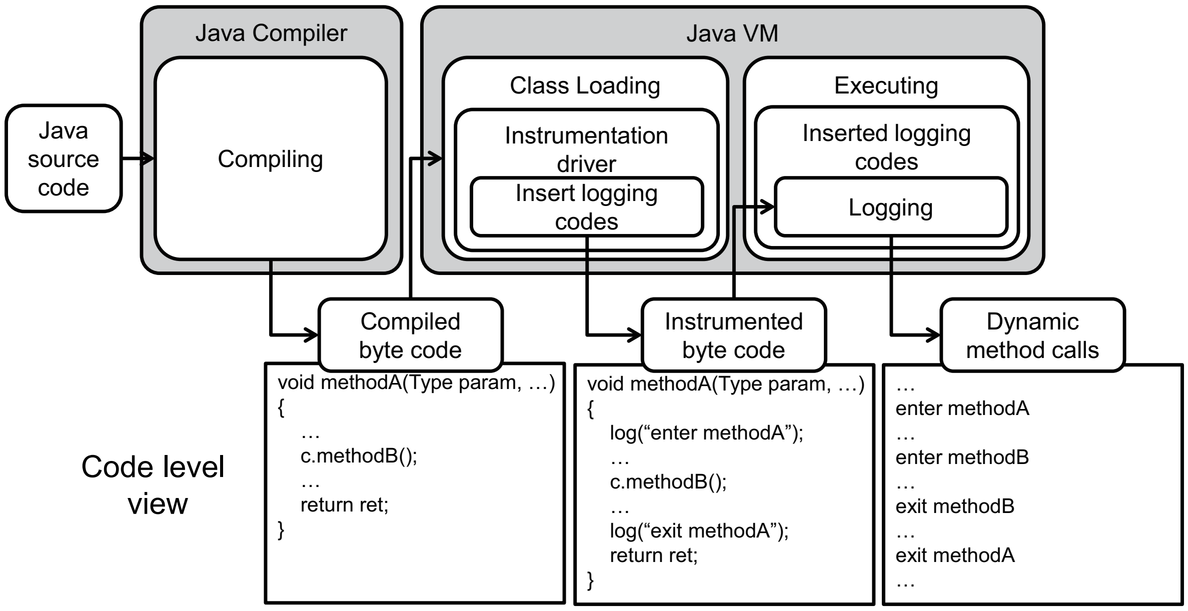
Use of Dynamic Information to Find Refactoring Candidates in Change-Prone Parts

Previous studies have shown that the data capturing how the system is utilized is an important factor for estimating changes. Robbes et al. [77] show that using program usage data recorded from Integrated Development Environments (IDEs) significantly improves the overall accuracy of change prediction approaches. The experimental results of the other study [3] and our study [39] show that dynamic coupling measures and behavioral dependency measures— that are obtainable during run-time execution and pinpoint the systems' parts that are often used—are good indicators for predicting change-prone classes.

Being motivated by these works, we have come to argue that if changes are more prone to occur in the pieces of codes that users more often utilize, then applying refactorings in these parts would fast improve maintainability of software. The underlying assumption is that the pieces of codes that have been used more are more likely to undergo changes in a future version; therefore, investing efforts on the refactorings involving such codes may effectively improve maintainability. By using only static information (i.e., that can be obtained by analyzing source codes statically without running a program) such as structural complexity of the program, refactorings may be suggested on rarely, or, even worse, never-used entities. If changes have never occurred in such entities, then the benefit—for example, reduced maintenance cost for accommodating the changes—of the application of those refactorings may be little to none. In this case, refactorings need to be applied on the other entities.

Dependencies of Entities Based on Dynamic Method Calls (DMCs)

The procedure of dynamic profiling technique used in our approach is presented; and the definition and measurement of the DMC is provided. At last, the profiled model use in our approach is explained.



```
void methodA(Type param, …)
{
    …
    c.methodB();
    …
    return ret;
}
```

```
void methodA(Type param, …)
{
    log("enter methodA");
    …
    c.methodB();
    …
    log("exit methodA");
    return ret;
}
```

```
…
enter methodA
…
enter methodB
…
exit methodB
…
exit methodA
…
```

Figure 4.2: Procedure for dynamic profiling.

Dynamic profiling. Dynamic profiling is a form of dynamic program analysis that measures, for example, the use of memory, the use of particular instructions, or frequency or duration of method calls; it is achieved by instrumenting either the program source code or its binary executable form. Dynamic profiling has been used by many researchers [47, 16, 84]. The most common use of dynamically-profiled information is to aid program optimization—for example, the compiler writers use it to find out how well their instruction scheduling or branch prediction algorithm performs. In our approach, the dynamic profiling technique is used to obtain the dependencies of entities of methods and classes by executing programs the same way as in live operation based on user scenarios or operational profiles—a quantitative representation of how the software will be used. Note that in software reliability engineering, for making reliability estimation, user scenarios or operational profiles [66] are developed and maintained to describe how users actually utilize the system. These dependencies are obtained by logging the frequency of method calls; those dynamic dependencies are defined as DMCs.

Fig. 4.2 depicts the procedure of dynamic profiling used in this thesis. The java instrumentation technique [23] is used for dynamic profiling. On the compiled byte code, entering and exiting logging codes are inserted at the start and the end of all method declarations. This enables the tracing of logs of executed methods while executing a program without modifying the original source codes.

Table 4.1: Difference between the static method call (SMC) and the dynamic method call (DMC).

	SMC	DMC
Source of measurement	Source codes or structural models	Programs execution according to user scenarios or operational profiles
Subject of measurement	Class and method	Object and message
Degree of measurement	Number of distinct methods	Number of all messages

Dynamic Method Call (DMC). DMC is an instantiated form of a static method call (SMC). The differences between the DMC and the SMC is explained in Table 4.1. Definition 1 offers a precise definition of the DMC.

Definition 1 (Definition of DMC)**.** *When an object o_1 sends a message n to an object o_2, there exists a DMC. We denote this relation as $o_1 \overrightarrow{n} o_2$. In the definition, DMC is represented as dmc and it consists of six attributes as follows:*

- *id: a unique identifier.*
- *m_{callee}: a method from which the message n is initiated; a method called from the method m_{caller}.*
- *m_{caller}: a method which calls m_{callee}.*
- *$c_{callType}$: a calling type class; a structural callee class.*
- *c_{caller}: an owner class of method m_{caller}.*
- *c_{callee}: an owner class of method m_{callee}.*

As listed, id refers to a unique identifier of the dmc. The dmc can be defined with two ends of methods m_{caller} and m_{callee}, and two ends of classes c_{caller} and c_{callee}, which are the owner classes of those methods. The $c_{callType}$ is a calling type class that denotes a structural callee class. The DMCs existing in the system can be retrieved with respect to two parameters: (1) entity (ε) such as method and class; and (2) direction (δ) such as import and export. The DMC for a class or method in the *import* direction occur when the class or method imports services from external class(es); in other words, the class or method uses other methods that are defined in external class(es). On the other hand, the DMC for a class or method in the *export* direction occur when the class or method exports services to external class(es); in other words, other methods defined in external class(es) use the class or method. We specify each direction of import and export using the following symbols, $\prec$ and $\succ$, respectively. We denote $DMC(\varepsilon, \delta)$ as the list of DMCs that are retrieved respect to the entity ε and the direction δ.

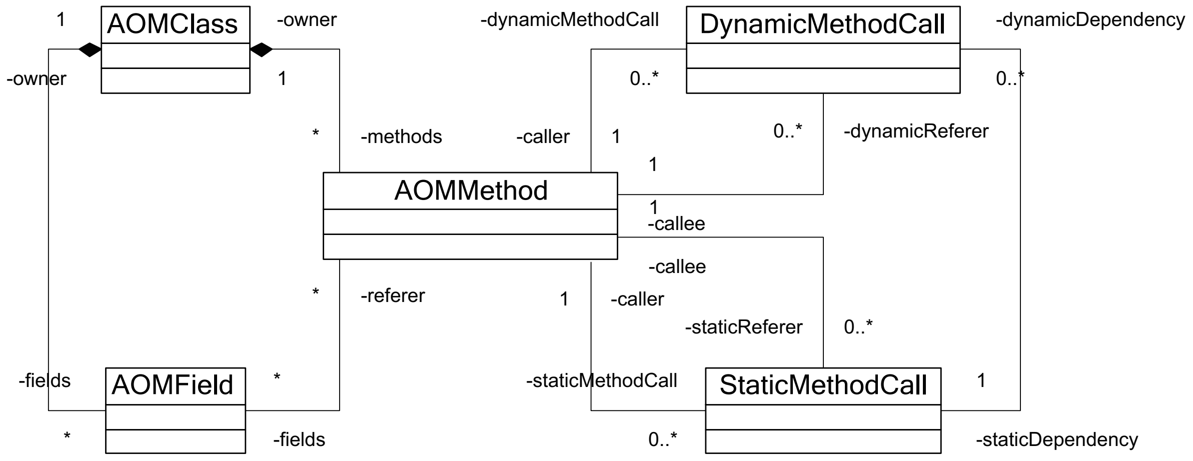


Figure 4.3: A metamodel of the AOM used in this thesis.

Abstract Object Model (AOM). AOM is the profiled model; and it is specifically designed for saving the dynamically and statically profiled information. Fig. 4.3 shows the metamodel of the AOM. A method meta-class AOMMethod is associated with a DMC meta-class DynamicMehtodCall and a SMC meta-class StaticMethodCall. This enables a DMC/SMC to be navigable with two ends: a caller method and a callee method. In the opposite direction, a method is able to navigate the DMCs/SMCs that the method calls and the DMCs/SMCs by which the method is referred. The DMC meta-class DynamicMehtodCall is also associated with the SMC meta-class StaticMethodCall. If multiple or even zero method calls exist between two entities (such as methods or classes) during run-time execution, the SMC counts this as one. In other words, the multiplicity of the SMC to the DMC is 0..*, whereas the multiplicity of the DMC to SMC is one. This enables the DMC to be navigable with the SMC from which the DMC is instantiated. In the opposite direction, the SMC is able to navigate DMCs that are actually instantiated. By maintaining the metamodel of the AOM, information related to the DMCs can be updated without re-doing dynamic profiling at every trial of refactoring. When dynamic profiling, the DMCs are *mapped* into the corresponding SMCs from which those DMCs are instantiated. Therefore, for each application of refactoring, by adjusting the information related to the SMCs—such as the classes and fields of two ends of caller and callee methods of the SMCs—, the updated information related to the DMCs can be obtained by tracing the information related to the SMCs.

4.1.2 Refactoring Candidate Extraction Rules

Based on the DMCs, the rules are defined for extracting refactoring candidates. By trying every refactoring-candidate extraction rule, pairs of entities (i.e., methods and classes) are extracted as refactoring candidates according to the heuristic design strategy, which is defined in a way aimed at reducing dependencies of those entities; then, using the max function, the part of refactoring candidates that are highly-ranked with the scoring function are chosen to be assessed. The heuristic design strategies used in our approach are explained in subsection 4.1.2.

Elements of Refactoring-Candidate Extraction Rule

Refactoring candidate extraction rules specify where to refactor and which refactoring to use. Each rule consists of three elements: (1) the scoring function, (2) the max function, and (3) the specific corresponding refactoring to apply:

Scoring function. A scoring function is a kind of a fitness function. It represents how much each pair of entities—which is extracted as a refactoring candidate according to a heuristic design strategy—fits into the heuristic design strategy. Therefore, a scoring function is designed to retrieve how many times a pair of entities is extracted as a refactoring candidate using the heuristic design strategy.

Max function. It is infeasible to assess all the refactoring candidates extracted from all the defined rules, because there are too many. Note that to assess refactoring candidates, each refactoring candidate has to be individually applied to the current version of the program and its effect on the refactored program is evaluated, which requires a large computation cost. Therefore, we assess only the refactoring candidates that are highly-ranked (top k) with scoring functions. The role of the max function is to cut off the top k refactoring candidates to be assessed. In the rule, the $cutline$ number represents k.

Refactoring. As stated in subsection 4.1.1, we use two types of refactorings—Move Method and Collapse Class Hierarchy—, and the operations of those refactorings are presented in Algorithm 1 and Algorithm 2, respectively. We formulate pre- and post-conditions referring to [88, 36, 71] and check before and after refactoring applications. We do not specify these conditions in this thesis.

Algorithm 1 Collapse Class Hierarchy

```
Require: $C_{merging}$: a class that is merging the other class,
Require: $C_{merged}$: a class that is to be merged
  for all $M_{merged} \in C_{merged}$ do
    Move Method($C_{merging}$, $M_{merged}$)
  $C_{merged}.ancestor \leftarrow C_{merged}.ancestor \cup C_{merging}.ancestor$
  $C_{merged}.descendant \leftarrow C_{merged}.descendant \cup C_{merging}.descendant$
  $C_{merged}.field \leftarrow C_{merged}.field \cup C_{merging}.field$
  remove $C_{merged}$
```

Algorithm 2 Move Method

```
Require: $C$: a target class to which a method is moved
Require: $M$: a method to be moved
  $M.overridingMethod \leftarrow$ findingOverriding($C$, $M$)
  /*findingOverriding function is specified in Algorithm 3.*/
  $C.method \leftarrow C.method \cup \{M\}$
```

Algorithm 3 findingOverriding

```
Require: $c$: a target class to which a method is moved
Require: $m$: a moving method
/*Algorithm findingOverriding returns the method by which $movingMethod$ is overrided.*/
$queue \leftarrow$ ancestor classes of $c$
$visitedClass \leftarrow \emptyset$
while $queue \neq \emptyset$ do
    $tmpClass \leftarrow$ remove one element of class from $queue$
    add $tmpClass$ to $visitedClass$
    for all $tmpMethod \in$ methods in $c$ do
        if $tmpMethod = m$ then
            return $tmpMethod$
    for all $ancClass \in$ ancestor classes of $tmpClass$ do
        if $visitedClass$ does not contain $ancClass$ then
            add $ancClass$ to $queue$
return $null$
```

Design of Refactoring-Candidate Extraction Rule

For each design strategy, pairs of methods (or classes) are extracted as the entities of the refactoring candidate of Move Method (or Collapse Class Hierarchy), and the number of extractions for the pairs of methods (or classes) is retrieved by the scoring function. The rules are defined in the following way: a part of refactoring candidates that are highly-ranked with the scoring function are chosen to be assessed using the max function. Note that for each strategy, two types—method and class—of scoring functions are obtained, and three rules are defined.

In the following, for each type of heuristic design strategy, we present a brief explanation and the procedure for obtaining the corresponding scoring functions. We then define the refactoring-candidate extraction rules using the obtained scoring functions in a semi-formal way.

[Heuristic design strategy type 1.]

Explanation. It is better to gather the methods, which are called by one method but are spread over many different classes, into one class. Let a method m call the methods, and those called methods are implemented in different classes. The N stands for the threshold to determining the situation such that those called methods are implemented in many different classes. Therefore, we define the following heuristic design strategies: when those called methods are implemented in the N (N = 2, 3, 4, 5, and 6) classes, those methods (or their owner classes) are extracted as the entities of refactoring candidates of Move Method (or Collapse Hierarchy Class). In this thesis, we set the N from 2 to 6, because we have tested for all methods in all three subjects—jEdit, Columba, and JGIT—, and the maximum number of different classes for each subject does not exceed 6. Note that 1 need not to be examined because it means all the called methods are in the same class. The N is not fixed and can be differentiated according to the characteristic of the used project.

Algorithm 4 get$N$Diff_M_and_NDiff_C (N = 2, 3, 4, 5, 6)

```
for all $c \in$ classes in the system, $m \in$ methods in $c$ do
  $diffClass \leftarrow \emptyset$    /*a set for saving different callee classes*/
  for all $dmc \in DMC(m,\prec)$ do
    if $dmc.c_{callee} \neq c$ then
      add $dmc.c_{callee}$ to $diffClass$
  if $diffClass.size \geq N$ then
    for all $dmc_1, dmc_2 \in DMC(m,\prec)$ do
      if $dmc_1 \neq dmc_2$ then
        $c_1 \leftarrow dmc_1.c_{callee}$, $c_2 \leftarrow dmc_2.c_{callee}$
        if $c_1 \neq c_2$ && $c_1 \neq c$ && $c_2 \neq c$ then
          $N$Diff_C$(\{c_1, c_2\}) \leftarrow N$Diff_C$(\{c_1, c_2\}) + 1$
        $m_1 \leftarrow dmc_1.m_{callee}$, $m_2 \leftarrow dmc_2.m_{callee}$
        if $m_1 \neq m_2$ && $m_1 \neq m$ && $m_2 \neq m$ then
          $N$Diff_M$(\{m_1, m_2\}) \leftarrow N$Diff_M$(\{m_1, m_2\}) + 1$
```

Procedure. Algorithm 4 is illustrated for obtaining scoring functions as follows. For all class c in the system, and for all method m in class c, let a method m call the methods, and those called methods are implemented in different classes. If the number of different classes is greater than or equal to N—the threshold to determining the situation such that methods are implemented in many different classes—, then the pair of methods in the list of the called methods is extracted as the entities of the refactoring candidate of Move Method, and the number of extraction for the pair of methods is increased for the scoring function NDiff_M (when methods in the pair are neither identical to each other nor identical with the method m). This also applies to the class-level; therefore, the pair of classes in the list of classes—the owner classes of those called methods—is extracted as the entities of the refactoring candidate of Collapse Hierarchy Class, and the number of extraction for the pair of classes is increased for the scoring function NDiff_C (also when classes in the pair are neither identical to each other nor identical with the class c).

Rules. The N stands for 2, 3, 4, 5, and 6, hence for this type of the heuristic design strategy, five design strategies are defined; then, a total of 15 rules are defined.

- $\mathbf{R}_{1(N=2)}$, $\mathbf{R}_{4(N=3)}$, $\mathbf{R}_{7(N=4)}$, $\mathbf{R}_{10(N=5)}$, $\mathbf{R}_{13(N=6)}$:
  $^{\forall}(c_i, c_j) \in max(N\text{Diff\_C}, cutline)$
  $\rightarrow$ Collapse Class Hierarchy(c_i, c_j)

- $\mathbf{R}_{2(N=2)}$, $\mathbf{R}_{5(N=3)}$, $\mathbf{R}_{8(N=4)}$, $\mathbf{R}_{11(N=5)}$, $\mathbf{R}_{14(N=6)}$:
  $^{\forall}(m_i, m_j) \in max(N\text{Diff\_M}, cutline)$
  $\rightarrow$ Move Method(owner class of m_i, m_j)

- $\mathbf{R}_{3(N=2)}$, $\mathbf{R}_{6(N=3)}$, $\mathbf{R}_{9(N=4)}$, $\mathbf{R}_{12(N=5)}$, $\mathbf{R}_{15(N=6)}$:

$^{\forall}(m_i, m_j) \in max($NDiff_M, $cutline)$

$\rightarrow$ Move Method(owner class of m_j, m_i)

Algorithm 5 getI_C_and_I_M
```
for all $c \in$ classes in the system, $m \in$ methods in $c$ do
  if $m \neq null$ then
    for all $dmc_1 \in DMC(m, \prec)$ do
      $c_1 \leftarrow dmc_1.c_{caller}$, $c_2 \leftarrow dmc_1.c_{callee}$
      if $c_1 \neq c_2$ && $c_1 \neq c$ && $c_2 \neq c$ then
        $I\_C(\{c_1, c_2\}) \leftarrow I\_C(\{c_1, c_2\}) + 1$
      $m_1 \leftarrow dmc_1.m_{caller}$, $m_2 \leftarrow dmc_1.m_{callee}$
      if $m_1 \neq m_2$ && $m_1 \neq m$ && $m_2 \neq m$ then
        $I\_M(\{m_1, m_2\}) \leftarrow I\_M(\{m_1, m_2\}) + 1$
```

[Heuristic design strategy type 2.]

Explanation. Again, it is better to gather methods that have many interactions into one class. Let a method m call the other method n, and those methods are implemented in different classes. Then, we define the following heuristic design strategy: when those two methods have interactions, those methods (or their owner classes) are extracted as the entities of refactoring candidates of Move Method (or Collapse Hierarchy Class).

Procedure. Algorithm 5 is illustrated for obtaining scoring functions as follows. For all class c in the system, and for all method m in class c, let a method m call the other method n, and those methods are implemented in different classes. Subsequently, the pair of methods is extracted as the entities of the refactoring candidate of Move Method, and the number of extraction for the pair of methods is increased for the scoring function I_M (when the methods in a pair are not identical to each other). This also applies to the class-level; therefore, the pair of methods classes—the owner classes of those methods—is extracted as the entities of the refactoring candidate of Collapse Hierarchy Class, and the number of extraction for the pair of classes is increased for the scoring function I_C (also when classes in a pair are not identical to each other).

Rules. For this type of the heuristic design strategy, one heuristic design strategy is defined; then, three rules are defined. Note that for each rule, the refactoring candidates which are highly-ranked (top $cutline$) with scoring functions are chosen to be assessed; this can be said that the pairs of entities that have many interactions are chosen to be assessed.

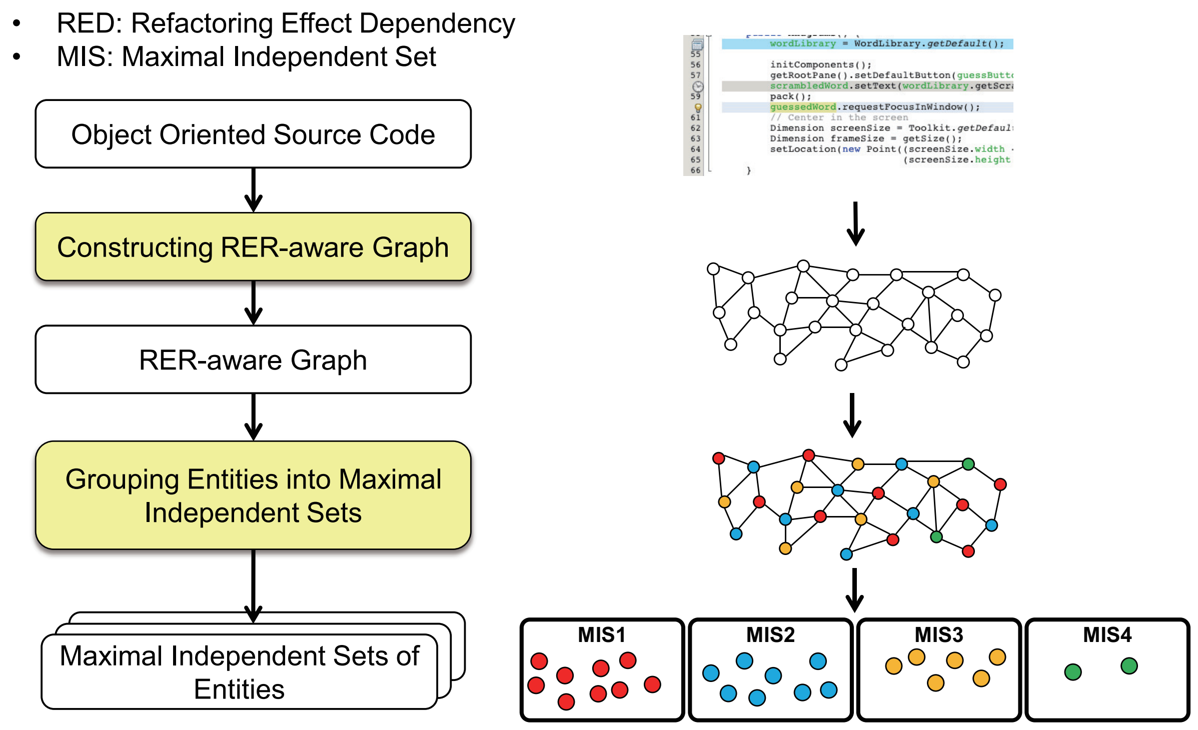
- **R**$_{16}$: $^{\forall}(c_i, c_j) \in max($I_C, $cutline)$
  $\rightarrow$ Collapse Class Hierarchy(c_i, c_j)

- **R**$_{17}$: $^{\forall}(m_i, m_j) \in max($I_M, $cutline)$
  $\rightarrow$ Move Method(owner class of m_i, m_j)

- **R**$_{18}$: $^{\forall}(m_i, m_j) \in max(I_M, cutline)$

  $\rightarrow$ Move Method(owner class of m_j, m_i)

4.2 Bottom-Up Approach: Grouping Entities into Maximal Independent Sets


• RED: Refactoring Effect Dependency
• MIS: Maximal Independent Set
Object Oriented Source Code
Constructing RER-aware Graph
RER-aware Graph
Grouping Entities into Maximal Independent Sets
Maximal Independent Sets of Entities
MIS1
MIS2
MIS3
MIS4

```
        wordLibrary = WordLibrary.getDefault();

        initComponents();
        getRootPane().setDefaultButton(guessButt
        scrambledWord.setText(wordLibrary.getScr
        pack();
        guessedWord.requestFocusInWindow();
        // Center in the screen
        Dimension screenSize = Toolkit.getDefaul
        Dimension frameSize = getSize();
        setLocation(new Point((screenSize.width
                              (screenSize.height
    }
```

Figure 4.4: Overall approach of the bottom-up approach: grouping entities into MISs.

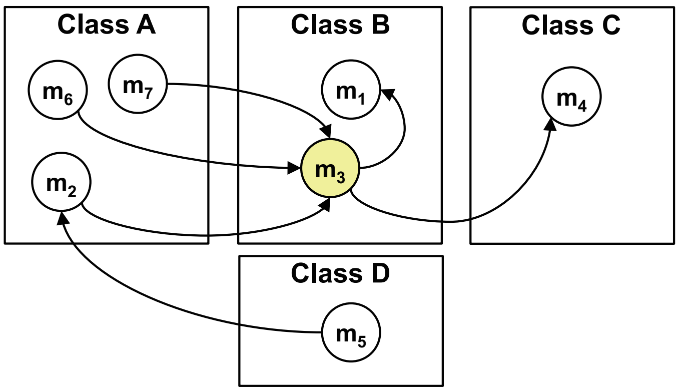
For grouping of elementary refactorings considering the RED, the RED-aware graph is constructed; and based on the graph, the entities of methods and attributes are grouped into the MISs. By referring the delta table—obtained from the refactoring effect evaluation framework for evaluating each elementary refactoring effect on maintainability—, methods involved in each MIS are mapped into a group of elementary Move Method refactorings. Note that the Move Method refactorings in the same group can be applied at the same time. The Fig. 4.4 illustrates the procedure for the approach of the bottom-up approach—grouping entities into MISs.

Before explaining about the algorithm of RED-aware grouping of MISs, we explain the importance of the RED.

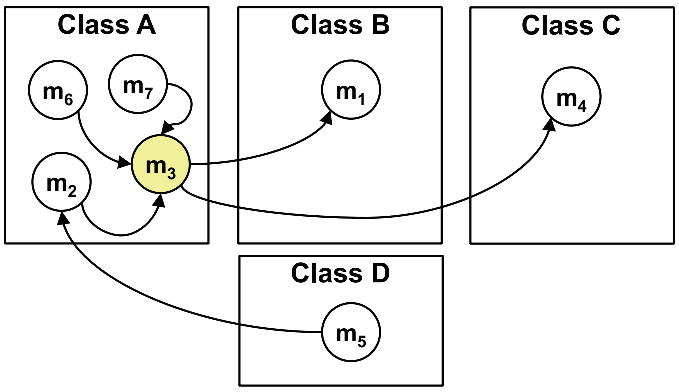
4.2.1 Refactoring Effect Dependency (RED) on Maintainability

When grouping elementary refactorings that are to be applied at the same time, refactoring dependency needs to be considered. The syntactic dependency of refactorings indicates that the application of one refactoring changes or deletes elements necessary for the other refactorings thus it disables those refactorings. Many research efforts have been invested in resolving the syntactic dependencies of the refactorings and applying as many refactorings as possible for improving maintainability of software.

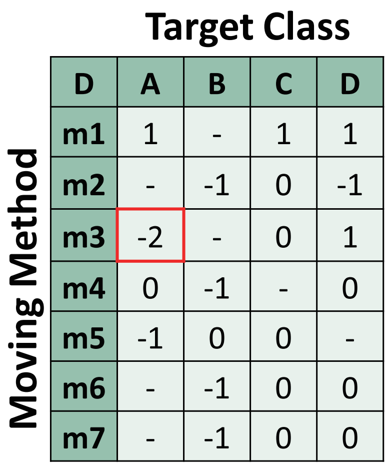
Even though the refactorings are not conflict each other, however, applying all the refactorings—that are expected to improve maintainability—does not guarantee to improve maintainability. This is because refactorings'



(a) An example design.



(b) After moving method m3 to class A (from Fig. 4.5(a)).

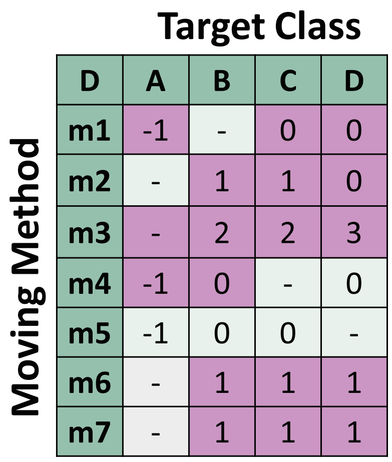


Target Class

Moving Method

D	A	B	C	D
m1	1	-	1	1
m2	-	-1	0	-1
m3	-2	-	0	1
m4	0	-1	-	0
m5	-1	0	0	-
m6	-	-1	0	0
m7	-	-1	0	0

(c) Δ coupling for each moving method refactoring (Fig. 4.5(a)).



Target Class

Moving Method

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

(d) Δ coupling for each moving method refactoring (Fig. 4.5(b)).

Figure 4.5: A motivating example of showing the need of the RED on maintainability.

effect on maintainability is dependent each other. Once a refactoring applied, the design configuration of the software is changed; and this may influent other refactorings' effect on maintainability. In other words, other refactorings' effect on maintainability may be changed as the status (e.g., how the entities are associated and where the entities are placed) of the software design is changed. Therefore, the pre-calculated (i.e., intended) effect on maintainability of other refactorings—that even do not have syntactic dependencies—may be changed.

Fig. 4.5 is a motivating example showing the need of RED. The system (in Fig. 4.5(a)) consists of four classes and each class contains methods A = {m_2, m_6, m_7}, B = {m_1, m_3}, C = {m_4}, and D = {m_5}; and the method calls are represented with directed edges. Let the number of edges across the classes be *coupling* value. Each cell in the table (rows: moving methods, columns: target classes) in Fig. 4.5(c) represents the delta of coupling value after the application of each Move Method refactoring on the design (Fig. 4.5(a)). For example, the delta of coupling value after moving the method m_3 to the class A is -2. (Let Move Method(method m_3, class A) denote this refactoring.) In other words, this application of refactoring reduces the coupling value as -2. Then, expected output of total reduced coupling value after the application of the two refactorings, Move Method(method m_3, class A) and Move Method(method m_2, class B), is -3 (= -2 -1). However, after applying Move Method(method m_3, class A), the design configuration is changed (as in Fig. 4.5(b)). The cells shaded with pink color in Fig. 4.5(d) represent the changed delta of coupling values after the application of the refactoring Move Method(method m_3, class A). As a result, the delta of coupling value after the application of the refactoring Move Method(method m_2, class B) is +1 (not -1); and the actual output of total reduced coupling value by applying those two refactorings is -1 (= -2 +1). Without considering the RED on maintainability, not intended results may come up; and even worse the total effect of the application of refactorings may be none. To the best of my knowledge, no one noticed or discussed this kind of refactoring dependency before.

The RED is essential to be considered to correctly identify a group of refactorings that most improve maintainability for each iteration of the refactoring identification process. As a result, when grouping elementary refactorings, we take into account the new dependency of refactorings—RED. We provide the clear definition of the RED on maintainability.

Definition 2 (Definition of the RED on Maintainability)**.** *The effect on maintainability of refactoring A and refactoring B is dependent each other, when the application of the refactoring A changes the effect on maintainability of refactoring B, and vice versa, even those two refactorings are not syntactically dependent.*

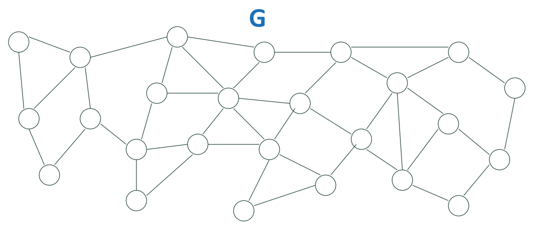
4.2.2 Algorithm of RED-aware Grouping: Maximal Independent Set (MIS) Calculation

We develop the method for RED-aware grouping of elementary refactorings by using the concept of the MIS in graph theory.

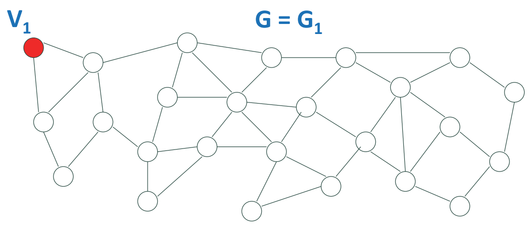
Maximal Independent Set in Graph Theory

We present a MIS [53, 59, 1] in graph theory. Its concept is used for grouping of elementary refactorings. Given a graph $G = (V, E)$, an Independent Set (IS) is a set of vertices $S \subseteq V$ such that if $u, v \in S$, then $(u, v) \notin E$. In short, an IS is a set of vertices in G such that no two vertices in IS are adjacent (i.e., connected by an edge). A Maximal Independent Set (MIS) is an IS to which no more vertices can be added without violating independence property. In short, a MIS is an IS that is not a subset of any other IS. A Maximum Independent Set (MaxIS) is an IS with maximum cardinality among all IS sets of G.

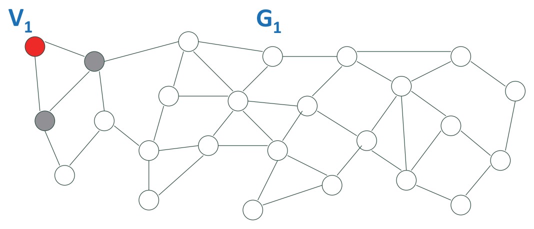
Finding a MIS is trivial in the sequential algorithm; it just scan the vertices in arbitrary order. If a vertex u



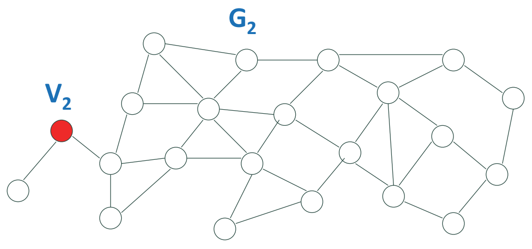
(a) Initially, I = empty.



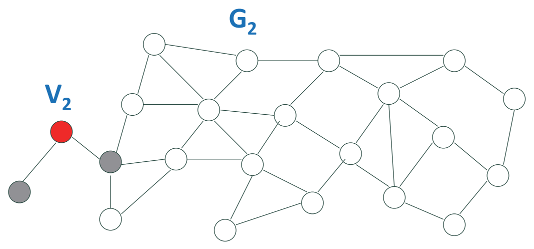
(b) Phase 1: Pick a node V1 and add it to I.



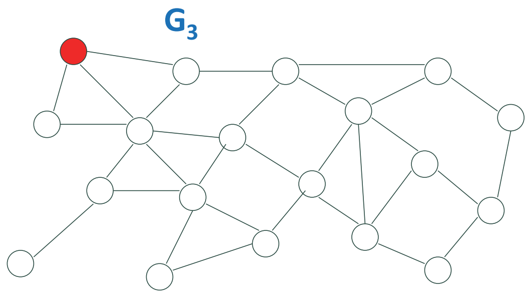
(c) Phase 1: Remove V1 and neighbors N(V1).



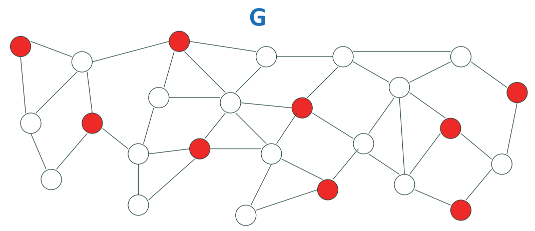
(d) Phase 2: Pick a node V2 and add it to I.



(e) Phase 2: Remove V2 and neighbors N(V2).



(f) Phase 3,4,5,... : Repeat until all nodes are removed.



(g) At the end, set I will be an MIS of G.

Figure 4.6: A sequential algorithm of calculating a MIS. This is reference from the lecture note of Costas Busch [17].

does not violate independence, then add u to the MIS. Otherwise, if u violates independence, then discard u. The figures (in Fig. 4.6) illustrate the sequential algorithm for calculating a MIS.

On the contrary, computing a MaxIS is a notoriously difficult problem. It is equivalent to maximum clique on the complementary graph. Both problems are NP-hard, in fact not approximable within $n^{\frac{1}{2}-\epsilon}$. Therefore, finding all the existing MISs is also the NP-hard problem. For the reason stated above, we use some heuristic for calculating MISs, which is scalable for large size of programs. We try to find MISs, each of which has as many independent entities (which are later to be transformed into elementary refactorings) as possible. This is because the more elementary refactorings are in a MIS, the larger maintainability improvement can be expected. Thus, we find the intermediate groups of entities—that already have the large number entities as possible—by grouping the entities using transitive independent relations. This algorithm is also deterministic—the calculated the MISs is always same for a certain system.

Algorithm of RED-aware Grouping: Maximal Independent Set Calculation

For RED-aware grouping of elementary refactorings, we calculate MISs of entities based on the graph of the RED-aware graph; and entities in a MIS is be mapped into elementary refactorings in a group. Note that each elementary refactoring in the group can be applied at the same time.

The RED-aware graph G_R = (V_R, E_R) of the corresponding object-oriented program is constructed as follows.

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

The vertices (V_R) indicate the entities of methods and attributes. The edges (E_R) indicate the association between entities. The entities that are associated when they are preferably to be located in the same class for improving maintainability (in term of low coupling and high cohesion). To this end, we connect the edge between the entities when (1) a method calls the other method (method_calls), (2) a method assesses an attribute (attribute_assesses$_1$), and (3) two methods assess the same attribute (attribute_assesses$_2$).

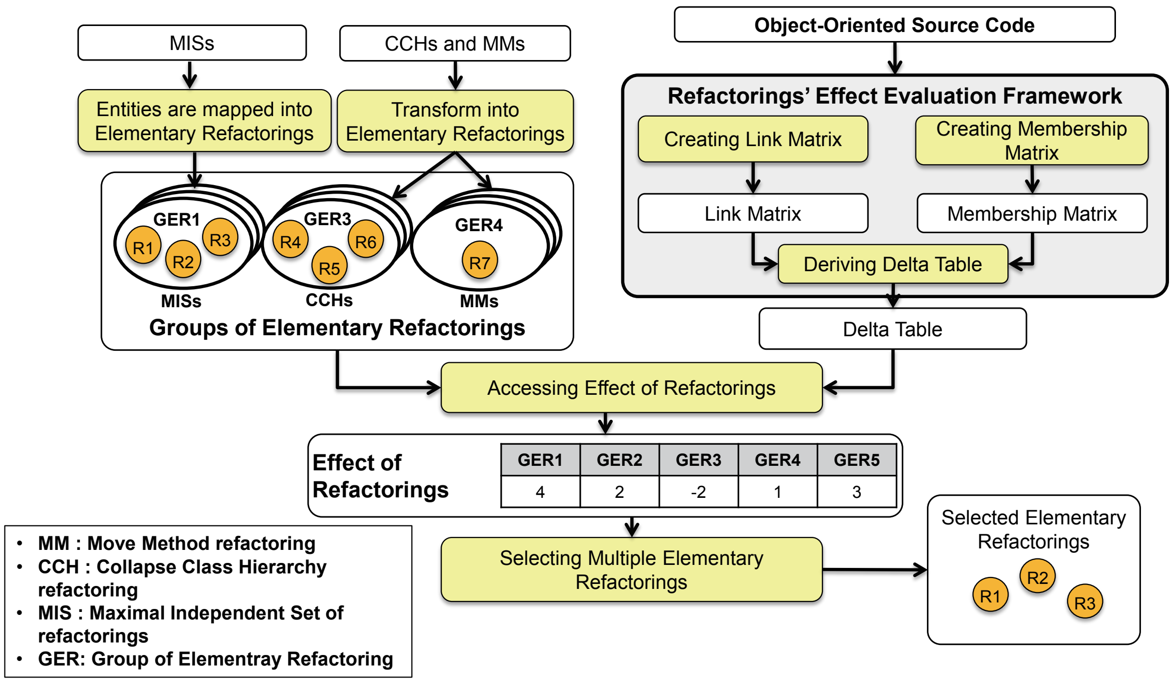
The Fig. 4.7 shows the algorithm for the method of calculating MISs. In the algorithm, we provide the strict constraints to prevent from producing different outcome of MISs for every execution on the same input system. For the first step, based on the RED-aware graph G_R = (V_R, E_R), the independent relations are extracted. An *independent relation* indicates the relation: two vertices u, $v \in V_R$ and $(u, v) \notin E_R$. For the second step, we find the intermediate groups of entities—that already have the large number entities as possible—by grouping the entities using transitive independent relations. The size of the intermediate groups of entities should be greater than threshold λ. The threshold λ is determined by the average size of the intermediate groups of entities of the certain programs. We use the threshold to cut the candidates of intermediate groups of entities; and in the experiment, we show that the calculation for MISs is scalable for large size of programs. After determining the intermediate groups of entities, we assign the remaining entities—that are not added into the groups—until no more entities can be added to any other groups of entities without violating the independence property. Finally, MISs of entities are obtained. Then, we exclude entities of attributes from the MISs of entities. Please note that, in the thesis, we use the Move Method refactoring as the elementary refactoring. As mentioned in subsection 4.1.1, we do not consider Move Field refactoring—moving attributes (i.e., fields) from one class to another class—, because fields have stronger conceptual binding to the classes in which they are initially placed since they are less

```
import sys
from MRModel import *
from MREntity import *
from MRMethod import *
from MRField import *
from MRClass import *
from numpy import *
import scipy as sp
import random
def generateIndependentSets(matrix, numMethod):
    (rows, cols) = matrix.nonzero()
    independent_vertex_set_set = set()
    for k in range(50):
        independent_vertex_set_len = len(independent_vertex_set_set)
        rowidxs = list(range(len(rows)))
        random.shuffle(rowidxs, random.random)
        independent_vertex_set = set(range(numMethod))
        while len(independent_vertex_set) > 0:
            next_independent_vertex_set = set()
            next_row_idx = []
            for i in rowidxs:
                if rows[i] in independent_vertex_set and cols[i] in independent_vertex_set:
                    independent_vertex_set.remove(cols[i])
                    next_independent_vertex_set.add(cols[i])
                    next_row_idx.append(i)
            rowidxs = next_row_idx
            random.shuffle(rowidxs)
            fivs = frozenset(independent_vertex_set)
            rm = set()
            for vs in independent_vertex_set_set:
                if vs < fivs:
                    rm.add(vs)
                if fivs < vs:
                    rm.add(fivs)
            independent_vertex_set_set.add(fivs)
            for vs in rm:
                independent_vertex_set_set.remove(vs)
            independent_vertex_set = next_independent_vertex_set
    return independent_vertex_set_set
```

Figure 4.7: The algorithm for calculating MISs.

likely than methods to change once assigned to a class [88]. The example of MISs that can be obtained from Fig. 4.5(a) or Fig. 4.5(b) is MIS$_1$ = $\{m1, m2, m4, m6, m7\}$ and MIS$_2$ = $\{m1, m4, m5, m6, m7\}$.

Chapter 5. Selection of Refactorings to be Applied



MISs
CCHs and MMs
Object-Oriented Source Code
Entities are mapped into Elementary Refactorings
Transform into Elementary Refactorings
Refactorings' Effect Evaluation Framework
Creating Link Matrix
Creating Membership Matrix
Link Matrix
Membership Matrix
Deriving Delta Table
GER1
R1
R2
R3
GER3
R4
R5
R6
GER4
R7
MISs
CCHs
MMs
Groups of Elementary Refactorings
Delta Table
Accessing Effect of Refactorings
Effect of Refactorings
GER1 GER2 GER3 GER4 GER5
4 2 -2 1 3
Selected Elementary Refactorings
R1
R2
R3
Selecting Multiple Elementary Refactorings
• MM : Move Method refactoring
• CCH : Collapse Class Hierarchy refactoring
• MIS : Maximal Independent Set of refactorings
• GER: Group of Elementray Refactoring

Figure 5.1: Overall approach of the selection of multiple refactorings.

We provide the method of selecting refactorings by supporting assessment and impact analysis of elementary refactorings based on the matrix computation. It is important that the method of elementary-level refactoring computation (our refactoring selection method) enables selecting multiple refactorings. Furthermore, this method supports to extend considering refactorings to other various type of refactorings; because the action of big refactoring (e.g., Collapse Hierarchy Class refactoring) comprises of elementary refactorings (e.g., Move Method refactorings).

The procedure of refactoring selection consists of the several activities: (1) calculation for each of elementary refactoring's effect on maintainability (Delta Table Derivation), (2) checking whether there are duplicated elementary refactorings (i.e., syntactic dependencies) or the RED among refactorings—that are identified as refactoring candidates in chapter 4—(Refactoring Impact Analysis), (3) finding multiple (elementary) refactorings—that can be applied at a same time—containing refactorings that most improve maintainability, and (4) identification of the impacted refactorings after applying selected refactorings (Refactoring Impact Analysis) and recalculation of the changed values for those impacted refactorings. The Fig. 5.1 illustrates the procedure for the approach of the selection of multiple refactorings.

Before explaining about the procedure of refactoring selection, we briefly present the method how to assess refactoring candidates.

Method of Refactoring Candidate Assessment

Before making a decision on which refactorings to apply, refactoring candidates need to be assessed.

In the previous studies, each of the refactoring candidates is assessed by using a simulation model or using a virtual application method. This is because (1) the actual application of the refactoring candidate on source code adds a significant overhead due to disk write operations (once for applying each refactoring and once for undoing it) [87]; in addition, (2) the evaluation of the design quality of maintainability does not necessarily require all the information on the source code. For using a simulation model, a refactoring candidate is applied and the effect of the applied refactoring is assessed by evaluating maintainability on the transformed simulation model (i.e., the refactored model); then the application of the refactoring candidate is rolled back. For instance, in Seng's work [82], they transform the source code into a suitable model using standard fact extraction technique; and they use this model for simulating the source code refactorings and calculating the impact of these refactorings on the fitness function. In our previous study of cost-effective refactoring identification [38], the AOM—the profiled model mentioned in the previous section 4.1.1—is used for this purpose. For using a virtual application method, the effect of a refactoring candidate is estimated (without application) by updating elements that are needed for evaluating maintainability (i.e., calculating fitness function). For instance, in Tsantalis' work [87], to assess the effect of a Move Method refactoring opportunity, they update the entity sets which are involved in the move of the corresponding method.

To evaluate maintainability of the refactored design of software, the quantitative fitness function of maintainability (e.g., QMOOD [8] and Maintainability Index (MI) [69]) is needed. By using the fitness function of maintainability, refactoring candidates are evaluated and ranked in the order of their expected degree of improvement on maintainability. Therefore, the refactoring(s) that mostly improve fitness value can be selected. For instance, for the quantitative fitness function of maintainability, the weighted sum of design metrics are used in Seng's work [82], and the maintainability evaluation function [38]—designed as cohesion metrics over coupling metrics—is used in our previous work. In Tsantalis' work, they use the Entity Placement metric [87].

In this thesis, we provide the method for assessing elementary refactorings by using matrix computation. We derive a delta table (see in 5.1), each cell of which indicates delta of maintainability after the application of each Move Method refactoring on the current design configuration. The matrix computation is used for calculating each of elementary refactoring's effect on maintainability. The maintainability is calculated based on the information of links—e.g., the entity a is associated with the entity b—and memberships—e.g., the entity a is placed in the class A—of the entities. This way of assessing the effect of refactoring is similar to the one using a virtual application method, since it estimates the effect of each of elementary refactoring by using the design status (e.g., links and membership of entities) information without actually applying those refactorings. In addition, maintainability (see subsection 5.1) used in the thesis provides the quantitative measure, therefore, each of elementary refactoring's effect on maintainability can be used for refactoring selection criteria.

5.1 Calculation for Each of Elementary Refactoring's Effect on Maintainability: Refactoring Effect Evaluation Framework

Maintainability

In object-oriented software, two objectives—high cohesion and low coupling—have been accepted as important factors for good software design quality in terms of maintenance [33]; because by adopting these design quality metrics, less propagation of changes to other parts of the system or side effects would occur [10, 6]. *Cohesion* corresponds to the degree to which elements of a class belong together, and *coupling* refers to the strength of association established by a connection from one class to another. To this end, the associations among the entities belonging to a class (inner entities) should be largest as possible (high cohesion). At the same time, the associations between the entities not belonging to a class but to other classes (outer entities) and the entities belonging to the class itself should be smallest as possible (low coupling). Based on this concept, we measure maintainability of the design for overall object-oriented software as the number of the associations across the classes. The associations indicate the edges in the constructed graph G_R that are presented in subsection 4.2.2. Therefore, maintainability can be assessed by the number of edges (i.e., associations) whose two ends of vertices are located in different classes. This number naturally represents the *lack* of degree of association to which entities of a class belong together (lack of cohesion) and, at the same time, the degree of association to which entities of one class to another have (coupling); as a result, by applying refactorings, we aim to reduce this number (for improving maintainability).

Definition of Delta Table

We derive a delta table, each cell of which indicates delta of maintainability after the application of each elementary refactoring on the current design configuration. This delta table is used for refactoring selection criteria. The matrix computation is used for calculating each of elementary refactoring's effect on maintainability. For the delta table, the row elements indicate the moving methods and attributes, while the column elements indicate the target classes. The matrix computation is fast, thus it provides efficient computation for deriving the delta table.

Establishment of Refactoring Effect Evaluation Framework: Derivation of Delta Table

The Fig. 5.2 shows the algorithm for deriving a delta table.

The delta table is calculated as follows. We use the RED-aware graph G_R to form the Link matrix (L). The L denotes the link information from an entity to an entity. Each entity indicates a method or an attribute—which is contained in a program. The value of cell of the L denotes the strength of the relation. When there is an association from an entity (row) to an entity (column), then the value of one is accumulated to the cell of the L is 1; otherwise, when there is no association between two entities, the cell of the L is 0. Note that we distinguish the direction of the edges, therefore, when two entities have associations each other, then the value of cell of the

```
def getInternalExternalLinkMatrix(self):
        internal_link_mask = self.membershipMatrix * self.membershipMatrix.T
        internal_link_matrix = self.linkMatrix.multiply(internal_link_mask)

        external_link_matrix = self.linkMatrix - internal_link_matrix
        return (internal_link_matrix, external_link_matrix)


def invertedMembershipMatrix(self, M):
         new_matrix = zeros((len(self.entities), len(self.classes)), dtype='int32')
        (rows, cols) = M.nonzero()
        for i in range(len(rows)):
            v = M[rows[i], cols[i]]
            new_matrix[rows[i], :] = new_matrix[rows[i], :] + v
            new_matrix[rows[i], cols[i]] = 0


        ret  = sp.coo_matrix(new_matrix)

        return ret


def getEvalMatrix(self):
        (internal_matrix, external_matrix) = self.getInternalExternalLinkMatrix()
        IP = internal_matrix * self.membershipMatrix
        EP = external_matrix * self.membershipMatrix
        IIP = self.invertedMembershipMatrix(IP)
        D = IIP - EP
        return D
```

Figure 5.2: The algorithm for deriving a delta table.

L becomes 2. Since there is no association between fields, the cell of the L is 0. The Membership matrix (M) denotes the membership information of an entity to a class. The cell of the M is 1, when an entity (row) is placed in a class (column); the cell of the M is 0, when the entity is not located in the class. Note that even though we do not consider Move Field refactorings, we need to consider attributes as entities of delta table for calculating delta of maintainability affected by the location of the attributes. When those two matrices, L and M, are multiplied, the Projection matrix (P) is produced. The P represents the link information from an entity (row) to a class (column). For deriving a delta table, we first compute two types of the P: P_{Int} (internal projection matrix) and P_{Ext} (external projection matrix), each of which is computed by the multiplication of the M with L_{Int} (matrix denoting internal links) and L_{Ext} (matrix denoting external links), respectively.

$$P_{Int} = L_{Int} \times M,$$

$$P_{Ext} = L_{Ext} \times M.$$

As was noted in subsection 5.1, the maintainability is assessed by the number of external links—edges (associations) whose two ends of vertices (entities) are located in different classes—in the system, and this number should be reduced for improving maintainability. The cell of the P_{Int} is 1, when the internal link exists from the entity (row) to the class (column). This means that moving the entity to other classes (other than the class itself) will *potentially* increase the external links in the system. To this end, we use $Inv()$ function for P_{Int}. The $Inv()$ function inverses the cell of P_{Int}(entity, class itself) as $1 \rightarrow 0$ and P_{Int}(entity, other classes) as $0 \rightarrow 1$. The cell having 1 in P_{Ext} means that the external link exists from an entity (row) to a class (column). This means that moving the entity to the class itself will decrease the external links in the system. Finally, by using the formulation below, we can get the delta table (D) of which each cell is delta of maintainability value after application of each Move Method refactoring on the design.

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

5.2 Refactoring Impact Analysis

Refactoring impact analysis is used in (a) checking whether there are duplicated elementary refactorings or the RED among refactorings, and (b) identifying of the impacted refactorings after applying refactorings for updating changed maintainability values.

The refactoring impact analysis of (a) is done when they attempt to grouping refactorings—that can be applied at a same time—(to make a more bigger refactoring). For this, refactorings are projected into elementary refactorings (i.e., Move Method refactorings). Then, among the elementary refactorings, we check whether there are duplicated elementary refactorings (i.e., syntactic dependency) or the RED.

The refactoring impact analysis of (b) is done for identifying of the impacted refactorings after applying refactorings for updating changed maintainability values. In the internal projection matrix (P_{Int}) and the external projection matrix (P_{Ext}), the following cells are changed: classes that are changed in M (i.e., owner class of moving method and target class) and linked methods with the moving method. Therefore, only those changed cells need to be recalculated.

5.3 Selection of Multiple Refactorings

We select the group of refactorings—that can be applied at the same time—containing the multiple elementary refactorings that best improves maintainability. The refactorings are selected from the identified refactoring candidates—(1) Collapse Class Hierarchy refactorings and (2) Move Method refactorings (in section 4.1) and 3) MISs (groups of elementary (i.e., Move Method) refactorings) (in section 4.2). For selecting multiple refactorings, accumulated values of delta of maintainability for each group of refactoring candidates are prioritized in the descending order, and the group of refactoring candidates which has the largest value are selected.

Using the delta table, methods involved in each MIS are transformed into a group of elementary refactorings. For instance, for each method m in a MIS, the Move Method refactoring that has the largest delta of maintainability is mapped out of all available Move Method refactorings (method m, class c), where $c \neq$ owner class of method

m and $c \in$ classes in the system. In short, the Move Method refactoring is determined by comparing the values of delta of maintainability for the row of entity of method m in the delta table.

After selection, we recalculate the changed values of impacted elementary refactorings for updating the delta table. In addition, the M and L matrices are updated. The procedure of refactoring identification is iterated until no more group of refactoring candidates for improving maintainability are found.

It is important to note that before selecting multiple refactorings, we check the *specialization ratio* (S) [44], which is a measure used to prevent merging of too many classes together and getting the class hierarchy wider. S is formulated as follows: ($\sharp$ of classes - $\sharp$ of root classes) / ($\sharp$ of classes - $\sharp$ of leaf classes), where the root classes are the distinct class hierarchies, and the leaf classes are the ones from which the other classes do not inherit. S measures the width of the inheritance tree; in other words, S is the average number of derived classes for each base class. Therefore, a higher value indicates a wider tree. If the S of the refactored model exceeds specific threshold γ, then an alternative refactoring (for example, a refactoring with the second-largest fitness value) is selected.

Example

By following the procedure explained above, Fig. 5.3 illustrates how to obtain the delta table (Fig. 4.5(c)) for the corresponding design (Fig. 4.5(a)). Note that Fig. 5.4 represents the matrices required to calculate the delta table (Fig. 4.5(c)).

P_{Int}	A	B	C	D
m1	0	1	0	0
m2	0	0	0	0
m3	0	1	0	0
m4	0	0	0	0
m5	0	0	0	0
m6	0	0	0	0
m7	0	0	0	0

=

L_{Int}	m1	m2	m3	m4	m5	m6	m7
m1	0	0	1	0	0	0	0
m2	0	0	0	0	0	0	0
m3	1	0	0	0	0	0	0
m4	0	0	0	0	0	0	0
m5	0	0	0	0	0	0	0
m6	0	0	0	0	0	0	0
m7	0	0	0	0	0	0	0

X

M	A	B	C	D
m1	0	1	0	0
m2	1	0	0	0
m3	0	1	0	0
m4	0	0	1	0
m5	0	0	0	1
m6	1	0	0	0
m7	1	0	0	0

(a) $P_{Int} = L_{Int} \times M$

P_{Ext}	A	B	C	D
m1	0	0	0	0
m2	0	1	0	1
m3	3	0	1	0
m4	0	1	0	0
m5	1	0	0	0
m6	0	1	0	0
m7	0	1	0	0

=

L_{Ext}	m1	m2	m3	m4	m5	m6	m7
m1	0	0	0	0	0	0	0
m2	0	0	1	0	1	0	0
m3	0	1	0	1	0	1	1
m4	0	0	1	0	0	0	0
m5	0	1	0	0	0	0	0
m6	0	0	1	0	0	0	0
m7	0	0	1	0	0	0	0

X

M	A	B	C	D
m1	0	1	0	0
m2	1	0	0	0
m3	0	1	0	0
m4	0	0	1	0
m5	0	0	0	1
m6	1	0	0	0
m7	1	0	0	0

(b) $P_{Ext} = L_{Ext} \times M$

D	A	B	C	D
m1	1	0	1	1
m2	0	-1	0	-1
m3	-2	0	0	1
m4	0	-1	0	0
m5	-1	0	0	0
m6	0	-1	0	0
m7	0	-1	0	0

=

Inv(P_{Int})	A	B	C	D
m1	1	0	1	1
m2	0	0	0	0
m3	1	0	1	1
m4	0	0	0	0
m5	0	0	0	0
m6	0	0	0	0
m7	0	0	0	0

−

P_{Ext}	A	B	C	D
m1	0	0	0	0
m2	0	1	0	1
m3	3	0	1	0
m4	0	1	0	0
m5	1	0	0	0
m6	0	1	0	0
m7	0	1	0	0

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

Figure 5.3: Example of calculating the delta table (D) of Fig. 4.5(c) for the design of Fig. 4.5(a).

L_{Int}	m1	m2	m3	m4	m5	m6	m7
m1	0	0	1	0	0	0	0
m2	0	0	0	0	0	0	0
m3	1	0	0	0	0	0	0
m4	0	0	0	0	0	0	0
m5	0	0	0	0	0	0	0
m6	0	0	0	0	0	0	0
m7	0	0	0	0	0	0	0

(a) Internal link matrix (L_{Int}).

L_{Ext}	m1	m2	m3	m4	m5	m6	m7
m1	0	0	0	0	0	0	0
m2	0	0	1	0	1	0	0
m3	0	1	0	1	0	1	1
m4	0	0	1	0	0	0	0
m5	0	1	0	0	0	0	0
m6	0	0	1	0	0	0	0
m7	0	0	1	0	0	0	0

(b) External link matrix (L_{Ext}).

M	A	B	C	D
m1	0	1	0	0
m2	1	0	0	0
m3	0	1	0	0
m4	0	0	1	0
m5	0	0	0	1
m6	1	0	0	0
m7	1	0	0	0

(c) Membership matrix (M).

P_{Int}	A	B	C	D
m1	0	1	0	0
m2	0	0	0	0
m3	0	1	0	0
m4	0	0	0	0
m5	0	0	0	0
m6	0	0	0	0
m7	0	0	0	0

(d) Internal projection matrix (P_{Int}).

Inv(P_{Int})	A	B	C	D
m1	1	0	1	1
m2	0	0	0	0
m3	1	0	1	1
m4	0	0	0	0
m5	0	0	0	0
m6	0	0	0	0
m7	0	0	0	0

(e) Inversion of internal projection matrix (Inv(P_{Int})).

P_{Ext}	A	B	C	D
m1	0	0	0	0
m2	0	1	0	1
m3	3	0	1	0
m4	0	1	0	0
m5	1	0	0	0
m6	0	1	0	0
m7	0	1	0	0

(f) External projection matrix (P_{Ext}).

D	A	B	C	D
m1	1	-	1	1
m2	-	-1	0	-1
m3	-2	-	0	1
m4	0	-1	-	0
m5	-1	0	0	-
m6	0	-1	0	0
m7	0	-1	0	0

(g) Delta Table.

Figure 5.4: Matrices matrices required to calculate the delta table (Fig. 4.5(c)).

Chapter 6. Tool Implementation

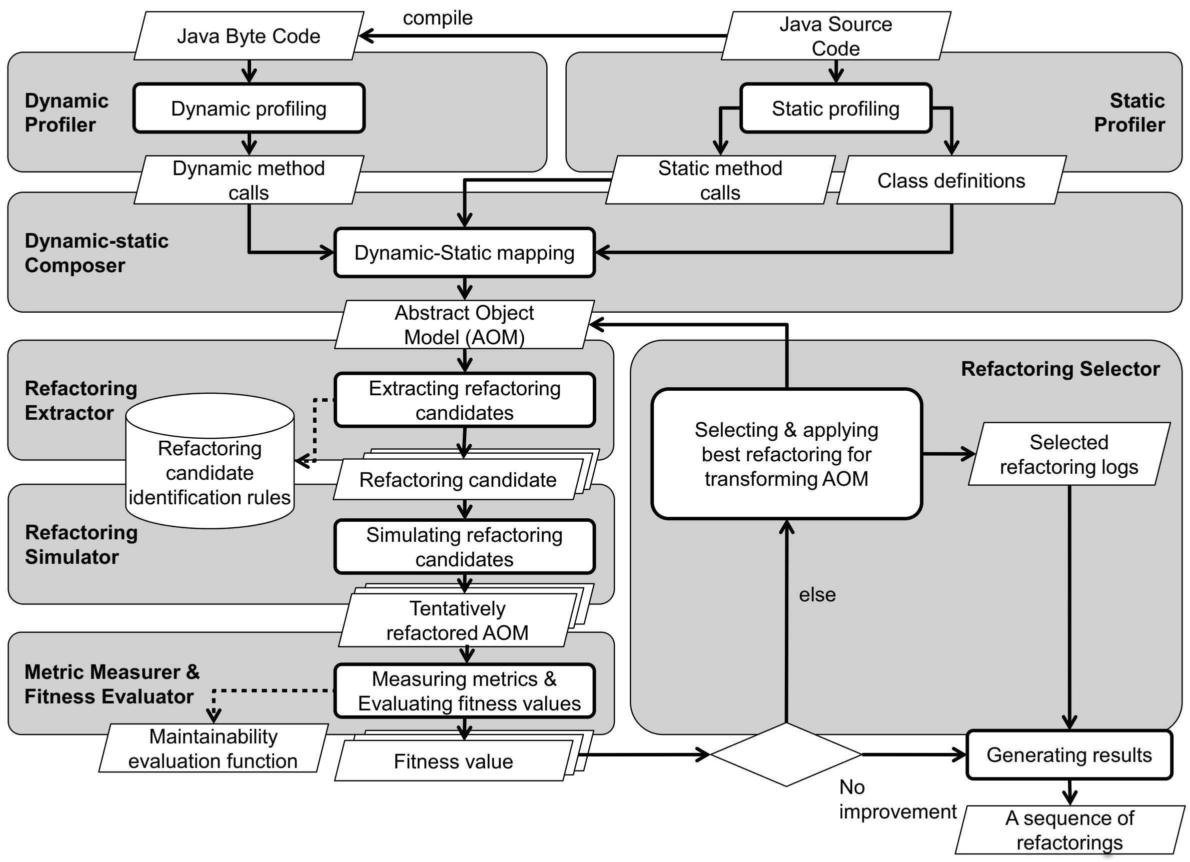
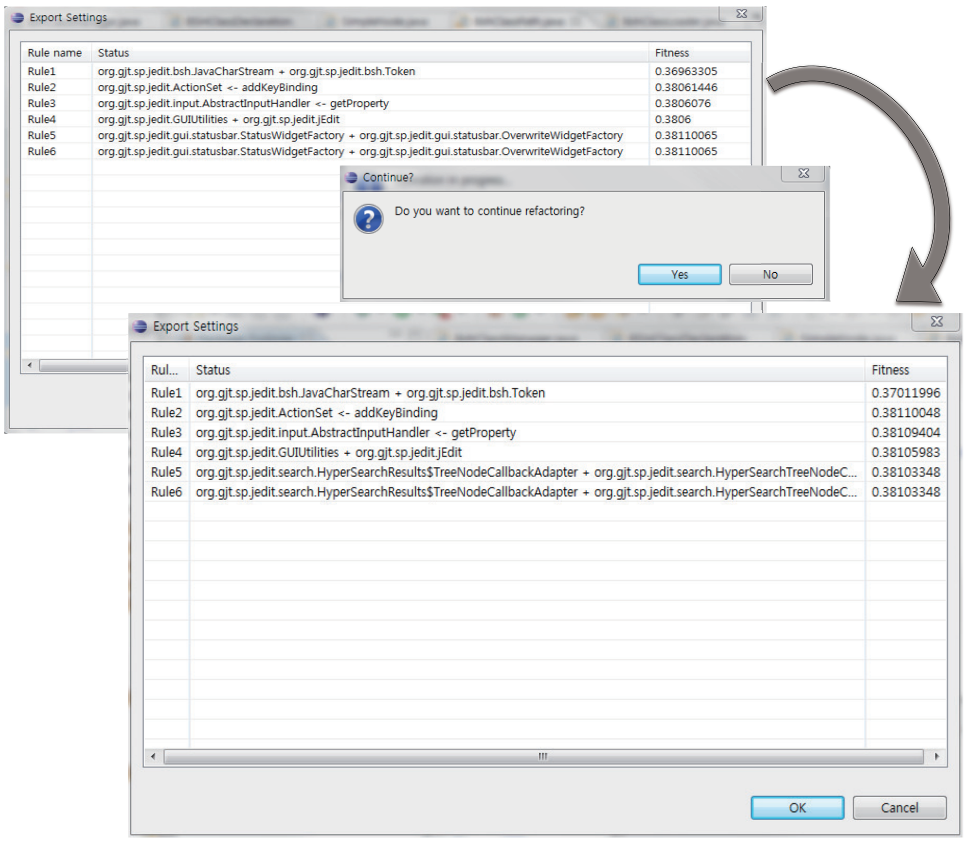

Java Byte Code
compile
Java Source Code
Dynamic Profiler
Dynamic profiling
Static profiling
Static Profiler
Dynamic method calls
Static method calls
Class definitions
Dynamic-static Composer
Dynamic-Static mapping
Abstract Object Model (AOM)
Refactoring Extractor
Extracting refactoring candidates
Refactoring candidate identification rules
Refactoring candidate
Refactoring Simulator
Simulating refactoring candidates
Tentatively refactored AOM
Metric Measurer & Fitness Evaluator
Measuring metrics & Evaluating fitness values
Maintainability evaluation function
Fitness value
Refactoring Selector
Selecting & applying best refactoring for transforming AOM
Selected refactoring logs
else
No improvement
Generating results
A sequence of refactorings

Figure 6.1: Overall tool architecture.

The proposed method has been implemented [37] using Java with Eclipse environment. Fig. 6.1 illustrates the overall tool architecture. The following main modules comprise the tool: static profiler, dynamic profiler, dynamic-static composer, refactoring simulator, metric measurer, fitness evaluator, and refactoring selector. In the static profiler, given the Java source code, code structure information such as SMCs and class definitions are extracted. On the other hand, in the dynamic profiler, DMCs are extracted by executing a Java byte code compiled from the Java source code using user scenarios or operational profiles. In the dynamic-static composer, the DMCs are mapped into corresponding SMCs from which those DMCs are instantiated, and the base Abstract Object Model (AOM) is constructed. More detailed explanation of AOM is provided in subsection 4.1.1. In the refactoring extractor, refactoring candidates are extracted using refactoring-candidate extraction rules. In the refactoring simulator, refactoring candidates are applied by transforming base AOM, and tentatively refactored AOMs are produced. In the metric measurer and the fitness evaluator, for all tentatively refactored AOMs, metrics are derived and used to calculate the fitness value of the maintainability evaluation function. In the refactoring selector, if no more refactoring candidates for improving fitness values are found, the tool is stopped, and it generates the selected refactoring logs as output indicating a sequence of recommended refactorings. Otherwise, the refactoring that makes the refactored AOM with the best fitness value is selected in a greedy way and applied, then the base AOM is updated into the refactored AOM, only when the specialization ratio of the refactored AOM does not exceeds the specific threshold γ. After that, the procedure of the refactoring extractor, the refactoring simulator, the metric measurer, the fitness evaluator, and the refactoring selector are iterated. In addition to the best selection mode, the tool can be operated in a user-interactive mode. In user-interactive mode, users can select the preferred refactoring. Fig. 6.2 shows a snapshot of the tool operation.



Rule name	Status	Fitness
Rule1	org.gjt.sp.jedit.bsh.JavaCharStream + org.gjt.sp.jedit.bsh.Token	0.36963305
Rule2	org.gjt.sp.jedit.ActionSet <- addKeyBinding	0.38061446
Rule3	org.gjt.sp.jedit.input.AbstractInputHandler <- getProperty	0.3806076
Rule4	org.gjt.sp.jedit.GUIUtilities + org.gjt.sp.jedit.jEdit	0.3806
Rule5	org.gjt.sp.jedit.gui.statusbar.StatusWidgetFactory + org.gjt.sp.jedit.gui.statusbar.OverwriteWidgetFactory	0.38110065
Rule6	org.gjt.sp.jedit.gui.statusbar.StatusWidgetFactory + org.gjt.sp.jedit.gui.statusbar.OverwriteWidgetFactory	0.38110065

Do you want to continue refactoring?

Rul...	Status	Fitness
Rule1	org.gjt.sp.jedit.bsh.JavaCharStream + org.gjt.sp.jedit.bsh.Token	0.37011996
Rule2	org.gjt.sp.jedit.ActionSet <- addKeyBinding	0.38110048
Rule3	org.gjt.sp.jedit.input.AbstractInputHandler <- getProperty	0.38109404
Rule4	org.gjt.sp.jedit.GUIUtilities + org.gjt.sp.jedit.jEdit	0.38105983
Rule5	org.gjt.sp.jedit.search.HyperSearchResults$TreeNodeCallbackAdapter + org.gjt.sp.jedit.search.HyperSearchTreeNodeC...	0.38103348
Rule6	org.gjt.sp.jedit.search.HyperSearchResults$TreeNodeCallbackAdapter + org.gjt.sp.jedit.search.HyperSearchTreeNodeC...	0.38103348

Figure 6.2: A snapshot of tool operation.

Chapter 7. Evaluation

In the first subsection, research questions that we investigate in this experiment are presented. In the second subsection, the experimental subjects and data processing method for those subjects are explained. In the third and fourth subsections, the evaluation design is explained, and the results are presented, respectively. The final subsection ends with threats to validity.

7.1 Research Questions

We evaluate our approach of refactoring identification in two aspects: dynamic information-based identification of refactoring candidates and RED-aware selection of multiple refactorings. The research questions for our experiments are as follows. The first research question is to test the usefulness of the overall approach by using dynamic information in identifying refactoring candidates, while the second research question is to test the capability of the approach in extracting refactoring candidates. The third and fourth research questions are to test the effect of using Groups of Elementary Refactorings (MISs) and the effect of the RED, respectively.

RQ1. Effect of dynamic information for effective refactoring identification

Is the dynamic information helpful in identifying refactorings that effectively improve maintainability?

RQ2. Effect of dynamic information for extracting refactoring candidates in frequently changed classes

Is dynamic information helpful in extracting refactoring candidates in the classes where real changes had frequently occurred?

RQ3. Effect of selection for multiple refactorings

Do the multiple refactorings help to improve maintainability and reduce the costs of search space exploration and computation?

RQ4. Effect of the RED

Is the RED important when grouping entities into MISs for improving maintainability?

7.2 Subjects and Data Processing

Three projects are chosen for experimental subjects: jEdit [49], Columba [20], and JGIT [52]. A number of reasons led us to select these subjects.

- The full source code of each version is available.
- They contain a relatively large number of classes.

Table 7.1: Characteristics and development history for each subject.

Name	jEdit	Columba	JGIT
Type	Text editor	Email client	Distributed source version control system
Total ♯ of revisions	19501	458	1616
Report period	2001-09 ∼ 2011-09	2006-07 ∼ 2011-07	2009-09 ∼ 2011-09
Number of developers	25	9	9
Version to apply the proposed approach	jEdit-4.3	Columba-1.4	v1.1.0.201109151100-r

Table 7.2: Measures of each subject.

Name	jEdit (jEdit-4.3)	Columba (Columba-1.4)	JGIT (jGit-1.1.0)
Class ♯	952	1506	689
Method ♯	6487	8745	5334
Attribute ♯	3523	3967	2989
Link ♯	26626	23981	18280
Fitness function value	0.023287	0.023117	0.021357

- They are written in Java; our proposed method applies to object-oriented software.
- Their development histories are well-managed in version-control systems.

Table 7.1 summarizes characteristics and development histories of each subject. In addition, measures of each subject—the classes, method, attributes, links (i.e., associations), and fitness function values—are presented in Table 7.2.

To apply the proposed approach on each subject, one version of the source code is selected as input data (as the last row in Table 7.1). It is important to mention that we select a version after which major changes have occurred. We also do not select the early version because, at that time, the software is unstable, and meaningless changes may occur frequently. In short, we take into account a mature version.

Performing the Dynamic Profiling

To obtain dynamic information of the dependencies of entities, dynamic profiling is performed by executing the selected version of the program of each subject according to its user scenarios or operational profiles. In this experiment, the user scenarios or operational profiles data are not available, since the subjects are chosen from open source projects. Therefore, the dynamic information is obtained by executing the programs from various users who exhibit normal behavior in using the programs. To obtain more reliable dynamic profiling results, we set specific criteria for use of software regarding characteristics of users, experimental environment, or experimental conditions. Note that we do not take into account abnormal or extraordinary scenarios, because they may result in suggesting refactorings in parts not actually in use. For example, for jEdit, we do not log the bootstrapping part of the editor; we profile only the editing part. Similarly, for Columba, we do not log the initializing part of the e-mail client, but only functions such as retrieving messages from a mailbox, composing messages, and submitting messages to a server. In the following, for each subject, we present the criteria and the rationale for using it.

jEdit [49] is a java-based text editor which is developed for using the same editor on different platforms or operating systems. It also provides very common graphic user interfaces like other text editors. Since it provides only one interface, GUI, we set the criteria of the experimental condition: characteristics of language (natural language and formal language) and length of written text (long and short). To detect typos while editing the natural language, the text editor should search an entire dictionary that is rather big; while a typo on the formal language (e.g., programming language) can be relatively easily detected. On the other hand, for a long description, the users tend to change the structure of the description while writing the description. However, for a short description, like a short e-mail message, the description is written without revision. The dynamic profiling was conducted as follows.

- Long description with formal language: C and python server code, 2 days, 1 man.
- Short description with formal language: html, python, 1 day, 2 men.
- Long description with natural language: latex, 2 days, 1 man.
- Short description with natural language: E-mail message, 1 day, 2 men.

Columba [20] is an E-mail client program implemented using Java. It supports standard protocols—POP3 and IMAP4—for e-mail clients and provides usual GUI features, such as showing the list of received and sent mails and e-mail composition. According to the interfaces which Columba has, we set the criteria of the experimental condition: network protocol and GUI. For the network protocol, we take into account POP3 and IMAP4. As we mentioned above, the GUI is composed of three common mail client actions, and we distinguish read-intensive users and write-intensive users. By observing the usage pattern of the users, we found that the graduate school students tended to be read-intensive users, while the business people tended to be write-intensive users, relatively. Therefore, we chose the two user groups for realizing the experiment condition. The dynamic profiling was conducted for four days with six graduate school students and six business men working in a venture company. Since all of them used the G-mail, they used the Columba with the POP3 on first two days and then they used it with the IMAP4 on next two days.

JGIT [52] is a java implementation of git, which is a well-known distributed version control system. The git provides very complex version control operations, such as three-way merging, cherry-picking, and rebase. Moreover, the git uses various protocols: https, ssh, or git. Among those functionalities, the JGIT does not provide all of them, but only provide core functionalities: clone, push, commit, fetch or branch. In fact, usage pattern of JGIT may be varied by the habit of the user. For example, some developers prefer the small commit and big push, while some other developers prefer the big commit and big push, and few developers prefer the small commit and small push. On the other hand, some users does not use commit or push functionalities, but only use clone and fetch—of course, these developers do not use branch. The dynamic profiling was conducted for three days by three developer and one manager of the venture company as follows.

- Each of three developers has characteristics as follows:
  - Domain: server side; language: Erlang and Python; commit interval: short; push interval: short; clone or pull interval: rare
  - Domain: client side (iOS); language: objective-C and C; commit interval: short; push interval: long; clone or pull interval: rare
  - Domain: client side (Android); language: Java and C; commit interval: long; push interval: long; clone or pull interval: rare
- The manager's characteristics are as follows:
  - Domain: server, client, and library side; language: Java, objective-C and C; commit interval: long; push interval: long; clone or pull interval: short.

Extracting Changes

Table 7.3: Examined range of revisions.

jEdit	Columba	JGIT
$18000 \sim 19000$	$300 \sim 450$	$1 \sim 1616$

For each subject, real changes—that had occurred within the examined revisions of the development history—are extracted. We examined the revisions (as in Table 7.3) that had been made after the selected version. The changed methods include method body changes as well as method signature changes, such as changes in name, parameter, visibility, and return type.

To test hypothesis 1, the changed methods across the revisions are added to the list of changed methods, which is used as the input for change impact analysis. Note that the methods in the list of changed methods can be redundant to take into account the effect of their frequency of occurrences. To test hypothesis 2, the set of changed classes—which are the owner classes of those changed methods—and the corresponding number (i.e., frequency) of changes for those classes are used to be compared with extracted classes as refactoring candidates. The procedure used for extracting the list of changed methods is as follows. First, we retrieve the source code in which each revision occurred. Next, we analyze files in the source code for each revision to obtain the following information.

abstract_java abstract_java is a function such that:

(rev_number, file_name) $\rightarrow$ a set of file_info_entry

file_info_entry file_info_entry is a tuple such that:

(start_line_number, end_line_number, class_name, method_name)

Then, we use Diff and obtain changed line numbers.

line_change line_change is a function such that:

(former_rev_number, latter_rev_number, file_name) $\rightarrow$ changed_line_number

Finally, using these changed line numbers, we can obtain the changed methods which had been changed across the revisions.

7.3 Evaluation Design

RQ1: Effect of Dynamic Information for Cost-Effective Refactoring Identification

Before explaining the experimental design of RQ1, we clearly provide the definition of cost-effective refactorings. Cost-effectiveness of applied refactorings can be explained as maintainability improvement over invested refactoring effort (cost). Therefore, it can be said that refactoring X is more cost-effective than refactoring Y, when maintainability improvement in relation to the invested effort of applying refactoring X is larger than that of applying refactoring Y. By using cost-effective refactorings, the less effort of applying refactorings is required to accomplish the same maintainability improvement.

For RQ1, which aims to investigate whether dynamic information is helpful in identifying cost-effective refactorings that fast improve maintainability and lead to high rate of maintainability improvement, we use the method of change simulation. We compare the results of change simulation to observe how quickly the number of propagated changes is reduced on the refactored models whose applied refactorings are identified using the following three comparison groups.

1. The approach using dynamic information only (group 1)
2. The approach using static information only (group 2)
3. The combination of the two approaches (group 3)

For each subject, refactorings are identified from three comparison groups as follows. (1) In group 1, for each iteration of refactoring identification process, 180 refactoring candidates (i.e., 18 rules [6 types of scoring functions $\times$ 3 types of refactorings] $\times$ 10 top refactoring candidates) are assessed. In this experiment, the $cutline$ number—the threshold number for limiting the consideration set of refactoring candidates—is set to 10. The best refactoring is selected and applied. We continue to perform the refactoring identification process until no more refactoring candidates for improving maintainability are found. At last, we obtain a sequence of refactorings. (2) Group 2 follows the same approach of the group 1 but substitutes the DMCs with SMCs (i.e., using static measures instead of dynamic measures) in the refactoring-candidate extraction rules and the maintainability evaluation function for extracting and assessing refactoring candidates, respectively. (3) In group 3, the considered refactoring candidates are the ones extracted from both approaches—refactoring candidates from group 1 $\bigcup$ refactoring candidates from group 2—and the best refactoring is selected and applied in an iterative way (with the same method of our study). The approach of group 2 is to test the effect of using dynamic information, and the approach of group 3 is to test whether the dynamic information can be additional or complementary information to the static information. Note that the aim of this test is not to compare the performance of the approach of using dynamic information versus the approach of using static information but to investigate the effect of using dynamic information for identifying cost-effective refactorings.

Change simulation. To assess the capability of refactorings for maintainability improvement, we use the *change simulation* method. The basic idea is to inject changes—extracted as real changes that have occurred during software maintenance—and then obtain propagated changes by performing change impact analysis. We call the former change the *original change* and the latter change the *propagated change*. This evaluation method is based on the belief that propagated changes indicate how the design can withstand original changes; in other words, the more easily the design accommodates changes, the fewer propagated changes will occur. Therefore, the capability of refactorings for maintainability improvement is measured by the reduced number of propagated changes. For performing change simulation, the list of changed methods (explained in subsection 7.2) is used as original changes (input for change impact analysis). Then, change impact analysis is performed on each of the refactored model—produced by every application of a sequence of refactorings—to obtain propagated changes (output for change impact analysis). Change impact analysis is the method to identify the potential consequences for a change; therefore, the propagated changes are computed by taking the directly and indirectly affected methods from the method. Change impact analysis used in the experiment is implemented as follows. For each method in the list of changed methods, the propagated methods, that refer to this method but are defined in other classes, are retrieved. Note that the change impact analysis is performed in the batch processing mode. Subsequently, the propagated methods or classes—the owners of the propagated methods—are *accumulated* for all the methods in the list of changed methods. The two-steps of indirect propagated methods are considered using the weight value of 0.5, while the direct propagated methods use the weight value of 1.

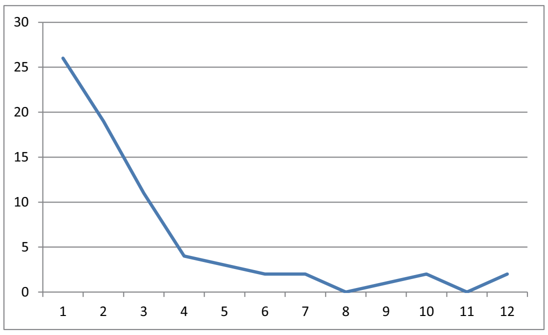
Two indicators for cost-effective refactorings. The cost-effectiveness of the identified refactorings can be evaluated by observing how fast the number of propagated changes is reduced. In our approach, we assume that invested refactoring effort (cost) is the number of applied refactorings. We notice the limitation of estimating refactoring cost as the number of applied refactorings; to more accurately estimate refactoring cost, we need to consider the effort needed to perform the activities of the entire refactoring process. We deal with this issue in the discussion chapter 8. Two indicators are used: (1) the percentage of reduction for propagated changes and (2) the rate of reduction for propagated changes. The indicators can be calculated as follows. Let r_n represent the n^{th} applied refactoring. Let the number of propagated changes for accommodating changes on the initial profiled model be ic_0, ic_{last} on the design applying all the identified refactorings from the first to the last refactoring, and ic_n on the design applying a sequence of identified refactorings r$_1$, ..., r$_n$. The percentage of reduction for propagated changes ($percntRPC$) of r$_n$ is as below.

$$percntRPC(r_n) = \frac{ic_0 - ic_n}{ic_0 - min\{ic_{last}\}} \times 100,$$

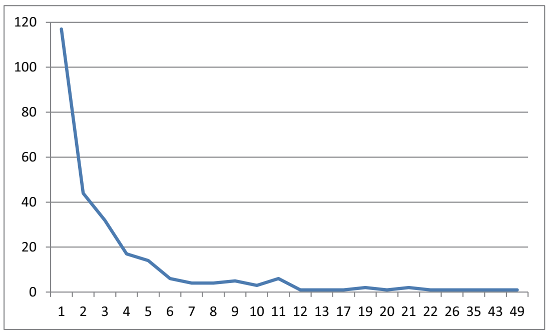
where $min\{x\}$ returns the minimum number of propagated changes among all comparison groups, which is needed for normalization. The rate of reduction for propagated changes ($rateRPC$) between r$_m$ and r$_n$, when r$_m$ precedes r$_n$, is calculated by the differences of percentage of reduction for propagated changes over the number of applied refactorings as below.

$$rateRPC(r_m, r_n) = \frac{percntRPC(r_n) - percntRPC(r_m)}{\sharp\ of\ applied\ refactorings\ between\ r_n\ and\ r_m.}$$

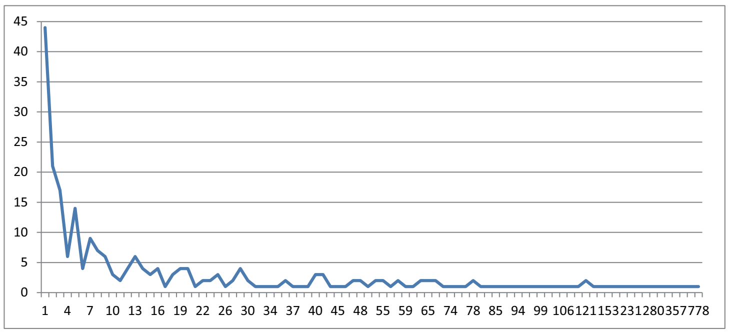
In this experiment, the rate of reduction for propagated changes is considered for every applied refactoring; therefore, it can be represented as to $percntRPC(r_n) - percntRPC(r_{n-1})$, since the number of applied refactorings between refactorings is 1.



(a) jEdit



(b) Columba



(c) JGIT

Figure 7.1: Change distribution graph (X-axis: $\sharp$ of occurred changes for each class, Y-axis: $\sharp$ of corresponding classes)

RQ2: Effect of Dynamic Information for Extracting Refactoring Candidates in Frequently Changed Classes

The underlying assumption of our approach is that changes are more prone to occur in the pieces of codes the users more often utilize and that, hence, applying refactorings in these parts would fast improve maintainability of software. For this reason, we extract refactoring candidates in the entities—involved in given scenarios/functions of a system—in a way that reduces dependencies of those entities.

To validate the assumption, we test RQ2, which aims to investigate whether dynamic information is helpful in extracting refactoring candidates in the classes where real changes had frequently occurred. For each subject, we compare a) the classes of the top 10%, 20%, 30% and 100% (i.e., all changes) most frequently changed during the real development history (explained in subsection 7.2) with b) the classes of refactoring candidates extracted from the approach using dynamic information and the approach using static information, respectively, to observe how many classes extracted as refactoring candidates are found in real changed classes. In addition to all changes, we consider top $k\%$ (k = 10, 20, 30) most frequently changed classes to examine the capability of each approach for extracting refactoring candidates from highly-ranked frequently changed classes. We have considered up to the top 30% most frequently changed classes, because, for three subjects—jEdit, Columba, and JGIT—most of the changes occur in those changed classes (see Fig. 7.1). For instance, the ratio of the number of occurred changes in the top 30% most frequently changed classes over the total number of occurred changes in changed classes is 143 out of 207 $\approx$ 70%, 788 out of 993 $\approx$ 80%, and 8754 out of 9773 $\approx$ 90%, for jEdit, Columba, and JGIT, respectively. Note that, as with RQ1, the aim of this test is not to compare the performance of the approach of using dynamic information versus the approach of using static information but to investigate the effect of using dynamic information for extracting refactoring candidates in the frequently changed classes.

We use the classes of refactoring candidates that are obtained from group 1 (i.e., the approach using dynamic information only) and group 2 (i.e., the approach using static information only) in RQ1. The lists of the extracted classes as refactoring candidates for each approach and the list of real changed classes are ranked in a descending order according to the number of occurred changes for each class. Let the ranked list of the top $k\%$ most frequently changed classes be $rankKChanged$, the ranked list of the classes extracted as refactoring candidates for group 1 be $rankDynamic$, and the ranked list of group 2 be $rankStatic$. By comparing $rankKChanged$ with $rankStatic$ and $rankDynamic$, we first obtain (1) commonly found classes (i.e., intersect set). We then obtain (2) the two distance measures (K: Kendall's tau, F: Spearman's footrule [27]), which are the measures for comparing similarity of two top k lists.

RQ3: Effect of Multiple Refactorings

To investigate the effect of multiple refactorings (RQ3), we compare our approach—whose refactoring candidates are from both top-down and bottom-up approaches, (1) Collapse Class Hierarchy refactorings (i.e., big refactorings) and Move Method refactorings identified from the rule-based identification and (2) MISs, which should be supported on assessment and impact analysis of elementary refactorings—with the approach whose refactorings candidates are only (1). To evaluate the capability of the selected refactorings for maintainability

$$fitness = \frac{\dfrac{MSC_{avg} - MSC_{min}}{MSC_{max} - MSC_{min}}}{w_1\dfrac{DC_{avg} - DC_{min}}{DC_{max} - DC_{min}} + w_2\dfrac{SC_{avg} - SC_{min}}{SC_{max} - SC_{min}} + w_3\dfrac{DC^{S}{}_{avg} - DC^{S}{}_{min}}{DC^{S}{}_{max} - DC^{S}{}_{min}} + w_4\dfrac{SC^{S}{}_{avg} - SC^{S}{}_{min}}{SC^{S}{}_{max} - SC^{S}{}_{min}}}$$

Figure 7.2: Maintainability evaluation function for producing $fitness$ value.

improvement, we use the maintainability evaluation function [38] as a fitness function to measure maintainability of the refactored design.

Maintainability evaluation function. In search-based approaches, to combine multiple objectives into a single-objective function, methods such that (1) metrics for each objective are normalized, weighted, and added up [81, 61], or (2) Pareto optimality [43], are used. We adopt the former approach for conflating two objectives of metrics—high cohesion and low coupling—into a single fitness function. We design the maintainability evaluation function as (cohesion / coupling), because the maintainability evaluation function of this design produces larger fitness values as the software gets more maintainable (with higher cohesion and lower coupling). In addition, the two objectives may conflict in many cases, and the maintainability evaluation function of this design prevents merging of unrelated units of codes, which reduces couplings but lowers cohesion.

Fig. 7.2 shows the formulation of the maintainability evaluation function, which produces the fitness value of the refactored model. Each metric is normalized in the following way: the difference between the average and minimum values is divided by the difference between the maximum and minimum values of the metric. The average value of the metric is obtained by summing all the values of the classes and dividing this by the number of classes. For composing all coupling metrics, weight values, whose total sum is one, are multiplied to each normalized coupling metric, then all the normalized coupling metrics are added up. Note that by using the weight values, a user can decide to focus on certain aspects of the maintainability evaluation function. In our approach, we assign a weight value of 0.25 for each coupling metric.

In the following, each metric—constituting the maintainability evaluation function—is explained. For cohesion, the Method Similarity Cohesion (MSC) [11] metric is used. In this metric, the similarity for all pairs of methods are integrated and normalized to measure how cohesive the class is. Its difference from the other cohesion metrics is that it considers degree of similarity between a pair of methods in a class. For instance, Lack of Cohesion in Methods (LCOM) [18] does not account for the degree of similarity between methods; instead, it categorizes the sets into two groups—empty and non-empty—and produces the same results for a pair of methods whether it has one instance variable or all instance variables shared in common. Another cohesion metric, Cohesion Among Methods in Class (CAMC) [7], is not considered, because this metric only deals with the parameter types (not usage of instance variables or methods). MSC for a class C is calculated as follows:

$$MSC(C) = \frac{2}{n(n-1)} \sum_{i=1}^{\frac{n(n-1)}{2}} \frac{IV_c}{IV_t} i,$$

where class C has n methods, and for a pair of methods, IV_c and IV_t stand for the common (i.e., intersect set) and total instance (i.e., union set) variables used by the pair of methods repeatedly. Since there are $\frac{n(n-1)}{2}$ distinct combinations of pairs of methods in a class, i ranges from 1 (i.e., first pair) to $\frac{n(n-1)}{2}$ (i.e., last pair), and $\frac{|IV|_c}{|IV|_t}i$ indicates the similarity of the pair of methods, respectively.

For coupling, four coupling metrics—defined based on the DMCs as well as the SMCs—are used. The size of the DMCs in both directions for all methods in a class C is defined as Dynamic Coupling (DC). DC can be specified as

$$DC(C) = \sum_{m_i} |DMC(m_i, \prec)| + |DMC(m_i, \succ)|,$$

where every method m in class C. In the same way, the size of SMCs, measured on a method between caller and callee classes, in both directions for all methods in a class C, is defined as Static Coupling (SC); SC can also be specified based on SMCs. Let $SMC(\varepsilon, \delta)$ denote the list of SMCs retrieved in respect to the entity ε and the direction δ likewise $DMC(\varepsilon, \delta)$. Then, SC can be specified as

$$SC(C) = \sum_{m_i} |SMC(m_i, \prec)| + |SMC(m_i, \succ)|,$$

where every method m in class C. The modified versions of DC and SC are defined and named DC^S and SC^S by converting from lists into the set of DMC and SMC. In other words, redundant elements are eliminated from the lists of DMC and SMC for degrading the effect of strength of dependencies; therefore, only distinct elements remain in the set of DMC^S and SMC^S. Each of the defined coupling DC^S and SC^S is specified as follows.

$$DC^S(C) = \sum_{m_i} |DMC^S(m_i, \prec)| + |DMC^S(m_i, \succ)|,$$
$$SC^S(C) = \sum_{m_i} |SMC^S(m_i, \prec)| + |SMC^S(m_i, \succ)|.$$

It is worth to mention that we have considered four couplings which capture 8 types of combination: (dynamic method call vs. static method call) $\times$ (import direction vs. export direction) $\times$ (distinct methods [set] vs. all invoked methods [list]). They cover not only many well-known coupling metrics but also additional features (i.e., dynamic aspects). For instance, Message Passing Coupling (MPC) [57] counts static method calls for all invoked methods in the import direction, and Request For a Class (RFC) [18] counts static method calls for distinct methods in the import direction, while Coupling Between Objects (CBO) [18] counts static method calls for distinct methods in both directions. The coarse-grained metrics, such as Coupling Factor (CF) [13], are not considered, because they are measured based on the number of coupled classes, not on the methods. All the mentioned coupling metrics capture only static aspects, which are based on static method calls that can be obtained by analyzing source codes without running a program.

Table 7.4: Fitness function values of the original design.

	jEdit	Columba	jGit
Fitness fn.	0.023287	0.023117	0.021357

Table 7.4 shows the fitness function values of the original design (i.e., before applying selected refactorings). For measuring the costs of computation and search space exploration of the approach, we use the number of iterations for the selection process and the elapsed time. The elapsed time is measured under following conditions: processor 1.8GhHz Intel Core i5, Memory 8G 1600 MHz DDR3, Graphic Intel HD Graphics 4000 512MB, and Software OS X 10.8.2. We compare the two approaches on fitness function values (maintainability improvement), the number of iterations, and the elapsed time (computation cost and search space exploration cost).

RQ4: Effect of the RED

To investigate the effect of the RED (RQ4), we compare the approach considering the RED (our approach) with the approach without considering the RED. We compare these two approaches on fitness function values (maintainability improvement) as well. To analyze the different results of the two approaches, the accumulated deviation is measured as follows.

$$\sum_{i=0}^{\sharp iterations} |Expected_i - Actual_i|,$$

where $Expected_i$ and $Actual_i$ are expected and actual maintainability (i.e., external links assessed in the refactoring effect evaluation framework) on i-th iteration, respectively.

7.4 Results

7.4.1 Dynamic Information-based Identification of Refactoring Candidates

RQ1: Effect of Dynamic Information for Cost-Effective Refactoring Identification

The results are represented in Fig. 7.3, Fig. 7.4, and Fig. 7.5 for jEdit, Columba, and JGIT, respectively; the x-axis shows each applied refactoring, and the y-axis shows the number of propagated changes of methods or classes to accommodating original changes. In addition, Table 7.5 summarizes the percentage of reduction for propagated changes (i.e., methods) and the rate of reduction for propagated changes (i.e., methods) for each applied refactoring.

For jEdit, as in Fig. 7.3(a) and Fig. 7.3(b), the same number of propagated changes is reduced for all approaches in the first applied refactoring. However, from the next applied refactorings, the approaches using dynamic information (group 1 and group 3) reduce the number of propagated changes faster than the approach using static information only (group 2) does. For this reason, to reach the same number of reduced propagated changes—for example, where the percentage of reduction for propagated changes is around 72% $\sim$ 75%—the required numbers of refactoring application are 5, 6, and 7 for group 3, group 1, and group 2, respectively. As a result, the average rate of reduction for propagated changes of all nine of applied refactorings for the approaches using dynamic information (group 1 and group 3) are higher than that of the approach using static information only (group 2). For instance, the average rates of reduction for propagated changes of all nine applied refactorings are 11.11%, 10.56%, and 9.44% for group 3, group 1, and group 2, respectively. Furthermore, at the final solution, where propagated changes do not drop anymore, the number of reduced propagated changes of the approaches using dynamic information (group 1 and group 3) is greater than that of the approach using static information only (group 2). For instance, when positing the total percentage of reduction of propagated changes for the combination of the two approaches (group 3) as 100%, then those for group 1 and group 2 are 95% and 85%, respectively.

For Columba, as in Fig. 7.4(a) and Fig. 7.4(b), the approaches using dynamic information (group 1 and group 3) also reduce the number of propagated changes much faster and bigger than the approach using static information only (group 2) does. For this reason, as in jEdit, to reach the same number of reduced propagated changes—for example, where the percentage of reduction for propagated changes is around 75% $\sim$ 76%—the required numbers of refactoring application are 4, 6, and 10 for group 3, group 1, and group 2, respectively. As a result, for instance, the average rate of reduction for propagated changes of all 11 applied refactorings are 9.09%, 7.67%, and 7.10% for group 3, group 1, and group 2, respectively. In addition, when positing the total percentage of reduction for propagated changes for the combination of the two approaches (group 3) as 100%, then those for group 1 and group 2 are 85% and 78%, respectively. It is also worth mentioning that in Columba, the absolute scale of reduction for propagated changes is relatively small, because there are not many revisions to be retrieved. Referring to the report period (in Table 7.1), we assume that the maturity level of the development for Columba is relatively lower than jEdit, and Columba may still be in the development process. In fact, jEdit has been developed and maintained for over ten years. Furthermore, the developers are much smaller, while the size of the program is much bigger than jEdit's; they may not have exerted as much effort for the revisions as jEdit does.

For JGIT, as in Fig. 7.5(a) and Fig. 7.5(b), group 1 reduces the number of propagated changes faster than group 2 does, though only from the fourth to the eighth applied refactorings. Even at the final solution, the number of reduced propagated changes of group 1 is smaller than that of group 2. As a result, for instance, the average rate of reduction for propagated changes of the total of 12 applied refactorings are 8.33%, 5.13%, and 6.85% for group 3, the group 1, and group 2, respectively. In addition, when positing the total percentage of reduction for propagated changes for combination of the two approaches (group 3) as 100%, then those for group 1 and group 2 are 62% and 82%, respectively.

As opposed to the results with jEdit and Columba, with JGIT, group 1 does not outperform group 2. By analyzing the revision history and source codes of JGIT, we found the following observations that can explain this result. JGIT is a distributed source version control system and provides many special features for working in a distributed environment with high speed. Since the most common use of scenarios for using version control systems are committing, pushing, cloning, or pulling a file into a repository, we have mostly captured these normal scenarios when performing dynamic profiling. However, real changes—which had occurred in the examined

Table 7.5: Indicators of cost-effective refactorings: (1) Percentage of reduction for propagated changes, (2) Rate of reduction for propagated changes for jEdit, Columba, and JGIT.

jEdit

Percentage of reduction for propagated changes (%)				Rate of reduction for propagated changes (%)			
♯ of applied refactoring	Dynamic + Static	Static	Dynamic	♯ of applied refactoring	Dynamic + Static	Static	Dynamic
1	30	30	30	1	30	30	30
2	42.5	37.5	40	2	12.5	7.5	10
3	50	42.5	52.5	3	7.5	5	12.5
4	60	47.5	60	4	10	5	7.5
5	72.5	60	65	5	12.5	12.5	5
6	82.5	60	77.5	6	10	0	12.5
7	90	75	85	7	7.5	15	7.5
8	95	85	90	8	5	10	5
9	100	85	95	9	5	0	5
Average	69.17	58.06	66.11	Average	11.11	9.44	10.56

Columba

Percentage of reduction for propagated changes (%)				Rate of reduction for propagated changes (%)			
♯ of applied refactoring	Dynamic + Static	Static	Dynamic	♯ of applied refactoring	Dynamic + Static	Static	Dynamic
1	31.3	6.3	31.3	1	31.3	6.3	31.3
2	46.9	23.4	43.8	2	15.6	17.2	12.5
3	59.4	35.9	50.0	3	12.5	12.5	6.3
4	75.0	45.3	65.6	4	15.6	9.4	15.6
5	84.4	57.8	73.4	5	9.4	12.5	7.8
6	87.5	65.6	76.6	6	3.1	7.8	3.1
7	89.1	70.3	78.1	7	1.6	4.7	1.6
8	92.2	73.4	79.7	8	3.1	3.1	1.6
9	96.9	75.0	81.3	9	4.7	1.6	1.6
10	98.4	76.6	82.8	10	1.6	1.6	1.6
11	100.0	78.1	84.4	11	1.6	1.6	1.6
Average	78.27	55.26	67.90	Average	9.09	7.10	7.67

JGIT

Percentage of reduction for propagated changes (%)				Rate of reduction for propagated changes (%)			
♯ of applied refactoring	Dynamic + Static	Static	Dynamic	♯ of applied refactoring	Dynamic + Static	Static	Dynamic
1	9.46	9.46	2.23	1	9.46	9.46	2.23
2	14.33	14.33	3.01	2	4.87	4.87	0.78
3	18.88	18.74	12.48	3	4.55	4.41	9.47
4	28.71	21.57	21.43	4	9.83	2.83	8.95
5	48.70	23.79	43.74	5	19.99	2.22	22.31
6	51.53	27.78	45.55	6	2.83	3.99	1.81
7	57.98	30.98	51.30	7	6.45	3.20	5.75
8	60.76	40.21	51.95	8	2.78	9.23	0.65
9	81.26	63.96	52.50	9	20.50	23.75	0.55
10	92.72	75.05	56.63	10	11.46	11.09	4.13
11	93.60	77.50	61.54	11	0.88	2.45	4.91
12	100.00	82.24	61.54	12	6.40	4.74	0.00
Average	54.83	40.47	38.66	Average	8.33	6.85	5.13

revisions of the development history for JGIT—are related to developing and correcting errors of the algorithms that are *not* frequently used but contain critical functions, and the complexity of these algorithms is high. An example of such algorithms is `packing`; JGIT stores each newly created object as a separate file, and this takes a great deal of space and is inefficient. Thus, periodic packing of the repository is required to maintain space efficiency, which requires very complex computation. For the reasons stated above, in JGIT, group 1 may rarely identify refactorings on those parts of the algorithms; thus, the percentage of reduction for propagated changes and the rate of reduction for propagated changes are rather small. However, the combination of the two approaches (group 3) still outperforms the approach using static information alone (group 2). It is obvious that some of the refactoring candidates—not identified in the approach using static information (group 2) and *only* identified in the approach using dynamic information (group 1)—contribute to improving maintainability even faster. Here, these two approaches are mutually complementary; thus, it can be said that using the dynamic information in addition to the static information helps to improve maintainability even faster.

From the results presented above, we can conclude that, in three subjects—jEdit, Columba, and JGIT—, dynamic information is helpful in identifying cost-effective refactorings that fast improve maintainability; and, considering dynamic information in addition to static information provides even more opportunities to identify cost-effective refactorings because of the refactoring candidates that are uniquely identified by the approach using dynamic information only.

RQ2: Effect of Dynamic Information for Extracting Refactoring Candidates in Frequently Changed Classes

For each subject, jEdit, Columba, and JGIT, the commonly found classes (i.e., intersect set) of each approach using static information and approach using dynamic information are represented in Table 7.6. The intersect set is represented as (1) the number of the classes (Class ♯), and (2) the number of occurred changes in those classes (Change ♯). The asterisk (*) is appended to the results of better solutions (i.e., those in which a greater number of the classes or a greater number of occurred changes in those classes are commonly found). For two subjects, jEdit and Columba, in the approach using dynamic information (group 1), more classes—extracted as refactoring candidates—are found in the classes where real changes had occurred. For JGIT, in the approach using dynamic information (group 1), more classes—extracted as refactoring candidates—are found only in the classes of top 10% and 20% most frequently changed.

On the other hand, the two distance measures (K: Kendall's tau, F: Spearman's footrule [27]) of the approach using static information and the approach using dynamic information are represented in Table 7.7. The distance measures count the number of pairwise disagreements between two top k-ranked lists. Therefore, the larger the distance, the more dissimilar the two top k ranked lists are; conversely, the smaller the distance, the more similar the two top k-ranked lists are. The asterisk (*) is also appended to the results of better solutions (i.e., those with the smaller distance measures). Likewise the results in the commonly found classes, for two subjects, jEdit and Columba, in the approach using dynamic information (group 1), the ranked lists of classes—extracted as refactoring candidates—are more similar to the ranked list of the real changed classes. For JGIT, in the approach

Table 7.6: Commonly found classes between the real changed classes and the extracted classes as refactoring candidates for each approach using static information and approach using dynamic information.

jEdit

Changed			Static ∩ Changed		Dynamic ∩ Changed	
Top %	Class ♯	Change ♯	Class ♯	Change ♯	Class ♯	Change ♯
10.00%	7	67	6	60	7*	67*
20.00%	16	110	7	64	10*	79*
30.00%	27	143	9	70	12*	85*
100.00%	72	207	19	82	21*	99*

Columba

Changed			Static ∩ Changed		Dynamic ∩ Changed	
Top %	Class ♯	Change ♯	Class ♯	Change ♯	Class ♯	Change ♯
10.00%	27	458	9	220	14*	269*
20.00%	55	624	13	241	15*	275*
30.00%	79	788	16	251	17*	282*
100.00%	265	993	22	260	24*	292*

JGIT

Changed			Static ∩ Changed		Dynamic ∩ Changed	
Top %	Class ♯	Change ♯	Class ♯	Change ♯	Class ♯	Change ♯
10.00%	20	5039	9	2872	10*	2899*
20.00%	50	7296	19	3709	22*	3758*
30.00%	91	8754	27*	3992*	26	3938
100.00%	258	9773	44*	4109*	33	3998

† The asterisk (*) is appended to the results of better solutions (i.e., those in which a greater number of the classes or a greater number of occurred changes in those classes are commonly found).

Table 7.7: Top k ranking distance measures (K: Kendall's tau; F: Spearman's footrule [27]) between the real changed classes and the extracted classes as refactoring candidates for each approach using static information and approach using dynamic information.

	jEdit				**Columba**				**JGIT**			
Changed	Static		Dynamic		Static		Dynamic		Static		Dynamic	
Top %	K	F	K	F	K	F	K	F	K	F	K	F
10.00%	68.5	69	53*	52*	397	207	258.5*	177*	1779.5	695	1284.5*	449.25*
20.00%	158	69.25	129.5*	65.25*	1000.5	712	791*	659*	3494	1198.25	2702*	1026*
30.00%	256.5	105	194*	96*	1785	1653	1617.5*	1520*	4411.5*	2099*	5489.5	2199.25
100.00%	1229	1329.25	1003.5*	1196.25*	15857	19824	15655.5*	19328*	26499*	22752*	29623	22974.25

† The asterisk (*) is appended to the results of better solutions (i.e., those with the smaller distance measures).

using dynamic information (group 1), the ranked list of classes—extracted as refactoring candidates—are more similar only to the ranked lists of the top 10% and 20% most frequently changed.

The results presented above in three subjects—jEdit, Columba, and JGIT—show that dynamic information is helpful in extracting refactoring candidates in the classes where real changes had occurred. In addition, overall, the approach using dynamic information even outperforms the approach using static information for finding *frequently* changed classes. Even though the former approach is not always better than the latter approach, we find that the correlation does exist between the frequently changed classes and the classes of refactoring candidates extracted from the approach using dynamic information. The results offer promising support for using dynamic information for extracting refactoring candidates from highly-ranked frequently changed classes, and, further, that using dynamic information in addition to static information can be a great help for cost-effective refactoring identification.

7.4.2 RED-aware Grouping of Multiple Elementary Refactorings

RQ3: Effect of Multiple Refactorings

Table 7.8 summarizes the results of which each approach has reached to the final solution (i.e., no more refactorings that improve maintainability are found): fitness function values (maintainability improvement), the number of iterations, and the elapsed time (computation cost and search space exploration cost) for jEdit, Columba, and JGIT, respectively. The graphs in Fig. 7.6 are presented to show the visual results. In all three projects, the fitness

Table 7.8: Results of the effect of multiple refactorings.

Subject	Comparators	Fitness fn.	Computation cost and search space exploration cost	
			Iteration ($\sharp$)	Elapsed Time (sec)
jEdit	Rulebased_RCs only	154	0.030322	431.89
	Our approach	23	0.032312	241.12
Columba	Rulebased_RCs only	220	0.036951	581.53
	Our approach	41	0.038132	205.74
jGit	Rulebased_RCs only	43	0.022549	198.77
	Our approach	72	0.026701	232.68

† Rulebased_RCs: Refactoring Candidates from Rule-based Identification.

† Rulebased_RCs only: approach without MISs.

† Our approach: approach with Rulebased_RCs + MISs.

function values of our approach (the approach of selecting multiple refactorings) are greater than the approach of refactoring identification using the Rule-based_RCs only (without MISs): jEdit ($0.032312 > 0.030322$), Columba ($0.038132 > 0.036951$), and jGit ($0.026701 > 0.022549$). In addition, the total number of iterations and taken time to reach to the solution of our approach is much lesser than the approach of refactoring identification using the Rule-based_RCs only (without MISs): $\sharp$ of iterations—jEdit ($23 < 154$) and Columba ($41 < 220$); time

Table 7.9: Results of the effect of the RED.

Subject	Comparators	Fitness fn.	Accumulated Deviation
jEdit	Not_RED	0.032379	9246
	Our approach	0.033472	0
Columba	Not_RED	0.030720	40758
	Our approach	0.037123	0
jGit	Not_RED	0.023602	13058
	Our approach	0.028192	0

† Not_RED: approach without considering the RED.

† Our approach: approach considering the RED.

(sec)—jEdit ($241.12 < 431.89$) and Columba ($205.74 < 581.53$).

We take a close look the graphs in terms of the number of iterations (Fig. 7.7). The graphs of jEdit (Fig. 7.7(a)), Columba (Fig. 7.7(b)), and JGIT (Fig. 7.7(c)), present the results of the effect of multiple refactorings on the number of iterations, respectively; the x-axis shows the number of iterations, and the y-axis shows the fitness function values (cohesion / coupling). In both jEdit and Columba, fitness function values grows fast in our approach; in contrast, the approach of refactoring identification using the Rule-based_RCs only (without MISs) grows gradually. It takes certain overhead of computing MISs in the first procedure of selection for multiple refactorings (denoted as the processing time in the graph). However, the benefit of reduced time overcomes this overhead.

In jGit, the approach of refactoring identification using the Rule-based_RCs only (without MISs) faces the local optimum problem. By examining the logged results, we found the following observation. First, during the iterative process of refactoring selection, the approach of refactoring identification using the Rule-based_RCs only (without MISs) selects the refactorings in the same place. Thus, the second best refactoring rarely selected. In contrast, our approach selects refactorings globally, which helps to prevent this problem. Second, in the approach of refactoring identification using the Rule-based_RCs only (without MISs), it kills the other Move Method refactoring opportunities. Sometimes, a group of smaller pieces of refactorings are useful, as in our approach.

From the results, we can conclude that our approach selects refactorings that lead the software design to reach higher fitness function values (better improve maintainability) with smaller costs (i.e., smaller search space exploration cost and shorter time). Even though it takes certain overhead of computing MISs at the very beginning of our approach, the benefit of reduced cost overcomes this overhead. Furthermore, in some project, the approach of refactoring identification using the Rule-based_RCs only (without MISs) may face the local optimum problem. Our approach tends to have better performance in avoiding local optimum by selecting refactorings globally.

RQ4: Effect of the RED

Table 7.9 summarizes the results of which each approach has reached to the final solution (i.e., no more refactorings that improve maintainability are found): fitness function values (maintainability improvement) for jEdit, Columba, and JGIT, respectively. In all three projects, the fitness function values of our approach (the approach considering the RED) are greater than the approach without considering the RED: jEdit (0.033472 > 0.032379), Columba (0.037123 > 0.030720), and jGit (0.028192 > 0.023602). To explain the different results of the two approaches, the deviation between actual and expected maintainability (i.e., external links assessed in the refactoring effect evaluation framework) is measured for each iteration of the selection process and accumulated. The accumulated deviations of the approach without considering the RED are 9246, 40758, and 13058, for jEdit, Columba, and jGit, respectively. It can be interpreted that to this amount, the approach without considering the RED miscalculates the maintainability of the suggested groups of refactorings; and the group of refactorings that does not mostly improve maintainability could be selected.

From the results, we can conclude that when selecting multiple refactorings, considering the RED on is important to correctly identify the group of refactorings that best improves maintainability. Even though the refactorings are not syntactically dependent, the RED should be considered when selecting multiple refactorings.

7.5 Threats to Validity

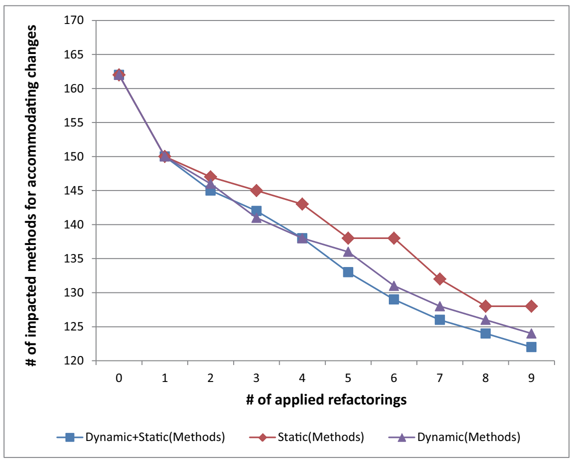
We assume that the cost of each refactoring is the same; therefore, the number of applied refactorings is regarded as the refactoring cost (effort). However, the number of applied refactorings does not actually reflect the effort required to apply them. For practical use of our approach, several factors need to be considered. More detailed discussion is provided in the next subsection.

The capability of identified refactorings for maintainability improvement is assessed by using the change simulation method. In the experiment, we obtained changes from the change history for the input of the change impact analysis. For changes obtained from the change history, it would be good to extract intentional changes by excluding ripple effects—that the intentional changes necessitated—among the obtained changes, perform change impact analysis for those intentional changes, and compare the results of change impact analysis. However, discernment of intentional changes among the obtained changes is not feasible, because it is nontrivial to identify whether a change is an intentional change or a ripple effect; therefore, we did not use the intentional changes as the input of the change impact analysis. Thus, we use the obtained changes (i.e., input as original changes), then perform change impact analysis to identify the potential consequences (i.e., output as propagated changes) for those obtained changes.

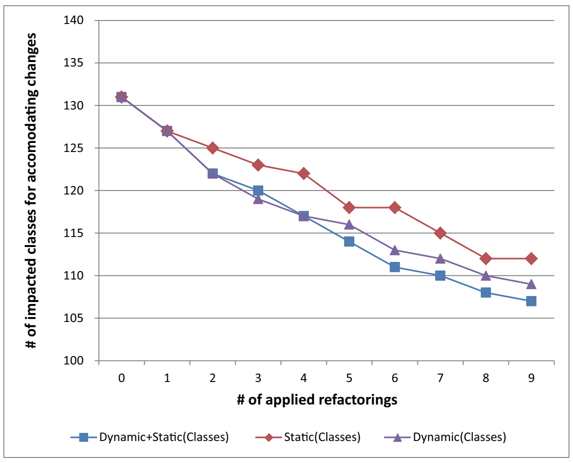
For implementing change impact analysis, the two-steps of direct and indirect propagated methods are considered by using different weight values. The further step of indirect propagated methods can be considered.

To evaluate the capability of the selected refactorings, we use the maintainability evaluation function, which is based on the coupling and cohesion metrics. This maintainability evaluation function fits to our evaluation criteria, because we regard improving maintainability of software as having high cohesion and low coupling. The maintainability evaluation function designed for other goals or weighting on the specific criteria may produce different fitness function values.

When constructing RED-aware graph, more types of dependencies such as structural dependencies can be considered for grouping entities into sets more precisely.

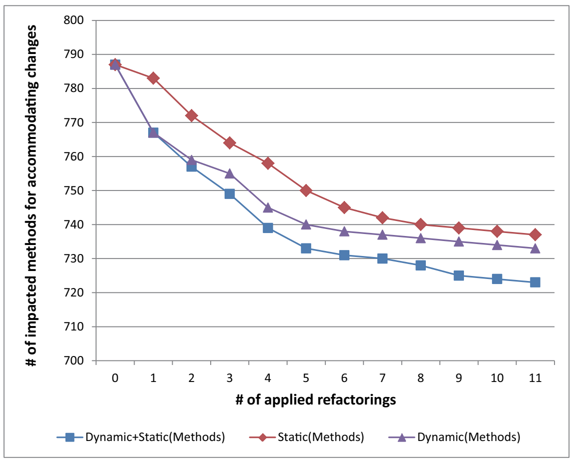


(a) Number of propagated methods for accommodating changes on jEdit

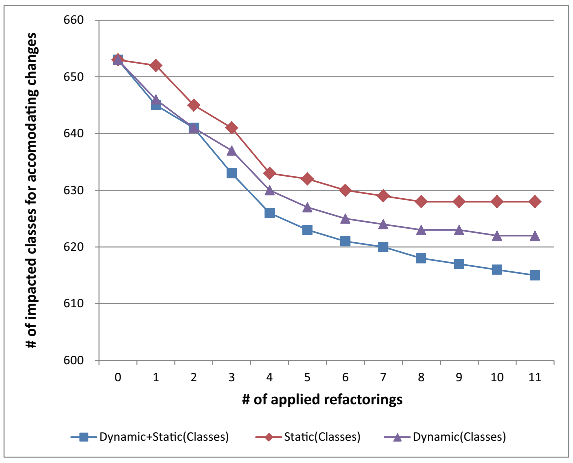


(b) Number of propagated classes for accommodating changes on jEdit

Figure 7.3: Change simulation for jEdit.

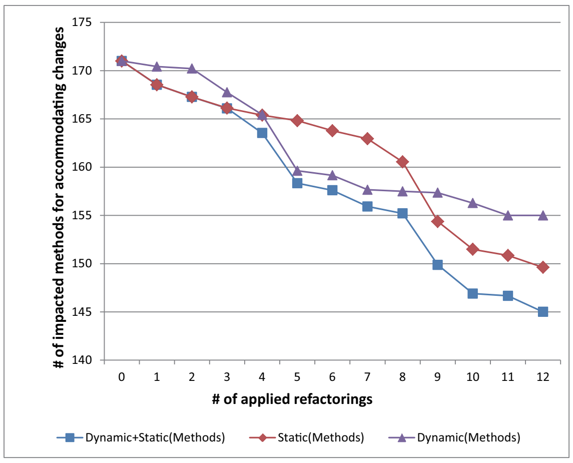


(a) Number of propagated methods for accommodating changes on Columba

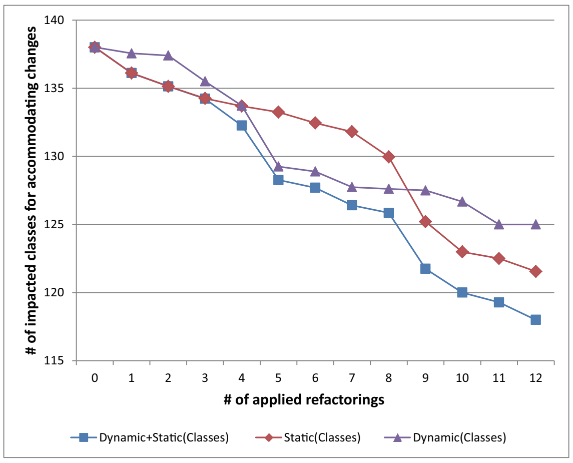


(b) Number of propagated classes for accommodating changes on Columba

Figure 7.4: Change simulation for Columba.

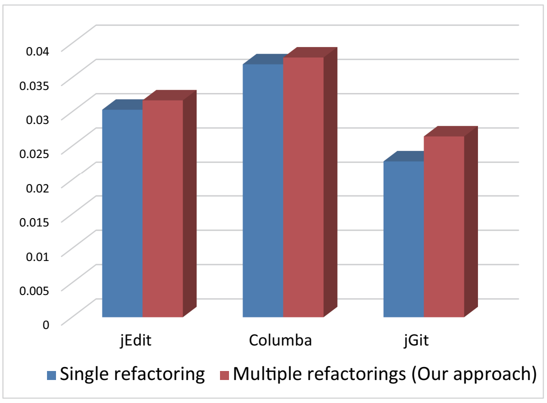


(a) Number of propagated methods for accommodating changes on JGIT

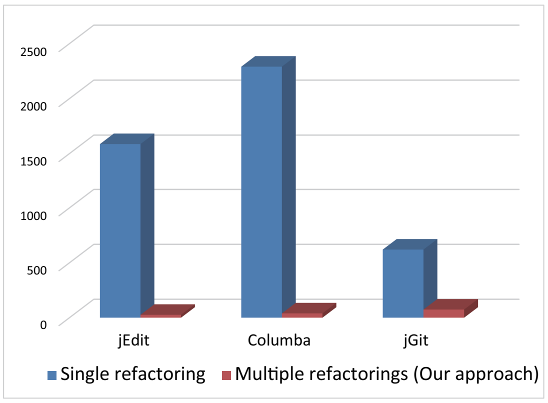


(b) Number of propagated classes for accommodating changes on JGIT

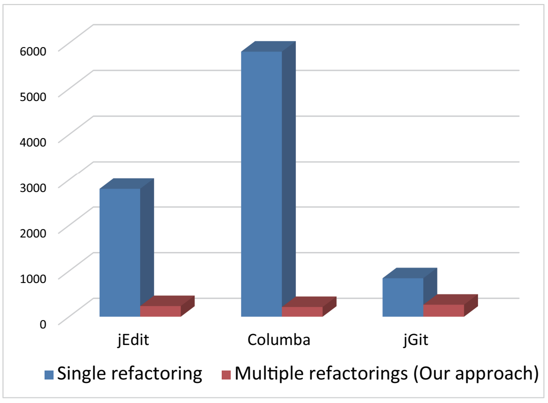
Figure 7.5: Change simulation for JGIT.



(a) Fitness fn. [38]

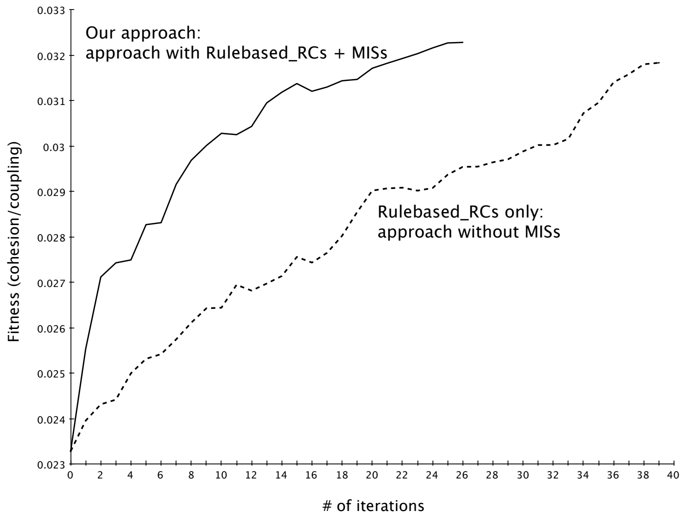


(b) ♯ of iterations

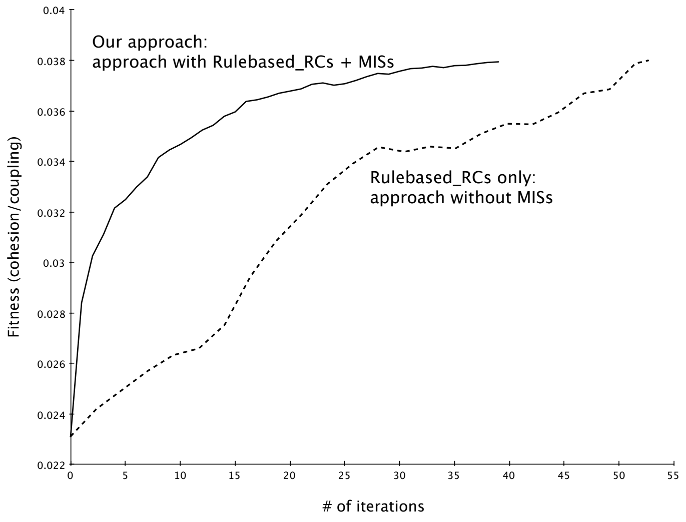


(c) Elapsed time (sec)

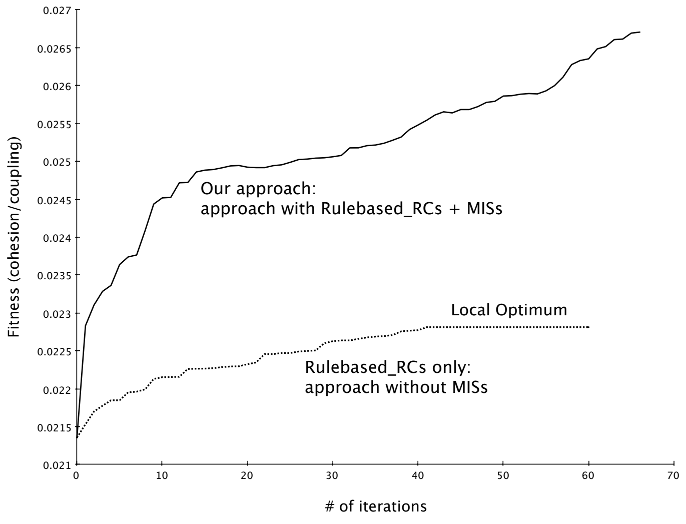
Figure 7.6: The effect of multiple refactorings.



(a) jEdit



(b) Columba



(c) JGIT

Figure 7.7: The effect of multiple refactorings on the number of iterations.

Chapter 8. Discussion

Some of the researches addressed the method of estimating refactoring cost; for example, Zibran and Roy [93] propose a refactoring effort model that takes into account several types of effort needed to remove software code clones. To more accurately estimate refactoring cost, we need to consider the effort needed to perform the activities—refactoring identification, refactoring application, and refactoring maintenance—of the entire refactoring process (explained in Section 3). For refactoring identification, refactoring complexity (e.g., big or small for code modification, or easy or difficult for understanding context) needs to be considered. It is reasonable to expect that big refactorings—which consist of a series of small refactorings—would require more effort to be applied than small refactorings would do, because they should affect larger portion of source codes; and at the same time, impact of big refactorings on maintainability improvement tends to be larger. For instance, in the experiment—performed without considering refactoring complexity—, class-level refactorings (i.e., Collapse Class Hierarchy refactorings) are selected in many cases than method-level refactorings (i.e., Move Method refactorings); because the impact on maintainability improvement of class-level refactorings tends to be larger than that of method-level refactorings. If the refactoring complexity of the application is taken into account for estimating refactoring cost, method-level refactorings may be more selected. Refactoring complexity of the application can be considered by dividing each refactoring into fine-grained (e.g., atomic-level) transformations and giving each a different weight. For refactoring application, 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 costs such as the effort of relocating codes, especially when the refactorings are complex. Refactoring inspection costs also need to be considered, because it is a human who decides whether to refactor or not. For instance, the developer or the maintainer needs to take time to decide whether identified refactorings should be applied or not. Even though the refactorings are beneficial to maintainability improvement, they could be rejected to be applied due to the confliction with other design practices and principles. Finally, for refactoring maintenance, the effort involved in testing the refactored code and checking consistency with other software artifacts needs to be considered.

In the experiment, the main key to obtain a better outcome is how strongly the frequently utilized parts are correlated with the parts that actually have been changed and how much more refactorings are identified and applied in those parts. For instance, in jEdit and Columba, changes have occurred in the parts that are often utilized; while in JGIT, change-occurred parts are not strongly correlated with the frequently used parts. By examining the changes made to JGIT, we notice that development of system's main functionalities has almost been finished; and developers seem to focus on perfective maintenance. It is reasonable that, in this case, changes can be made to the places dealing with exceptional scenarios or containing functionalities utilized only by high-end users. Even though the use of frequency is rather low, the importance or complexity of developing such parts can be high. For this reason, for JGIT, other predictors, such as structural complexity (e.g., class size), may need to be additionally considered to identify better cost-effective refactorings. Nevertheless, it is worth pointing out that the dynamic information is the important factor for identifying cost-effective refactorings, because the experimental results show that, the combination of two approaches—the approach using dynamic information and the approach of static approach—still outperforms the approach using static information alone. We discussed with senior developers—who work in IT industries over ten years—for interpreting these experimental results. They support the arguments by providing the following explanations: the system having intensive user interactions tends to be gradually developed by actively accommodating users' requests; thus, changes are more likely to be occurred where users more utilize. On the other hand, the system, which is algorithmic-based and has rather less interactions with users, tends to be developed in a way of completing each decomposed function; thus, changes are not likely to be occurred where the development is completed.

We defined a total of 18 refactoring extraction rules. Given the inherent limitation of the rule-based approach, the rules cannot be complete. Further, more rules need to be developed and refined to find better refactoring candidates. In addition, other methods of finding refactoring candidates are needed. Using our rule-based approach, for refactorings such as Extract Class and Extract Method, determining specific code blocks to be split in an automated way is difficult.

While the dynamic profiling-based approach of refactoring identification needs efforts of dynamic profiling in addition to the approach of using static information only, the benefit of using the dynamic profiling-based approach may outweigh the efforts of dynamic profiling. In addition, the efforts of dynamic profiling are manageable because dynamic profiling is done just once at the beginning of the approach.

We assume that the application of elementary refactorings (i.e., Move Method refactorings) do not change—delete or merge—entities in the MISs; but only the membership information—which entity is placed in which class—is changed. As a result, the vertices and edges of the constructed RED-graph G_R is remained same after the application of the selected refactorings. Therefore, MISs do not need to be calculated for every iteration of the refactoring selection process. The calculation of MISs is done once at the beginning of the refactoring selection process. For the future work, we plan to consider more types of refactorings. According to this, we need to develop the method of recalculating the MISs for accommodating the situations that the application of elementary refactorings deletes or merges entities.

It is worth to mention that the goal of our method of refactoring selection is not to find an optimal sequence of refactorings. We attempt to select multiple refactorings that can be applied at the same time; and the sequence of refactorings is generated by logging the results of the selected multiple refactoring for each refactoring selection process. By this way of selection, we take the advantages of (1) considering refactoring dependencies and the creation of new refactoring candidates after the application of the refactoring suggestions, and (2) more efficient computing and searching than the approach of selecting just the single best refactoring.

Chapter 9. Conclusion and Future Work

9.1 Summary of Contributions

In the thesis, we provide the methods for supporting systematic refactoring identification: identification of refactoring candidates and selection of refactorings to be applied. For identification of refactoring candidates, we attempt top-down and bottom-up approaches. First, for the top-down approach—finding refactoring opportunities by using heuristic rules for eliminating violations of design principles (e.g., removing bad smells) in object-oriented software systems—we establish the rules to extract the refactoring candidates with the aim of reducing dependencies of entities of methods and classes. When establishing the rules, entities are identified based on how the users utilize the software (e.g., user scenario and operational profile); and within these entities, refactoring candidates are identified. Second, for the bottom-up approach—identification of refactoring opportunities without humans' insights—we develop the method for grouping elementary refactorings by using the concept of the MIS in graph theory. For grouping of elementary refactorings, we develop the method for forming entities (i.e., methods and attributes) into MISs by taking the RED into account. The methods involved in each MIS are transformed into a group of Move Method refactorings. Each of the group has elementary refactorings that can be applied at the same time. For selecting refactorings to be applied, we provide the method of selecting refactorings (refactoring effect evaluation framework) by supporting assessment and impact analysis of *elementary* refactorings. By referring the refactoring effect evaluation framework (delta table), we select the group of refactorings containing the multiple elementary refactorings that best improves maintainability. We evaluate our proposed approach in three open-source projects—jEdit, Columba, and JGIT. From the experimental results, we conclude that dynamic information is helpful in identifying refactorings that efficiently improve maintainability; because dynamic information is helpful for extracting refactoring candidates in frequently changed classes. Furthermore, the experimental results show that the selection method of multiple refactorings reduces search space exploration cost; and the RED should be considered when selecting multiple refactorings.

The contributions of the thesis can be summarized as follows.

- Establish the framework of systematic refactoring identification
- Develop the method for dynamic information-based identification of refactoring candidates
- Recognize the new dependency of refactorings (i.e., RED) that is essential to be considered to correctly identify a group of refactorings that most improve maintainability
- Develop the method for RED-aware grouping of elementary refactorings (by using the concept of the MIS in graph theory)

- Provide the method for selecting multiple (elementary) refactorings by supporting assessment and impact analysis of elementary refactorings (based on the matrix computation, which enables fast computation)
- Perform empirical studies on large-sized open source programs

9.2 Future Work

The framework of refactoring selection provides the methods of assessment and impact analysis of elementary refactorings (based on the matrix computation), and these methods support to easily extend considering refactorings to other various type of refactorings; because the action of big refactoring (e.g., Collapse Hierarchy Class refactoring) comprises of elementary refactorings (e.g., Move Method refactoring). Therefore, to provide more complete methods for supporting systematic refactoring identification, we plan to consider more types of refactorings (that are considered in [22]), for example, Pull Up Method refactoring and Form Template Method refactoring.

References

[1] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

[2] G. Arévalo. Understanding behavioral dependencies in class hierarchies using concept analysis. *Proceedings of LMO*, 3:47–59, 2003.

[3] E Arisholm, LC Briand, and A Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, pages 491–506, 2004.

[4] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transaction on Software Engineering*, 30:491–506, 2004.

[5] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17, New York, NY, USA, 2006. ACM.

[6] Felix Bachmann, Len Bass, and Robert Nord. Modifiability tactics. *Technical Report*, 2007.

[7] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs, journal of object-oriented program. *Journal of Object-Oriented Program*, 11(8):47–52, 1999.

[8] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, 2002.

[9] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Emf model refactoring based on graph transformation concepts. *Electronic Communications of the EASST*, 3(0), 2007.

[10] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring - improving coupling and cohesion of existing code. *Working Conference on Reverse Engineering*, 0:144–151, 2004.

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

[12] Borland. *Together 2006 Release 2 for Eclipse*, 2006. http://www.borland.com/us/products/together/index.html.

[13] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91–121, 1999.

[14] LC Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 57–66, 2003.

[15] Lionel C. Briand and Jurgen Wust. Empirical studies of quality models in object-oriented systems. In *Advances in Computers*, pages 97–166. Academic Press, 2002.

[16] M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM, 1999.

[17] Costas Busch. *Maximal Independent Set*, 2000. http://www.slidefinder.net/m/maximal_independent_set/mis/15624838.

[18] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[19] KJ Cios, W. Pedrycz, and RM Swiniarsk. Data mining methods for knowledge discovery. *IEEE Transactions on Neural Networks*, 9(6):1533–1534, 1998.

[20] Columba. *Columba*, 2011. http://sourceforge.net/projects/columba/.

[21] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufmann, 2002.

[22] Java development user guide. *Refactoring Actions*, 2013. http://help.eclipse.org/juno/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm.

[23] M. Dmitriev. Selective profiling of java applications using dynamic bytecode instrumentation. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 141–150. IEEE, 2004.

[24] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring - improving coupling and cohesion of existing code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 144–151, Washington, DC, USA, 2004. IEEE Computer Society.

[25] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, pages 1–7. Citeseer, 2000.

[26] A.L. Edwards. *An introduction to linear regression and correlation*. W.H. Freeman, San Francisco, CA, 1976.

[27] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–36. Society for Industrial and Applied Mathematics, 2003.

[28] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 2012.

[29] M. Fowler and K. Beck. Refactoring: Improving the design of existing code. *Addison-Wesley Professional*, 1999.

[30] R. France, J. Bieman, and B.H.C. Cheng. Repository for Model Driven Development (ReMoDD). *Lecture Notes in Computer Science*, 4364:311, 2007.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[32] V. Garousi, L. Briand, and Y. Labiche. Analysis and visualization of behavioral dependencies among distributed objects based on uml models. *Model Driven Engineering Languages and Systems*, pages 365–379, 2006.

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

[34] David Gilbert and Thomas Morgner. *JFreeChart*, 2010. http://www.jfree.org/jfreechart/.

[35] M.S. Gittens. *The Extended Operational Profile Model for Usage-based Software Testing*. Library and Archives Canada Bibliothèque et Archives Canada, 2005.

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

[37] Ah-Rim Han. *ARTool*, 2011. https://github.com/igsong/ARTOOL.

[38] Ah-Rim Han and Doo-Hwan Bae. Dynamic profiling-based approach to identifying cost-effective refactorings. *Information and Software Technology*, 55(6):966–985, 2013.

[39] Ah-Rim Han, Sang-Uk Jeon, Doo-Hwan Bae, and Jang-Eui Hong. Measuring behavioral dependency for improving change-proneness prediction in uml-based design models. *The Journal of Systems & Software*, 83(2):222 – 234, Feb 2010.

[40] A.R. Han, S.U. Jeon, D.H. Bae, and J.E. Hong. Behavioral Dependency Measurement for Change-Proneness Prediction in UML 2.0 Design Models. *Proceedings of the 32nd Annual IEEE International Computer Software and Applications*, pages 76–83, 2008.

[41] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Series in Data Management Systems, San Francisco, CA, second edition, 2006.

[42] M. Harman. Refactoring as testability transformation. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 414–421. IEEE, 2011.

[43] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113. ACM, 2007.

[44] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[45] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[46] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, pages 53–62, 2012.

[47] W.W. Hwu and P.P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *ACM SIGARCH Computer Architecture News*, pages 242–251. ACM, 1989.

[48] IBM Rational. *Rational Software Architect Standard Edition*, 2008. http://www-01.ibm.com/software/awdtools/swarchitect/standard/.

[49] jEdit. *jEdit*, 2011. http://www.jedit.org/.

[50] S.U. Jeon, J.S. Lee, and D.H. Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pages 337–345. IEEE, 2002.

[51] JetBrains. *IntelliJ IDEA*, 2012. http://www.jetbrains.com/idea/.

[52] JGIT. *JGIT*, 2011. http://eclipse.org/jgit/.

[53] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[54] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings. International Conference on Software Maintenance, 2002.*, pages 576 – 585, 2002.

[55] Joshua Kerievsky. *Refactoring to patterns*. Addison Wesley Pearson Education, 2005.

[56] Sukhee Lee, Gigon Bae, Heung Seok Chae, Doo-Hwan Bae, and Yong Rae Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Softw., Pract. Exper.*, 41(5):521–550, 2011.

[57] W Li and S Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[58] H. Liu, Z. Ma, W. Shao, and Z. Niu. Schedule of bad smell detection and resolution: A new way to save effort. *Software Engineering, IEEE Transactions on*, 38(1):220–235, 2012.

[59] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[60] P.C. Mahalanobis. On the generalized distance in statistics. *Proc. Nat. Inst. Sci. India*, 2(1):49–55, 1936.

[61] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 45–52. IEEE, 1998.

[62] J. Martin and C. L. McClure. *Software Maintenance: The Problems and its Solutions*. Prentice Hall, 1983.

[63] R. C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall, 2003.

[64] T Mens, G Taentzer, and O Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, Jan 2007.

[65] T Mens and T Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[66] J.D. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14–32, 1993.

[67] J. T. Nosek and P. Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance: Research and Practice*, 2(3):157–174, 1990.

[68] M O'Keeffe and M Ó Cinnéide. Search-based refactoring for software maintenance. *The Journal of Systems & Software*, 81(4):502–516, 2008.

[69] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceerdings of Conference on Software Maintenance, 1992.*, pages 337 –344, 9-12 1992.

[70] OMG. *UML 2.4 Superstructure Specification*, (formal/2010-05-05) edition, 2010. http://www.omg.org/spec/UML/2.4/Superstructure/PDF/.

[71] W. F. Opdyke. Refactoring: A program restructuring aid in designing object-oriented application framework. *Ph.D. thesis, University of Illinois at Urbana-Champaign*, 1992.

[72] D. Parnas. Software aging. In *Proceedings of The 16th International Conference on Software Engineering (ICSE94)*, pages 279–287. IEEE Computer Society Press, 1994.

[73] E. Piveta, J. Araújo, M. Pimenta, A. Moreira, P. Guerreiro, and R.T. Price. Searching for opportunities of refactoring sequences: Reducing the search space. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 319–326. IEEE, 2008.

[74] F. Qayum and R. Heckel. Search-based refactoring using unfolding of graph transformation systems. *Electronic Communications of the EASST*, 38(0), 2011.

[75] A.J. Riel. Object-Oriented Design Heuristics. *Addison-Wesley*, 1999.

[76] R. Robbes, M. Lanza, and D. Pollet. A benchmark for change prediction. *Faculty of Informatics, Universit della Svizzerra Italiana, Lugano, Switzerland, Tech. Rep*, 6, 2008.

[77] R. Robbes, D. Pollet, and M. Lanza. Replaying ide interactions to evaluate and improve change prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 161–170. IEEE, 2010.

[78] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Urbana*, 51:61801, 1997.

[79] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Control flow analysis for reverse engineering of sequence diagrams. *Rapport Technique, Ohio State University*, 2004.

[80] Chris Seguin. *JRefactory*, 2003. http://jrefactory.sourceforge.net/csrefactory.html.

[81] O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1045–1051. ACM, 2005.

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

[83] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 30–38. IEEE, 2001.

[84] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 39(4):528–539, 2004.

[85] Friedrich Steimann and Jens von Pilgrim. Refactorings without names. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 290–293. ACM, 2012.

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

[87] N Tsantalis and A Chatzigeorgiou. Identification of extract method refactoring opportunities. *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering-Volume 00*, pages 119–128, 2009.

[88] N Tsantalis and A Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347 – 367, 2009.

[89] N. Tsantalis and A. Chatzigeorgiou. Ranking refactoring suggestions based on historical volatility. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 25–34. IEEE, 2011.

[90] H. Van Vliet. *Software Engineering: Principles and Practice*. Wiley, second edition, 2000.

[91] Ruediger Zarnekow and Walter Brenner. Distribution of cost over the application lifecycle - a multi-case study. In *Proceedings of the Thirteenth European Conference on Information Systems*, 2005.

[92] L. Zhao and J.H. Hayes. Predicting classes in need of refactoring: An application of static metrics. In *Proceedings of the workshop on predictive models of software engineering (PROMISE), associated with ICSM2006*, pages 1–5. Citeseer, 2006.

[93] Minhaz F Zibran and Chanchal K Roy. Conflict-aware optimal scheduling of code clone refactoring: a constraint programming approach. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 266–269. IEEE, 2011.

Chapter A. Behavioral Dependency Measure as a Good Indicator for Change-Proneness Prediction [39]

A.1 Behavioral Dependency Measure

Behavioral Dependency

A change in a class can affect other classes enforcing them to be modified. In order to predict the class affected when a class changes occurs, we need to examine the dependencies of pairs of entities (i.e., classes or objects) in the system. In this paper, we focus on *behavioral* dependency.

Essentially, we assume that the object sending a message has a behavioral dependency on the object receiving the message. This is derived from the insight that modifying the class of the object receiving a message may affect the class of the object sending the message. It is important to note that when an object sends a message to another object, the class implementing the corresponding method of the message may be different from that of the object receiving the message. This is due to the use of inheritance relationships and polymorphism, which may cause dynamic binding of methods. In this case, the class of the object sending a message must be bound to (i.e., have a behavioral dependency on) the class implementing the actual method of that message. Therefore, we need to consider inheritance relationships and polymorphism according to the behavior of objects in order to correctly identify dependencies between classes and ultimately predict change-proneness accurately. This issue will be explained in detail in Subsection A.1. We also assume that a high intensity of behavioral dependency represents high possibility of changes to be occurred. The rationale behind this assumption is that the more external services upon which the class of an object is dependent, the more likely it is that the class will be changed.

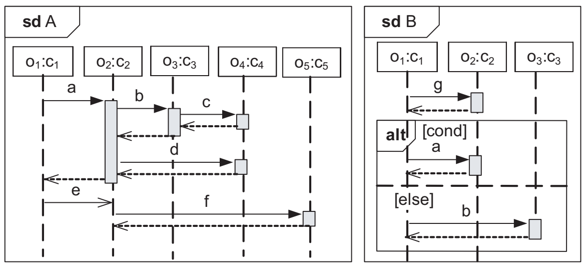


Figure A.1: Examples of Sequence Diagrams (SDs)

To quantify the behavioral dependency, we define two kinds of behavioral dependencies: direct and indirect. Each is defined as follows. Let O denote a set of objects existing in a system.

Definition 3 (Direct behavioral dependency)**.** *For op_1, $op_2 \in O$, op_1 has a direct behavioral dependency on op_2 if op_1 needs some services of op_2 by sending a synchronous message to op_2 and receiving a reply from op_2. We denote direct behavioral dependency as a relation $\rightarrow$.*

Definition 4 (External service request relation)**.** *For op_1, op_2, $op_3 \in O$, $op_1 \rightarrow op_2$ and then $op_2 \rightarrow op_3$ because op_1 needs external service which is provided from op_3 via op_2. We denote this as an external service request relation $\curvearrowright$. Therefore, in this case, ($op_1 \rightarrow op_2$) $\curvearrowright$ ($op_2 \rightarrow op_3$).*

Definition 5 (Indirect behavioral dependency)**.** *We denote indirect behavioral dependency as a relation $\rightsquigarrow$. For example, for op_1, op_2, op_3, ..., $op_n \in O$, if we have the relation ($op_1 \rightarrow op_2$) $\curvearrowright$ ($op_2 \rightarrow op_3$) $\curvearrowright$ ... $\curvearrowright$ ($op_{n-1} \rightarrow op_n$), then we can derive the indirect behavioral dependency as $op_1 \rightsquigarrow op_n$ except $n = 2$, which means a direct behavioral dependency $op_1 \rightarrow op_2$.*

A synchronous message entails a dependency between two objects since the sender object depends on the receiver object. On the other hand, an asynchronous message does not entail such dependency since the sender object does not wait for a reply but continues to proceed. This means the reply will not affect the sender object's behavior. Therefore, in our approach, we only consider synchronous messages with replies.

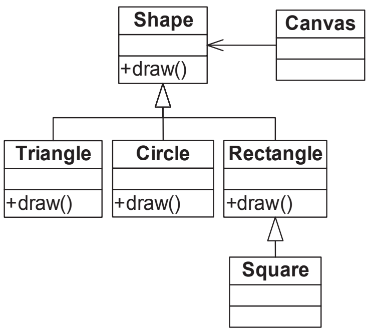
Fig. A.1 shows two examples of SDs. In SD sd A, object o_1 has a direct behavioral dependency on object o_2 because it sends a synchronous message a to object o_2 and receives a reply from it. On the other hand, object o_1 has an indirect behavioral dependency on object o_3; before object o_1 receives a reply for message a from object o_2, message b is sent from object o_2 to object o_3. By the same reasoning, object o_1 and object o_4, as well as object o_2 and object o_4, have indirect dependencies. Asynchronous message e from object o_1 to object o_2 does not entail a behavioral dependency since object o_1 (the sender) does not wait for a reply from object o_2. All messages in SD sd B cause direct behavioral dependencies.

It is important to note that an indirect behavioral dependency is not a transitive relation. For example, in Fig. A.1, object o_1 and object o_5 do not have a behavioral dependency, even though object o_1 and object o_2 have a behavioral dependency because of message a and object o_2 and object o_5 have a behavioral dependency because of message f. This is because message f is sent from object o_2 to object o_5 after object o_1 receives the reply for message a from object o_2. For this reason, we need to save the information of the message that triggers the current message to precisely identify the indirect behavioral dependency between the two objects. In this way, when object o_i has an indirect dependency on object o_j, we can derive a reachable path (a sequence of exchanged messages between two objects) by traversing stored messages from object o_j to object o_i in a backward direction.

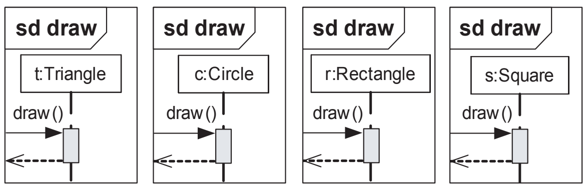
Features of the Behavioral Dependency Measure

The proposed BDM has a number of features that are different from existing metrics.

First, the most important feature of the BDM that makes it unique is that it considers inheritance relationships and polymorphism. In general, polymorphism indicates method overriding and method overloading. We do not take method overloading into account because it refers to methods that have same name with different numbers or types of parameters in one class; as a result, method overloading does not occur dependency among classes. Therefore, in this paper, polymorphism means method overriding on the classes having inheritance relationships. As the system contains more inheritance relationships and polymorphism, dependency among classes becomes more complex because of dynamic binding of methods. Hence, inheritance relationships and polymorphism as they relate to the behavior of objects need to be considered in order to correctly identify dependency among classes.

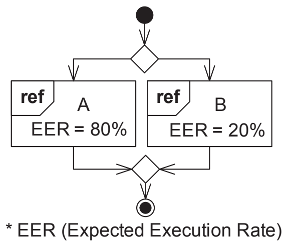


(a)

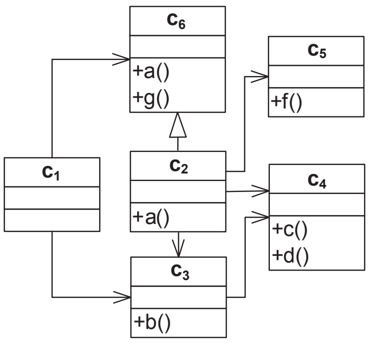


(b)

Figure A.2: An example of using inheritance relationships and polymorphism: (a) A class diagram representing classes and their relationships. (b) SDs representing the behaviors of objects that are instantiated from the classes in A.2(a).



(a)



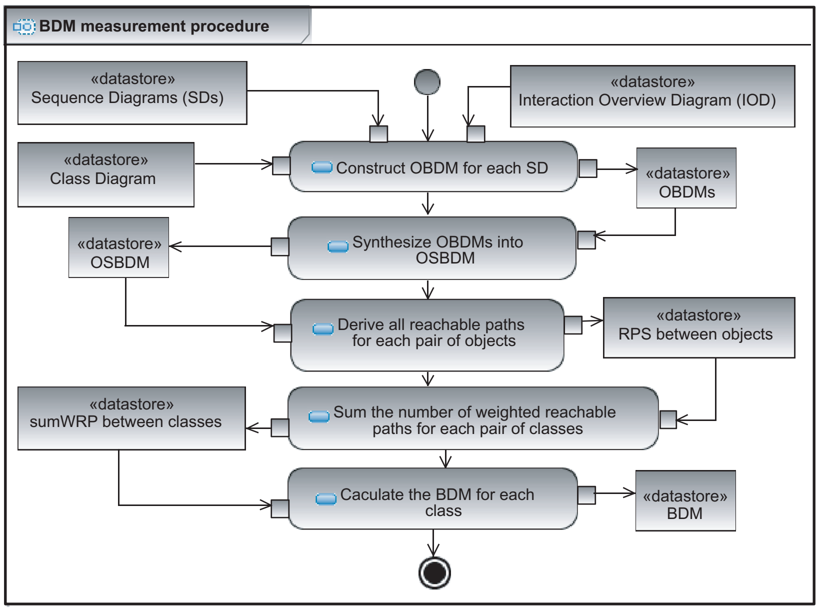
(b)

Figure A.3: (a) An example of the Interaction Overview Diagram (IOD). (b) An example of the class diagram that has classes from which the objects, in the SDs in Fig. A.1 are instantiated.

Indeed, this is critical for the accurate prediction of change-proneness. If we were not to consider inheritance relationships and polymorphism, a class may be mistakenly predicted to be prone to change. The example in Fig. A.2 shows the importance of considering inheritance relationships and polymorphism in relation to the behavior of objects when measuring dependency between classes. In the class diagram in Fig. A.2(a), the Canvas class has an association with the Shape class, which indicates that the Canvas class calls the method draw in the Shape class. In other words, the Canvas class is dependent on the Shape class. This static dependency is the information that we can derive from the class diagram. Most existing coupling metrics are measured based on static dependencies. However, if the message draw is sent to the object that is an instance of the subclass of the Shape class and that subclass overrides the method draw, the dependency is bound between the Canvas class and the subclass, even though the association is specified between the Canvas class and the Shape class. Furthermore, if the message draw is sent to the object that is an instance of the subclass of the Shape class but does not have the method draw, the dependency is bound between the Canvas class and the subclass's one of the parent classes that implement the method draw. The SDs in Fig. A.2(b) illustrate the behavior of the objects that are instances of subclasses (i.e., Triangle class, Circle class, Rectangle class, and Square class) of the Shape class in Fig. A.2(a). By considering the three foremost SDs, we can determine that the Canvas class is behaviorally dependent on the Triangle class, Circle class, and Rectangle class, all of which override the method draw. By considering the last SD, which tell us that the object of the Square class receives the message draw, we can also determine that the Canvas class is behaviorally dependent on the Rectangle class, since the method of the message draw is actually implemented in the Rectangle class. As a consequence, no matter where an actual method is implemented, the proposed BDM enables a class of the object sending a message to be bound to the class that implements the actual method of that message; this is the feature that makes the BDM more sensitive to systems with high levels of inheritance relationships and polymorphism.

Second, we consider the extent and direction when measuring the behavioral dependency. No matter how many times a class calls the method of another class, the established coupling metric (e.g., CBO) treats this as one in either direction. This is because the established coupling metric is based on method call dependencies that only capture the static characteristics of couplings. Let us consider two cases; class c_j implements one method called 100 times by class c_i, while class c_k implements two methods that called by class c_i once time for each method. The established static couplings for the former case and the latter case are one and two, respectively. However, class c_i might be more behaviorally dependent on class c_j than it is on class c_k. Therefore, it is important to keep the information relating to the extent and direction of a class's dependence.

Third, we consider the execution rate of the messages based on the control structure and the operational profile. We use two kinds of diagrams, an SD and an IOD, to depict a system's behavior. An SD in UML 2.0 provides combined fragments that allow us to express control structures such as branch and loop. An *alt* combined fragment that corresponds to a branch control structure describes the behavior of two or more mutually-exclusive alternatives. A message in an *alt* combined fragment can be executed depending on the condition. This may affect the behavioral dependency of objects that are related by this message. Without running a program (i.e., dynamic information), it is difficult to determine whether the message will be executed or not. Therefore, the probabilistic execution rate of a message is considered when measuring a behavioral dependency. For example, in the SD sd B of Fig. A.1, either message a or message b is executed whether the condition is satisfied or not (i.e., true or false).



BDM measurement procedure
«datastore» Sequence Diagrams (SDs)
«datastore» Interaction Overview Diagram (IOD)
«datastore» Class Diagram
Construct OBDM for each SD
«datastore» OBDMs
«datastore» OSBDM
Synthesize OBDMs into OSBDM
Derive all reachable paths for each pair of objects
«datastore» RPS between objects
«datastore» sumWRP between classes
Sum the number of weighted reachable paths for each pair of classes
Caculate the BDM for each class
«datastore» BDM

Figure A.4: Overview of our approach of BDM measurement

Therefore, the probabilistic execution rate of each message can be 0.5. An IOD in UML 2.0 illustrates an overview of a flow of control in which each activity node can be an SD. Some scenarios (i.e., SDs) might be executed more frequently than others, as specified in the operational profile [35]. The operational profile provides the expected execution rate of an SD. Therefore, the operational profile also needs to be considered for the better measurement of the behavioral dependency. We suggest specifying the Expected Execution Rate (i.e., the operational profile) of each SD in an IOD. For example, the IOD in Fig. A.3(a) shows that the Expected Execution Rates of SD A and SD B are 80% and 20%, respectively.

A.2 Procedure for Behavior Dependency Measure Measurement

In this section, we explain a systematic way of calculating the BDM in UML design models using SDs, a class diagram, and an IOD. An overview of our approach is shown in Fig. A.4. The BDM is computed through the following procedures. First, Object Behavioral Dependency Model (OBDM) is constructed for each SD based on all direct and indirect behavioral dependencies between objects by referring to the class diagram and the IOD. After that, we synthesize all OBDMs into the Object System Behavioral Dependency Model (OSBDM) for the entire system. Next, we derive all the reachable paths for each pair of objects in the system from the OSBDM. We then sum the weighted reachable paths for each pair of classes (a reachable path is weighted using the distance length between objects and the execution rate of the messages of which the reachable path is composed). Finally, we calculate the BDM for every class in the system. Detailed procedures are described in the following subsections.

Constructing OBDMs

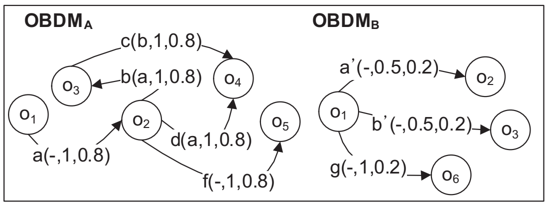
A dependency model OBDM$_A$ for SD A is a 2-tuple (O, M), where

- O is a set of nodes representing objects in the SD
- M is a set of edges representing messages that are exchanged between two nodes. Message $m \in M$ represents a synchronous message with a reply, which entails a direct dependency from a sender object to a receiver object. Message m has following six attributes.
  - $m_s \in O$ is the sender of the message.
  - $m_r \in O$ is the receiver of the message. $m_r \neq m_s$.
  - m_n is the name of the message.
  - $m_b \in M$ is the instance of a backward navigable message. $m_b \neq m$. "-" means none.
  - m_{meL} is the probabilistic message execution rate in an SD. $0 \leq m_{meL} \leq 1$. The default value is 1.
  - m_{meH} is the expected message execution rate in an IOD. $0 \leq m_{meH} \leq 1$. The default value is 1.

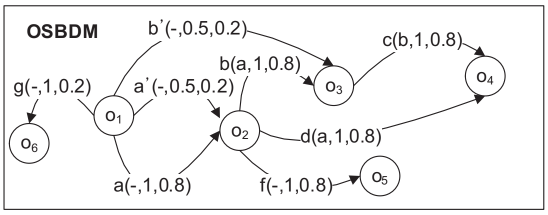
m_s and m_r represent the sender and receiver objects, respectively. Since we do not consider messages from an object to itself, they should not represent the same node. As was pointed out in Subsection A.1, when an object sends a message to another object, the class of the object receiving a message may be different from the class implementing the corresponding method. In such a case, the class of the object sending the message may change when the implemented method changes. Therefore, when binding a receiver node, it is important to note whether the method is actually implemented in the class of the receiver object. If not, the receiver node of the message is bound to an object of a parent class that actually implements the method.

m_b represents the message that triggers m and is called a backward navigable message. As was noted in Subsection A.1, m_b is essential for identifying indirect behavioral dependencies between objects. We can identify the message that activates the current message by tracing the backward navigable message. When deriving a reachable path from the OSBDM, identification of the message that triggers m prevents infinite loop of traversing.

As was described in Subsection A.1, m_{meL} and m_{meH} help to better predict the change-proneness of classes by considering the probabilistic or expected execution rates of the messages. Later, these rates are synthesized according to a reachable path and used to measure behavioral dependency. m_{meL} represents the probabilistic message execution rate in an SD. We consider a branch control structure that might affect the probability of the message execution. Note that a branch control structure is represented as an *alt* combined fragment in UML 2.0. When a message is in an *alt* combined fragment, it is executed only when a condition of the corresponding interaction operand is met. Therefore, m_{meL} is the same as the probability that one of the interaction operands that contain the message is selected. If an *alt* combined fragment is nested, the probability that a message will be executed in the corresponding combined fragment is multiplied to m_{meL} recursively. When a message is not contained in any combined fragments, its m_{meL} is 1. m_{meH} represents the expected message execution rate in an IOD. We specify the Expected Execution Rate (i.e., the operational profile) of each SD in an IOD. A message in an SD is executed only when the corresponding SD is activated. Therefore, m_{meH} is the same as the probability



(a)



(b)

Figure A.5: (a) OBDM$_A$ and OBDM$_B$ correspond to SD sd A and SD sd B in Fig. A.1. (b) The obtained OSBDM by synthesizing the two OBDMs in (a).

that the control flow of the software reaches the SD to which the message belongs. The m_{meH} values can be obtained by multiplying all the Expected Execution Rates on the way from the initial node to the corresponding SD node in the IOD. If an SD is always activated, the m_{meH} values of all the messages in the SD are 1.

Fig. A.5(a) shows an example of two OBDMs constructed from SD sd A and SD sd B in Fig. A.1. Each node of object o$_i$, $1 \leq i \leq 6$, corresponds to an instance of class c$_i$ in Fig. A.3(b). Each edge of message m is represented as $m_n(m_b, m_{meL}, m_{meH})$. Due to inheritance relationships and polymorphism, which may bring about dynamic binding of methods (Subsection A.1), we examine the behavior of objects in SDs and the structures of classes in a class diagram in order to correctly identify dependency between classes. In other words, when an object sends a message to another object, we first check whether the method of the message is implemented in the class of the receiving object or in one of its parent classes; we then create an edge corresponding to the message between the node of the object sending the message and the object of which the class actually implemented the method of the message. For instance, when object o$_1$ sends message a to object o$_2$, just as in SD sd A, we create the edge of message a between node o$_1$ and node o$_2$ in OBDM$_A$ since class c$_2$ overrides method a. On the other hand, when object o$_1$ sends message g to object o$_2$, just as in SD sd B, we create the edge of message a between node o$_1$ and node o$_6$ in OBDM$_B$ since class c$_6$ actually implements method g. To distinguish the messages in OBDM$_B$ from those in OBDM$_A$, we rename message a to a’ and b to b’. Since either the message a’ or b’ may be activated depending on the condition of the *alt* combined fragment, both a’$_{meL}$ and b’$_{meL}$ are 0.5. The Expected Execution Rates of SD A and SD B are represented in the IOD in Fig. A.3(a) and are reflected in the execution rates of the messages as 0.8 and 0.2, respectively.

Algorithm 6 retrieveReachablePathSet($o_1, o_2 : O$)

```
input $OUT \leftarrow$ outgoing message set of $o_1$
input $IN \leftarrow$ incoming message set of $o_2$
input $RP \leftarrow \emptyset$ an array for storing reachable path
        /*RP denotes a reachable path*/
input $RPS \leftarrow \emptyset$ a vector for saving a set of reachable paths
output $RPS$

for all $in \in IN$ do
  for all $out \in OUT$ do
    if $in == out$ then
      /*For RPS by the Direct Behavioral Dependency*/
      $RPS \leftarrow RPS \cup \{in\}$
    else
      /*For RPS by the Indirect Behavioral Dependency*/
      $RP \leftarrow RP + \{in\}$
      while $in_b != out$ && $in_b != \emptyset$ do
        if $in == out$ then
          $RP \leftarrow RP + \{in_b\}$
          $RPS \leftarrow RPS \cup RP$
          $RP \leftarrow \emptyset$
          break
        else
          $RP \leftarrow RP + \{in_b\}$
          $in \leftarrow in_b$
```

Synthesizing OBDMs into an OSBDM

To determine the behavioral dependencies between objects in the whole system, we synthesize OBDMs into $OSBDM = (O_s, M_s)$. O_s and M_s denote the set of objects and the union of messages that exist in the system, respectively. The method for constructing the OSBDM will be explained using the example in Fig. A.5. Fig. A.5(b) shows the obtained OSBDM by synthesizing the two OBDMs in Fig. A.5(a). This OSBDM is composed of $\mathsf{O_s} = \{\mathsf{o}_1, \mathsf{o}_2, .., \mathsf{o}_6\}$ and $\mathsf{M}_s = \{(m \in M$ of SD sd A$) \cup (m \in M$ of SD sd B$)\}$. Note that object o_1 in SD sd A and object o_1 in SD sd B are instantiated from the same class c_1. Therefore only one o_1 remains in the OSBDM. The sending message a from o_1 in SD sd A and another sending message a' from o_1 in SD sd B are connected with the corresponding target object o_2 in the OSBDM. If a message m is triggered by another message in the context of the system by examining the IOD, we set this other message as a backward navigable message of message m. There is no such case in this example.

Deriving Reachable Paths

We derive all reachable paths for each pair of objects in the system from the OSBDM. Let $RPS(o_i, o_j) = \{s|$ s is a reachable path between source object o_i and target object $o_j\}$ be a set of all the reachable paths between two objects. To retrieve the $RPS(o_i, o_j)$, we start traversing of the OSBDM from a message incoming to object o_j to a message outgoing from object o_i in reverse. When object o_i has a direct behavioral dependency on object o_j, one of the incoming messages to object o_j and one of the outgoing messages from object o_i are equal. This

Table A.1: $RPS(\mathsf{o_i}, \mathsf{o_j})$, which is a set of all the reachable paths for each pair of objects (Row: $\mathsf{o_i}$, Column: $\mathsf{o_j}$) in Fig. A.5(b).

	o$_1$	o$_2$	o$_3$	o$_4$	o$_5$	o$_6$
o$_1$	-	{a,a'}	{ab,b'}	{abc,ad}	-	{g}
o$_2$	-	-	{b}	{bc,d}	{f}	-
o$_3$	-	-	-	{c}	-	-
o$_4$	-	-	-	-	-	-
o$_5$	-	-	-	-	-	-
o$_6$	-	-	-	-	-	-

message is then added into a set of reachable paths. On the other hand, when object o_i has an indirect behavioral dependency on object o_j, we traverse the OSBDM from one of the messages incoming to object o_j iteratively by substituting it with a backward navigable message. In doing this, we finally reach one of the outgoing messages from object o_i. The stored sequence of messages encountered while traversing is the reachable path. The method for retrieving a reachable path set from object o_1 to object o_2 is presented in Algorithm 6. The set of reachable paths for each pair of objects in Fig. A.5(b) is presented in Table A.1.

Summing Weighted Reachable Paths

Prior to calculating the BDM for every class in the system, we sum the weighted reachable paths for each pair of objects using the RPS obtained above. In this process, an object is projected onto the class from which the object is instantiated. In this manner, the results of the summation of the weighted reachable paths are obtained for each pair of classes.

We formalize the sum of the Weighted Reachable Paths (SumWRP) from class c_i to class c_j as follows:

$$SumWRP(c_i, c_j) = \sum_{\forall s \in RPS(o_i, o_j)} DF(s) \times f_{meL} \times f_{meH}, \tag{A.1}$$

where o_i and o_j indicate the objects that correspond to projected instances of class c_i and c_j, respectively. We use three factors for weighting reachable path s: distance factor, f_{meL}, and f_{meH}. We define a distance factor by $DF(s) = 1/d$, where d is the distance length (i.e., the number of messages in the corresponding reachable path s). The rationale for using the distance factor is that an indirect behavioral dependency might be weakened by the successive calls. In other words, the farther an object is from the source of changes, the less the object is likely to be changed. Therefore, we need to degrade the impact when the distance of indirect behavioral dependency between two objects is great. We represent the first message in the reachable path s as f. Then, f_{meL}, the probabilistic message execution rate, and f_{meH}, the expected message execution rate, are taken into account as factors for weighting a reachable path.

We explain how to calculate SumWRP using Table A.1. To calculate $SumWRP(\mathsf{c}_2,\mathsf{c}_4)$, for example, we first obtain reachable paths, {bc,d}, between object o_2 and object o_4. We then identify weighting factors for each reachable path. For reachable path bc, the distance factor is $1/2$, because the number of messages in this

Table A.2: $SumWRP(\mathsf{c_i},\mathsf{c_j})$, which is the sum of the weighted reachable paths for each pair of classes (Row: $\mathsf{c_i}$, Column: $\mathsf{c_j}$) and the $BDM(\mathsf{c_i})$ value of each class in Fig. A.3(b).

	c_1	c_2	c_3	c_4	c_5	c_6	$BDM(c_i)$
c_1	0	0.9	0.5	0.67	0	0.2	2.27
c_2	0	0	0.8	1.2	0.8	0	2.8
c_3	0	0	0	0.8	0	0	0.8
c_4	0	0	0	0	0	0	0
c_5	0	0	0	0	0	0	0
c_6	0	0	0	0	0	0	0

reachable path is 2. The first message of this reachable path is b. Therefore, $\mathsf{b_{meL}}$, 1, and $\mathsf{b_{meH}}$, 0.8, are applied for weighting the reachable path. For reachable path d, the distance factor is 1; $\mathsf{d_{meL}}$ and $\mathsf{d_{meH}}$ are 1 and 0.8, respectively. Finally, we sum the weighted reachable paths and obtain $SumWRP(\mathsf{c_2},\mathsf{c_4})$ as follows:

$$SumWRP(\mathsf{c_2}, \mathsf{c_4}) = (1/2 \times 1 \times 0.8) + (1 \times 1 \times 0.8) = 1.2$$

Calculating the Behavioral Dependency Measure

Finally, the BDM for every class c_i in the system is obtained as follows. Let $C = \{c_i | 1 \leq i \leq n\}$ be all the classes existing in the system.

$$BDM(c_i) = \sum_{\forall c_j \in C,\ i \neq j} SumWRP(c_i, c_j). \tag{A.2}$$

Table A.2 summarizes the sum of the weighted reachable paths obtained from the OSBDM in Fig. A.5(b) and the BDM of each class in Fig. A.3(b). The BDM is used to predict change-proneness; the higher the class's BDM, the larger the likelihood the class will be changed.

A.3 Change-Proneness Modeling

In this section, we describe the method for building a change-proneness prediction model. In our study, the change-proneness is used for predicting change-prone classes in the successive versions.

Model Construction Method

To build the change-proneness prediction model, there are a large number of modeling techniques from which to choose, including standard statistical techniques (e.g., logistic regression) and data mining techniques (e.g., decision trees [41]). Multiple linear regression provides a regression analysis of variance for a dependent variable explained by one or more factor variables. Hence, we choose a stepwise multiple regression [26] to build the change-proneness model in this study. While constructing the regression, we remove outliers that are clearly over-influential on the regression results. Two kinds of techniques can be used for outlier analysis: Standard errors of the predicted values (S.E. of mean predictions) and the Mahalanobis distance [60]. The former is an estimate

Table A.3: The number of ground facts regarding 13 subsequent versions of JFlex (versions 1.3 to 1.4.3).

	1.3	1.3.1	1.3.2	1.3.3	1.3.4	1.4pre1	1.4pre3	1.4pre4	1.4pre5	1.4	1.4.1	1.4.2	1.4.3
Package	3	3	3	4	4	4	4	5	5	5	5	5	5
Class	44	44	44	48	48	47	47	61	59	59	59	62	62
Interface	4	4	4	4	4	4	4	3	3	3	3	4	4
Invoked Message	1768	1828	1828	2042	2048	2071	2135	2426	2418	2401	2376	2651	2651
Reachable Path	1394	1442	1442	2335	2339	2362	2282	3244	3244	3249	3242	3317	3319

of the standard deviation of the average value for dependent variable for cases that have the same values with the independent variables. The latter is a measure of how much a case's values on the independent variables differ from the average of all cases; case means a data instance for constructing a prediction model. Hence, we identify and remove the instances that have extremely large S.E. of mean predictions and large Mahalanobis distance values.

Model Variables

We first collected several data types from the object-oriented software.

The independent variables include the C&K metrics, Lorenz and Kidd metrics, MOOD metrics, and the BDM. We collect the C&K metrics and Lorenz and Kidd metrics using [12]. These are the most widely used metrics for evaluating object-oriented software. The set of metrics used in the case study are listed in the appendix. To calculate the BDM, which is measured on UML models, we have developed a tool built on the EMF (Eclipse Modeling Framework). It imports the UML 2.0 models in the format of XMI generated from [48], an Eclipse-based UML 2.0 modeling tool made by the Rational Division of IBM.

Following a common analysis procedure [5], we first perform a Principal Component Analysis (PCA) to identify the dimensions actually present in the data relating to the independent variables. We do not make use of a PCA to select a subset of independent variables since, as discussed in [15], experience has shown that this usually leads to suboptimal prediction models even though regression coefficients are easier to interpret. The resulting principal components can be described in terms of categories such as size, complexity, cohesion, coupling, inheritance and polymorphism (see the appendix).

The dependent variable of the model is the change-proneness. To compute the change-proneness, the change data, which are obtained using a class-level source code *diff*, are collected for each application class. Based on this change data, the total amount of changes (i.e., source lines of code added and deleted) within consequent releases are measured.

A.4 Case Study

This section presents the results of a case study, the objective of which is to validate the usefulness of the BDM presented above. The first subsection explains the details of the system. In the next subsection, the goal of the case study and the validation method are described. In the third section, results are presented and interpreted. The last subsection ends with a discussion.

The Subjects

In order to investigate whether the BDM is statistically related to change-proneness, we need a target system that has well documented UML models with a class diagram, SDs an IOD, and subsequent releases for extracting change-related information. For our experiment, we reverse-engineered the UML design models from the existing system, JFlex, using a reverse engineering tool with manual supports. JFlex is a lexical analyzer generator for Java, which is written in Java. JFlex takes a specially formatted specification file containing the details of a lexical analyzer as input and creates a Java source file for the corresponding lexical analyzer. A number of reasons led us to select JFlex for the case study:

- It has evolved through 14 generations (at the time that we conducted this case study) and recorded the history of changes.
- The full source code of each version is available because it is an open-source project.
- It contains a relatively large number of classes.
- It is mature. The release dates are February 20, 2001 for the initial version (version 1.3) and January 31, 2009 for the latest version (version 1.4.3).
- It was written in Java. Our BDM is applied in object-oriented software that uses inheritance relationships, so polymorphism and dynamic binding may occur.

Among the 14 releases of JFlex, version 1.3.5 is not considered in the case study because the changes made between this version and version 1.3.4 are negligible. Table A.3 represents the number of ground facts regarding the 13 successive versions of JFlex. The initial version of JFlex 1.3 consists of 44 Java classes and 1394 reachable paths, while the latest version of JFlex 1.4.3 consists of 62 Java classes and 3319 reachable paths. The total number of reachable paths can be less than the total number of invoked messages in each version of the system in the following cases: (1) invoked messages for which call methods from the library are not considered (the scope of the measurement is limited to the application classes of JFlex) and (2) invoked messages for which call methods within the same class are not considered, since these internal messages do not cause behavioral dependency.

We collected several types of data (i.e., existing object-oriented software metrics, the BDM, and change data) for each class from nine versions of JFlex based on reverse-engineered models. It should be noted that we collected metrics that are available on design models. In other words, we did not gather metrics that are obtainable only from source codes, such as source lines of code (SLOC), number of fields (NOF), and number of parameters (NOP). To select classes with a long history of changes, we included the classes from the initial version that remain in the latest version (i.e., 42 classes for each version of JFlex). For each version on which the BDM and other metrics are measured, the change data was measured by counting the total number of changes in the next four subsequent

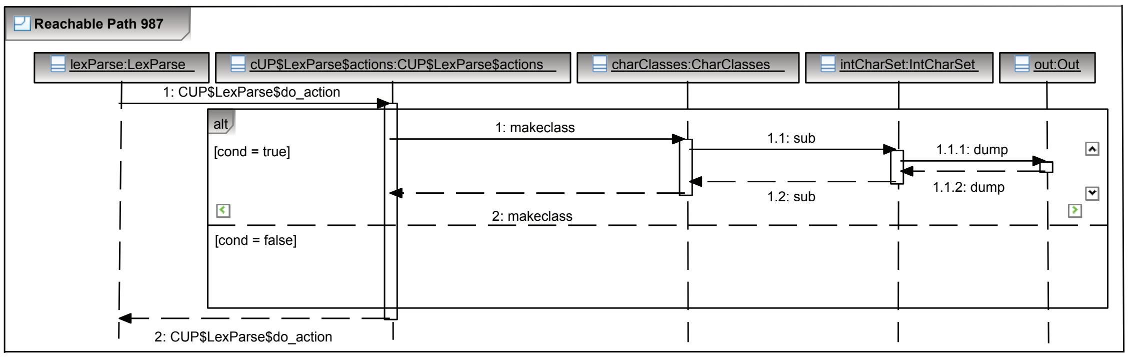


Figure A.6: An SD that was reverse-engineered from source codes of JFlex version 1.3.

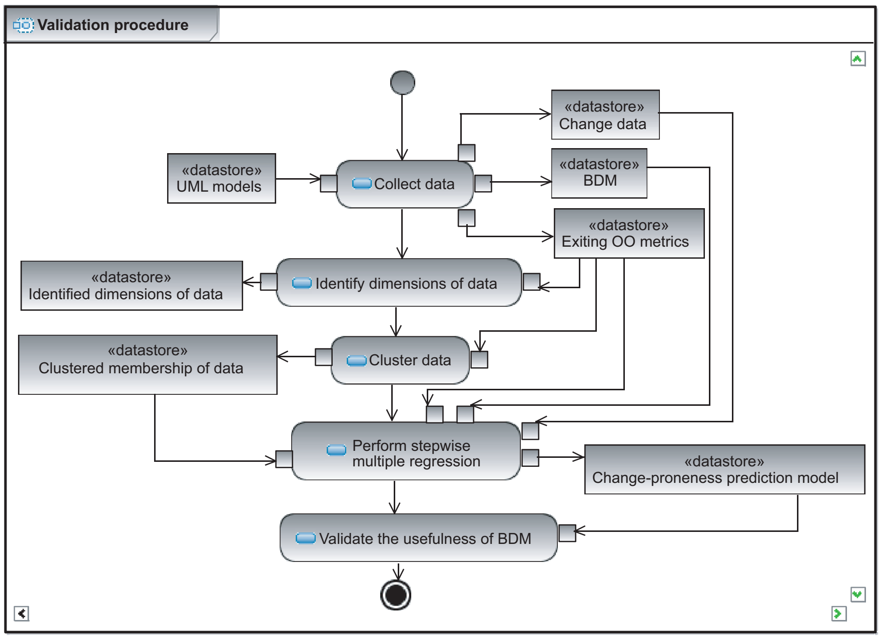

Validation procedure
«datastore» Change data
«datastore» UML models
Collect data
«datastore» BDM
«datastore» Exiting OO metrics
«datastore» Identified dimensions of data
Identify dimensions of data
«datastore» Clustered membership of data
Cluster data
Perform stepwise multiple regression
«datastore» Change-proneness prediction model
Validate the usefulness of BDM

Figure A.7: A validation procedure followed during in this case study

versions. This change data is used as change-proneness. We take 9 of the 13 releases into account because the change data is not available in the last four versions; we finally obtained 378 instances of classes.

We easily reverse-engineered the class diagram from JFlex source code. On the other hand, reverse-engineering SDs is difficult and sometimes even impossible [14], because an SD represents the partial behavior of the overall system; SDs can exist in various forms according to the various users' view on the system. Thus, in this case study, we construct the SD for each reachable path that consists of consecutive invoked messages, while extracting the structural information from source codes and reflecting it in the SD as *alt*, *opt*, or *loop* combined fragments. It should be noted that the SD in UML 2.0 uses the combined fragments to represent one or more sequences (traces) rather than specifying all the possible scenarios [79]. Hence, we do not need to execute the system and monitor its execution to retrieve meaningful information and reverse-engineer SDs from source codes. Fig. A.6 shows an example of the reverse-engineered SD obtained from JFlex version 1.3. This SD corresponds

Table A.4: The analysis-of-variance (ANOVA) table that includes the results for each clustering variable

	Cluster		Error			
	Mean Square	df	Mean Square	df	F	Sig.
MIF	320.333	1	.439	376	729.972	.000
PF	7500.000	1	.383	376	19583.333	.000
DIT	.000	1	2.168	376	.000	1.000
NOC	.593	1	.673	376	.881	.349

Table A.5: Descriptive statistics with respect to the four attributes for Group 1 (294 instances).

	N	Range	Minimum	Maximum	Sum	Mean		Std.	Variance	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error	Statistic	Statistic	Statistic	Std. Error
PF	294	2	36	38	11088	37.71	.041	.701	.491	-2.052	.142
MIF	294	2	70	72	20958	71.29	.041	.701	.491	-.462	.142
NOC	294	5	0	5	84	.29	.051	.882	.778	4.033	.142
DIT	294	5	0	5	210	.71	.086	1.471	2.164	1.864	.142

to the reachable path from the object of the LexParse class to the object of the Out class with four messages: CUP$LexParse$do_action, makeClass, sub, and dump. By analyzing source code from JFlex 1.3, for example, we extracted 1394 reachable paths and, at the same time, constructed 1394 SDs with 1768 messages. The IOD cannot be reversed from source codes; it is specified only from the early stages of software development to help developers get an overview of the system. Thus, in this experiment, the Expected Execution Rate in the IOD was not considered when calculating the BDM.

Goal and Validation Methodology

The goal of this case study is to confirm that the BDM is a significant additional explanatory variable over and above that which has already been accounted for by other existing metrics when the system contains complex inheritance relationships and polymorphism. It should be noted that the BDM considers dynamic features (see Subsection A.1). As a result, we expect the BDM to more accurately predict behavioral dependency when the system is involved in a complex dynamic binding occurrence environment. Indeed, accurate prediction of behavioral dependency helps to construct a better change-proneness prediction model. In order to achieve this goal, the validation procedure depicted in Fig. A.7 was performed.

To investigate whether the effect of the BDM is different according to intensity of use of inheritance relationships and polymorphism, we divide the data set into two groups and independently build change-proneness prediction models for each group. We cluster the data into the following groups:

- (Group 1) contains comparatively more complex inheritance relationships and polymorphism.
- (Group 2) contains comparatively less complex inheritance relationships and polymorphism.

To classify the data set into these two groups, a clustering technique was applied. Clustering is also called data

Table A.6: Descriptive statistics with respect to the four attributes for Group 2 (84 instances).

	N	Range	Minimum	Maximum	Sum	Mean		Std.	Variance	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error	Statistic	Statistic	Statistic	Std. Error
PF	84	0	26	27	2268	27.00	.000	.000	.000	.000	.263
MIF	84	1	73	74	6174	73.50	.055	.503	.253	.000	.263
NOC	84	2	0	2	16	.19	.060	.548	.301	2.782	.263
DIT	84	5	0	5	60	.71	.161	1.477	2.182	1.888	.263

segmentation and is used to partition large data sets into groups according to similarities. Clustering may serve as a preprocessing step for classification, which would then operate on the detected clusters and the selected attributes or features [41]. We used K-means clustering [19] with four inheritance- and polymorphism-related metrics: PF (Polymorphism Factor), MIF (Method Inheritance Factor), NOC (Number of Children), and DIT (Depth of Inheritance). Table A.4 represents the analysis-of-variance (ANOVA) table that includes the results for each clustering variable. At the α = 0.05 significance level, MIF and PF are significant explanatory variables to divide the groups since Sig. (p-value) $\approx 0.000 \leq 0.05 = \alpha$. The descriptive statistics with respect to the four attributes are provided for Groups 1 and 2 in Tables A.5 and A.6, respectively. The classification results show that the classes in Group 1 have higher PF and NOC values than those in Group 2. Therefore, Group 1 can be characterized as having more complex inheritance relationships and polymorphism than Group 2. As there is not much difference in MIF values and no difference in DIT values between the two groups, we did not consider these two attributes for identifying each group's characteristics. In analyzing the cluster membership of the classes in JFlex across the nine versions, we noticed that the classes that developed earlier tend to belong to Group 1, while the classes that developed later tend to belong to Group 2. In other words, the JFlex system has evolved in that it now has less complex inheritance relationships and polymorphism.

For Groups 1 and 2, we construct two change-proneness prediction models: one between the change and existing metrics and the other between the change and the BDM in addition to existing metrics. Consequently, we analyzed four change-proneness prediction models in total. To validate the assertion that the BDM helps to explain additional variations in change-proneness, we compare the goodness-of-the-fit of those two models. As a result, it is clear that the BDM contributes to obtain a better model fit. The results of the change-proneness prediction models are presented and discussed in detail in the next subsection.

Results

[Results of the change-proneness prediction models.]

We evaluated the performance of the prediction models according to the goodness-of-the-fit (R-square) and a sequence of the selection of independent variables. The sequence of the selection is important because the independent variable that has the largest positive or negative correlation with the dependent variable is selected at each step in a stepwise selection.

We first present the results of the two regression models obtained from the data set of Group 1. The results of

Table A.7: Prediction model using existing metrics in Group 1.

	Unstandardized coefficients		Standardized coefficients		
Selected variables	B	Std. Error	Beta	t	Sig.
(Constant)	.050	.054		.915	.361
COC	.050	.011	.215	4.695	.000
PProtM	-.010	.004	-.122	-2.809	.005
NOC	.199	.042	.204	4.715	.000
WMC	-.019	.004	-.399	-4.444	.000
MNOL	.114	.030	.249	3.748	.000

the stepwise regression using C&K metrics, Lorenz and Kidd metrics, and MOOD metrics as candidate covariates are presented in Table A.7. The prediction model included five variables. The model explains around 57 percent of the variance of the data set and shows an adjusted R^2 of 0.54. The sequence of variables entering into the model is COC, PProtM, NOC, WMC, and MNOL. The result after including the BDM in addition to existing metrics to make a prediction model, we obtain the result as shown in Table A.8. Around 64 percent of the variance in the data set is explained and an adjusted R^2 of 0.63 is obtained. In this model, COC, BDM, PProtM, NOC, MNOL, WMC, and NORM variables were included in the order of the sequence as listed. Note that the BDM is the second variable to be included with significant-level of p-value < 0.00001. Even when accounting for the difference in the number of covariates, the coefficient of determination (R^2) is increased by 9 percent (from 0.54 to 0.63) when using the BDM. Therefore, this experiment shows that the BDM helps to obtain a better change-proneness prediction model. In other words, even though the existing metrics still do most of the lifting, the BDM captures additional dimensions that enable the construction of a more accurate change-proneness prediction model.

From the data set of Group 2, we also constructed two regression models, one using only existing metrics for the baseline of the comparison and the other using the BDM in addition to existing metrics, to investigate whether the effect of the BDM performs differently according to the intensity of inheritance relationships and polymorphism. In this group, the BDM was not included when constructing the regression model. Hence, the results of the two models are the same. Table A.9 shows the results of the prediction model obtained from Group 2. This model explains the change variance of around 75 percent and shows an adjusted R^2 of 0.74. The goodness-of-the-fit of the prediction model in Group 2 is considerably higher than that of the prediction models in Group 1 because smaller data instances were used to create the model.

[Interpretation of Results.]

In the experiment, we divided the data set into two groups, one with comparatively more complex and the other with comparatively less complex inheritance relationships and polymorphism, in order to investigate the effects of the BDM according to the intensity of inheritance relationships and polymorphism. Intensity of use of inheritance relationships and polymorphism can be explained through the attributes (i.e., inheritance- and polymorphism-related metrics) used for K-means clustering groups; PF and NOC were used for characterizing each group, as mentioned in Subsection A.4. PF equals the number of actual method overrides divided by the

Table A.8: Prediction model using existing metrics and *BDM* in Group 1.

	Unstandardized coefficients		Standardized coefficients		
Selected variables	B	Std. Error	Beta	t	Sig.
(Constant)	.048	.053		.894	.372
COC	.050	.011	.214	4.632	.000
BDM	.019	.008	.108	2.431	.000
PProtM	-.012	.004	-.144	-3.263	.001
NOC	.202	.042	.206	4.811	.000
MNOL	.120	.029	.264	4.114	.000
WMC	-.017	.004	-.361	-3.931	.000
NORM	.008	.003	.184	2.465	.014

Table A.9: Group 2 prediction model (The result is same whether the BDM is used or not because the BDM is not included in the model).

	Unstandardized coefficients		Standardized coefficients		
Selected variables	B	Std. Error	Beta	t	Sig.
(Constant)	-1.211	.397		-3.052	.003
NOO	-.013	.003	-.307	-4.280	.000
CL	.021	.004	.571	5.393	.000
NOIS	.816	.107	1.252	7.615	.000
RFC	-.139	.024	-.798	-5.832	.000

maximum number of possible method overrides and is calculated as a fraction. The PF value increases as the system uses method overriding; if the system overrides everything, the PF is 100%. If subclasses seldom override their parent's methods, PF will be low. NOC equals the number of immediate subclasses derived from a base class and measures the breadth of a class hierarchy. Conversely, DIT measures the depth. Therefore, it can be concluded that Group 1 contains relatively complex inheritance relationships and polymorphism than Group 2 since the former has higher PF and NOC values than the latter.

It was determined that the BDM is a significant indicator as it helped to improve the accuracy of change-proneness prediction in Group 1 only. This result was anticipated because in Group 1, the system redefines the parent's methods more often(i.e., high PF) and inherits parent classes more often (i.e., high NOC) than the system, in Group 2. In short, Group 1 contains high degree of inheritance relationships and polymorphism, which may be the reason for the high probability that dynamic binding will occur. In Group 2, the BDM could not be selected as a variable to explain the variances in change-proneness. This indicates that the BDM is no more useful than existing metrics in systems that contain low degree of inheritance relationships or polymorphism. By analyzing the results of the experiment, we reached the conclusion that the BDM can help accurately predict changes when the system contains high degree of inheritance relationships and polymorphism. The reason for improving the accuracy of change-proneness prediction is the BDM's feature enabling a class of the object sending a message to be bound to the class that actually implements the method of the message, as mentioned in Subsection A.1. To put the point another way, the BDM is a specific measure for a consideration of the dynamic behavior of the system.

Discussion

In the case study, we showed that the BDM is the significant indicator for predicting change-proneness when the system contains high degree of inheritance relationships and polymorphism. However, it is not possible to determine the exact thresholds of the system's high degree of inheritance relationships and polymorphism because these thresholds are relative and empirical. However, we do not need specific guidelines that tell us when to use the BDM in change-proneness prediction. The BDM is an additional variable that may be used in conjunction with existing metrics for explaining variance in change-proneness for systems where dynamic binding is likely to occur. When constructing a change-proneness prediction model, the BDM is not selected if it cannot capture any features over and above those captured by existing metrics. In other words, the BDM is selected as a significant variable only if it helps to improve the accuracy of change-proneness prediction in addition to existing metrics.

In the case study, we used reversed UML models that were obtained from source codes, even though model-based change-proneness prediction was the goal of the study. This is because, in practice, most legacy systems which have been developed and maintained for a long period of time do not have well documented design models, especially for SDs and IODs. It is worth reiterating that an IOD is specified from the early stages of software development and cannot be reversed from source codes. If an IOD specified with the Expected Execution Rate in each SD is available, a more accurate BDM may be obtained. In the future, we plan to use the UML models which will soon be available from the Repository for Model-Driven Development (REMODD) project [30] in order to explore the usefulness of the BDM for model-based change-proneness prediction.

The fitness of the models for model-based change-proneness prediction is rather low compared to models for code-based change-proneness prediction. This is because the information extracted from UML models is not as sufficient for change-proneness prediction as the information from source codes. If other metrics derivable from only source codes were considered when building the change-proneness prediction model, the R^2 values would be higher. For example, SLOC, which indicates the size of the class, is known as a significant indicator to affect change-proneness [4]. Of course, the goal of this study is to determine whether the BDM helps to obtain a better model fit. Therefore, we need to see that it provides an improved predictive model when compared to models considering only existing metrics that are available on UML models. This provides the benefit of early change-proneness prediction at the moment a design model becomes available, without the necessity of implementing source codes.

The results from our earlier study [40] on JFreeChart [34] also showed that the BDM is a strong indicator and complementary to C&K metrics for explaining the variance of changes. In this paper, we performed the new experiment to compare the effect of the BDM with varying degrees of inheritance relationships and polymorphism. We used another system, because complex dynamic bindings may not occur in the system examined in the previous study, JFreeChart, since it is the graphic library for generating various types of charts. In other words, in JFreeChart, dependencies occurred by method calling would be simple. In our previous paper, we only used C&K metrics as the existing metrics. However, in this case study, we used more metrics to confirm that the BDM is effective with regard to change-proneness prediction.

요 약 문

객체지향 소프트웨어의 유지보수성 향상을 위한 리팩토링 식별 및 선택 방법에 대한 연구

객체지향 소프트웨어서는 유지보수 관련한 다양한 활동들을 지원하기 위하여 계속해서 변경이 일어나는데, 이러한 변경은 소프트웨어의 품질을 현격히 저하시킨다. 이러한 문제를 향상시킬 수 있는 방법 중 하나인 리팩토링을 사용하는데 이는 객체지향 소프트웨어의 설계를 변경하되 외부 행동은 변경시키지 않으면서 유지보수성을 향상시키고 유지보수 비용과 절감하고 시장으로의 출시 시간 (time-to-market)을 단축시킬 수 있는 이점이 있다.

본 연구에서는 객체지향 소프트웨어 체계적인 리팩토링 식별 방법 (리팩토링 후보군 식별과 리팩토링 선택 방법)을 제안한다. 리팩토링 후보 군을 식별하기 위하여 우리는 톱 다운 (top-down)과 보텀 업 (bottom-up) 방식을 제안한다. 첫 번째로, 톱 다운 방식은 전통적인 방식으로써, 객체지향 소프트웨어의 설계 원리들을 위반하는 요소들을 제거하기 하기 위한 휴리스틱 (heuristic)한 규칙들을 기반으로 리팩토링 후보 군을 식별한다. 본 연구에서는 리팩토링 후보 군들을 메소드와 클래스와 같은 엔티티 (entitiy)들 간의 의존성을 줄이는 방향으로 규칙들을 고안하였다. 또한 규칙을 만들 때, 시스템이 어떻게 사용되는지에 대한 동적인 정보가 앞으로 소프트웨어에 일어날 변경을 예측하기 위한 중요한 요소라는 연구를 기반으로 개발하였다. 이를 위하여, 유지보수성을 효과적으로 향상시킬 수 있는 리팩토링을 수행하기 위하여, 사용자 시나리오 (user scenario)와 사용 프로파일 (operational profile)과 같이 사용자들이 소프트웨어를 사용하는 정보를 기반으로 엔티티들을 식별하고 이러한 엔티티들을 중심으로 리팩토링 후보 군을 식별한다. 두 번째로, 보텀 업 방식은 리팩토링 후보 군들을 미리 정해진 패턴이나 규칙에 기반하지 않고 찾는 방법으로써, 그래프 이론에서의 최대 독립 집합 (maximal independent set) 개념을 기반으로 하여 메소드와 클래스와 같은 엔티티 (entitiy)들을 그룹핑하는 방법을 고안하였다. 엔티티들을 그룹핑 할 때 하나의 리팩토링이 다른 리팩토링의 수행과 충돌이 나는 현상인 기존의 리팩토링 의존성뿐만 아니라 본 연구에서 새롭게 발견된 유지보수성에 대한 리팩토링 효과 의존성 (refactoring effect dependency) 을 고려한다. 각각의 최대 독립 집합에 있는 엔티티들은 기본레벨 (elementary) 리팩토링들의 그룹으로 매핑되며 이들 리팩토링들은 한꺼번에 수행 가능하다. 마지막으로, 식별된 리팩토링 후보군들 중에서 실제로 적용할 리팩토링들을 선택한다. 이를 위하여, 기본레벨 리팩토링들의 평가(assessment) 및 영향 분석(impact analysis)를 지원함으로써 다수의 (multiple) 리팩토링을 선택할 수 방법을 리팩토링 효과 평가 프레임워크 (refactoring effect evaluation framework) 형태로 고안하였다. 리팩토링 효과 평가 프레임워크는 각각의 기본레벨 리팩토링의 효과를 매트릭스 계산을 이용하여 평가하고, 이를 이용하여 다수의 기본레벨 리팩토링들을 포함한 그룹 중 가장 많이 유지보수성을 개선할 수 있는 그룹을 선택한다.

제안한 방식은 jEdit, Columba, JGIT과 같은 세 개의 오픈 소스 프로젝트에 대하여 평가하였다. 그리고 실험 결과로부터 동적 정보가 효과적으로 유지보수성을 개선할 수 있는 리팩토링을 식별하는데 도움이

된다는 사실을 보였다. 이는 동적 정보가 자주 변경되는 클래스에서 리팩토링 후보 군들을 식별하기 때문이었다. 또한 다수의 리팩토링 선택 방식은 더 빨리 피트니스 함수 (fitness function) 값을 올리고, 더 적은 탐색 공간과 시간 측면에서 비용을 요구함을 보였다. 또한, 리팩토링 효과 의존성은 가장 많이 유지보수성을 개선하는 그룹을 적절하게 선택하기 위하여 필수적으로 고려되어야 한다는 사실도 실험 결과를 통하여 보였다.

감 사 의 글

2005년 3월 카이스트 SE LAB 배두환 교수님 연구실에 석사로 입학하여 박사 졸업하기 까지 많은 시간이 흘렀습니다. 저에게도 박사학위 감사의 글 쓰는 시간이 올까라고 생각하며 힘들어 하던 시기도 있었지만 많은 분들의 도움으로 무사히 박사학위 과정을 마칠 수 있었습니다. 이 글을 빌려 저를 도와주시고 지지해 주신 모든 분들께 감사의 말씀을 전하고자 합니다.

먼저 박사 과정 동안 많은 격려와 사랑을 베풀어주신 저의 지도 교수님이신 배두환 교수님께 큰 감사의 마음을 전합니다. 교수님께서 저의 지도교수님이신 것이 너무 자랑스럽습니다. 교수님께서 부족한 저에게 베풀어 주신 사랑 다른 사람들에게도 돌려주며, 앞으로 교수님 제자로서 멋지게 살아가도록 하겠습니다.

바쁘신 와중에도 소중한 시간을 내어 논문 심사를 해주시고 세심한 제안과 따뜻한 격려를 해 주신 백종문 교수님, 맹승렬 교수님, 허재혁 교수님, 홍장의 교수님께도 깊은 감사를 드립니다. 또한 연구자로서의 열정과 모범을 보여주시고 마음으로 응원을 보내주시는 차성덕 교수님, 권용래 교수님, 이윤준 교수님께도 깊은 감사의 말씀 전합니다.

박사과정 동안 옆에서 함께 해주는 연구실 선 후배와 동기들이 저에겐 큰 힘이 되었습니다. 먼저 힘든 일이 있을 때마다 고민을 들어주시고 진심으로 따뜻하게 격려해주신 지은경 박사님 감사드립니다. 일과 가정의 균형을 잃지 않으시고 매사에 긍정적이신 박사님의 모습은 저에게 많은 귀감이 되었습니다. 또한 랩 생활 동안 정말 많은 힘이 되었던 두 여학생, 희진이와 현정이 에게도 고맙다는 인사 전합니다. 밥 먹고 난 후 그대들과 마시는 커피 한잔은 저에게는 연구실 생활의 활력소가 되었습니다. 20대 꿈 많던 아가씨 때 만나서 결혼하기 까지 모두 지켜본 사이인 만큼 공유하는 추억이 참 많네요. 8년 동안 함께 있어줘서 고맙고 앞으로도 더욱 더 깊은 우정으로 평생 함께 할 수 있으면 좋겠습니다.

또한 랩에서 조용히 맡은 일을 묵묵히 해내며 연구실의 기둥이 되어준 영석이, 연구실에 활력을 불어넣어주고 든든한 버팀목이 되어준 박사과정들, 기곤이, 지민이, 지훈이, 동환이 모두 감사합니다. 앞으로도 지금처럼 좋은 연구하시고 건승하시길 바라겠습니다. 또한 산업체에서 오셔서 공부하시느라 수고가 많은 김영택 선배님, 한석씨, 종세 오빠, 미선 언니에게도 감사합니다. 덕분에 좋은 이야기도 많이 듣고 간접적으로 회사에 대하여 경험할 수 있었습니다. 학위과정 성공적으로 마무리 하시고, 학교에 계신 동안 좋은 추억 많이 만드셨으면 좋겠습니다. 또한 수업과 과제들, 그리고 랩의 궂은 일을 도맡아 하느라 수고가 많은 광의, (서)동원, (임)유진에게도 감사의 인사 전합니다. 특히 디펜스 다과 준비 정말 감동이었습니다. 앞으로도 특유의 유머와 긍정의 힘으로 어떤일이든 잘 해나가리라 믿습니다. 그리고 베트남에서 왔지만 한국어도 잘 하고 마음 착한 Nguyen Minh Chau에게도 감사합니다. 박사 고년 차인 저에게 어려워 하지 않고 다가와 주고 이야기도 많이 들려주어 고마웠습니다. 마지막으로, 함께 지내면서 정이 많이 든 백교수님 학생들 박진희, 류덕산, 김다정, 김태현, 이광규, 이낙원, 채민성 에게도 감사의 말씀 전합니다. 연구 열심히 하셔서 앞으로 원하시는 바 모두 이루길 기원합니다.

지금은 계시지 않지만 그동안 랩에서 함께 생활했던 선 후배님들께도 감사의 인사 전합니다. 연구

하는데 아낌 없이 조언해 주시고 가족처럼 챙겨주었던 경아 언니, 상욱 오빠, (강)동원 오빠, 승훈 오빠, 경식 오빠, 형인, 선경, 찬희, 슬기, 현식, 수현 오빠, 성태 오빠, 태훈 오빠, 숙희 언니, 석중 오빠, 진웅, 범석 오빠, Marew, Khanh, Giang, 승재, 인규 선배, 신호승 선배, 형호 오빠, (송)지환 오빠, 진호 오빠, 마유승 박사님의 따뜻한 마음들 잊지 못할 것입니다. 어디에 있으시던 건승하시고 행복 하시기를 바랍니다.

학위 과정 중에 만난 보물 같은 친구들에게도 감사를 전합니다. 특히 함께 박사 과정에 진학하여 동고동락한 저의 대녀 효실 이와 저의 아들의 대부인 혁윤군과 함께 보낸 시간들은 잊지 못할 것입니다. 둘의 결혼을 다시 한번 축하하며 행복한 가정 꾸리시고, 또한 일 에서도 앞으로 원하는 바 꼭 이루길 바라겠습니다. 또한 석사 동기들 영남, 은정, 영현 오빠, 형호, 태인, 지연 언니, 경수 오빠, 준일 오빠에게 감사합니다. 앞으로도 모두 열심히 노력하여서 자신의 목표를 실현하고 좋은 성과 거두기를 바랍니다.

또한 멀리 떨어져 있어서 연락도 소홀하고 잘 챙겨주지는 못함에도 불구하고 진심으로 아껴주는 친구들에게도 고맙고 미안한 마음 전합니다. 특히, 두 아들의 슈퍼 맘이자 나의 죽마고우 형이, 레지던트 말년 차라 고생하는 우리 현숙이, 유쾌 통쾌 상쾌 혜진이, 미국에서 박사 과정 중인 (이)주희, 똑 소리 나는 엄마이자 아내인 서강대99 퀸카 였던 미모의 경민 언니, 그리고 가정과 회사에서의 일 모두 훌륭하게 해내고 있는 멋진 친구들 주연, 윤경, 선미, 선미, (강)유진, 보민, 준희 언니, (하)주희 언니에게 진심으로 감사하다는 마음 전합니다. 또한 남편의 회사 위클레이 (Weclay Inc.) 식구들인 윤 대표, 우석 씨, 동주 씨, 주형 씨, 대영 씨, 성진 씨 에게도 감사의 마음 전합니다. 서로 아껴주면서 힘내서 열심히 일 하는 모습이 너무 멋지고, 분명히 좋은 결실을 맺으리라는 걸 믿어 의심치 않습니다. 항상 뒤에서 응원하겠습니다.

저를 위하여 기도해주시고 따뜻한 격려를 아끼지 않은 친척 분들께도 감사의 인사 드립니다.사랑하는 사촌들, 특히 브라질에 있는 지은 언니, 상도 오빠, 상훈 오빠, 슬기, 가은이, 보원 언니, 창배 오빠, 솔언니, 효정이, 정훈이, 캐나다에 있는 오늘 결혼한 Sarah, Sungmin 언니, David 오빠, 소라 언니, 정진이, 멕시코에 있는 (이)현정 언니, Hyun Hwa 언니에게 감사 전합니다.

마지막으로, 저의 사랑하는 가족에게 진심으로 고맙다는 말씀 전합니다. 먼저 원준이 태어나서 산후조리를 시작으로 세 돌이 되어가는 시점까지 지극 정성으로 원준이를 건강하고 예쁘게 키워 주신 친정 부모님 정말 감사합니다. 부모님 덕분에 무사히 졸업을 하였습니다. 부모님 고생이 헛되지 않게 저희 몫을 잘 해나면서 살겠습니다. 그리고 언제나 저와 남편, 원준이를 위하여 기도해 주시고 배려해주시는 시 부모님 진심으로 감사합니다. 정말 딸 같이 예뻐해 주시고 넘치는 사랑 주셔서 매일 감사하면서 살고 있습니다. 앞으로 더욱 잘 사는 모습 보여드리겠습니다. 그리고 건우 오빠, UBC에서 힘든 공부 중인 동생 정섭이 에게도 감사합니다. 열심히 한 만큼 좋은 결실을 맺을 수 있도록 기도하겠습니다. 그리고 제가 세상에서 제일 사랑하는 두 남자, 남편 송인권과 아들 송원준에게 감사의 말씀을 전합니다. 남편, 당신은 저의 최고의 연구 동료이자 최고의 친구입니다. 늘 곁에서 큰 힘이 되어 주어 정말 고맙고, 하는 일들이 고되고 힘들지만 내색 한번 안하고 즐기면서 하는 모습 보여주어 감사합니다. 지금 하는 일 들이 잘 되길 바랍니다. 건강하세요.

마지막으로 지면으로 통해서 일일이 언급을 하지 못했지만 그 동안 저를 아끼고 사랑해주신 모든 분들께 다시 한번 진심으로 감사 드립니다.

이 력 서

이 름 :	한 아 림
생 년 월 일 :	1981년 9월 30일
출 생 지 :	경기도 부천시
본 적 지 :	서울시 성북구 하월곡동
주 소 :	경기도 성남시 분당구 구미동 75 우성빌라 109동 301호
E-mail 주 소 :	arhan@se.kaist.ac.kr

학 력

1997. 3. – 2000. 2.	서울대학교 사범대학 부속고등학교
2000. 3. – 2004. 2.	서강대학교 컴퓨터학과 (B.S.) (Graduation Rank: No. 3 among all Computer Science students)
2005. 3. – 2007. 2.	한국과학기술원 전산학과 (M.S.) (advisor: Doo-Hwan Bae) Thesis: *Behavioral Dependency Measurement in UML 2.0 Sequence Diagrams for Change-proneness Prediction*
2007. 3. – 2013. 8.	한국과학기술원 전산학과 (Ph.D.) (advisor: Doo-Hwan Bae) Thesis: *Identification and Selection of Refactorings for Improving Maintainability of Object-Oriented Software*

경 력

2004. 2. – 2004. 4.	Intern, ZIO Interactive, Seoul, Korea
2004. 8. – 2004. 10.	Intern, Peace Corps (Headquarters), Washington, D.C., USA
2007. 3. – 2007. 12.	Chair of the Ph.D students of Computer Science Department, KAIST, Daejeon, Korea
2011. 3. – 2012. 8.	SAMSUNG Scholarship Program, SAMSUNG Electronics (supported by Video Display Division), Korea

연 구 업 적

1. **Ah-Rim Han**, Sang-Uk Jeon, Doo-Hwan Bae, Jang-Eui Hong, *Measuring behavioral dependency for improving change-proneness prediction in UML-based design models, Journal of Systems and Software* (JSS), Vol. 83, No. 2, pp. 222-234, Feb. 2010. (http://dx.doi.org/10.1016/j.jss.2009.09.038)

2. In-Gwon Song, Sang-Uk Jeon, **Ah-Rim Han**, Doo-Hwan Bae, *An approach to identifying causes of implied scenarios using unenforceable orders, Information and Software Technology* (IST), Vol. 53, No. 6, pp. 666-681, Jun. 2011. (http://dx.doi.org/10.1016/j.infsof.2010.11.007)

3. **Ah-Rim Han**, Doo-Hwan Bae, *Dynamic profiling-based approach to identifying cost-effective refactorings, Information and Software Technology* (IST), Vol. 55, No. 6, pp. 966-985, Jun. 2013. (http://dx.doi.org/10.1016/j.infsof.2012.12.002)

학 회 활 동

1. **Ah-Rim Han**, Sang-Uk Jeon, Doo-Hwan Bae, Jang-Eui Hong, *Behavioral Dependency Measurement for Change-proneness Prediction in UML 2.0 Design Models,* Proceedings of the 32nd Annual IEEE International Computer Software and Applications, pp. 76-83, Jul. 2008. (19.5% acceptance rate) **(Invited to the Special Issue of Journal of Systems and Software (JSS))**

2. **Ah-Rim Han**, Doo-Hwan Bae, *Automatic selection of multiple refactorings by considering refactoring effect dependency,* 28th IEEE/ACM International Conference on Automated Software Engineering (ASE) 2013, submitted.

3. **Ah-Rim Han**, Sang-Uk Jeon, Jang-Eui Hong, Doo-Hwan Bae, *Timing Consistency Checking in UML 2.0 Behavioral Models using OCL,* Proceedings of the Korea Computer Congress (KCC), Vol. 33, No. 1, pp. 181-183, Jun. 2006. (in Korean)

4. **Ah-Rim Han**, Dong-Won Kang, Hyeon-Jeong Kim, Doo-Hwan Bae, *An Approach to Extract Similar Process for Knowledge-Based Software Process Tailoring,* Proceedings of Workshop on Korea Software Engineering Technology, Vol.5, No. 1, pp. 42-52, Aug. 2007. (in Korean)

5. Hyung-In Ihm, In-Gwon Song, Sang-Uk Jeon, **Ah-Rim Han**, Jang-Eui Hong, Doo-Hwan Bae, *A Technique of Power Consumption Estimation for Embedded Software Design Models,* Proceedings of 2008 Korea Conference on Software Engineering (KCSE), Vol. 10, No. 1, pp. 113-120, Feb. 2008. (in Korean)

6. Hyung-In Ihm, **Ah-Rim Han**, Sang-Uk Jeon, Doo-Hwan Bae, Jang-Eui Hong, *Instruction Pattern-Based Power Consumption Estimation for Embedded Software Design Models,* Proceedings of 2009 Korea Conference on Software Engineering (KCSE), Vol. 11, No. 1, pp. 122-129, Feb. 2009. (in Korean)