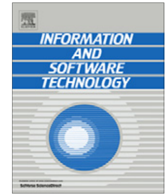
Information and Software Technology 55 (2013) 966–985

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Dynamic profiling-based approach to identifying cost-effective refactorings

Ah-Rim Han *, Doo-Hwan Bae

Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, South Korea

ARTICLE INFO

Article history:
Received 18 April 2012
Received in revised form 6 December 2012
Accepted 7 December 2012
Available online 20 December 2012

Keywords:
Cost-effective refactoring
Refactoring identification
Dynamic profiling
Dynamic method call
Maintainability improvement
Refactoring cost

ABSTRACT

Context: Object-oriented software undergoes continuous changes—changes often made without consideration of the software's overall structure and design rationale. Hence, over time, the design quality of the software degrades causing software aging or software decay. Refactoring offers a means of restructuring software design to improve maintainability. In practice, efforts to invest in refactoring are restricted; therefore, the problem calls for a method for identifying cost-effective refactorings that efficiently improve maintainability. Cost-effectiveness of applied refactorings can be explained as maintainability improvement over invested refactoring effort (cost). For the system, the more cost-effective refactorings are applied, the greater maintainability would be improved. There have been several studies of supporting the arguments that changes are more prone to occur in the pieces of codes more frequently utilized by users; hence, applying refactorings in these parts would fast improve maintainability of software. For this reason, dynamic information is needed for identifying the entities involved in given scenarios/functions of a system, and within these entities, refactoring candidates need to be extracted.
Objective: This paper provides an automated approach to identifying cost-effective refactorings using dynamic information in object-oriented software.
Method: To perform cost-effective refactoring, refactoring candidates are extracted in a way that reduces dependencies; these are referred to as the dynamic information. The dynamic profiling technique is used to obtain the dependencies of entities based on dynamic method calls. Based on those dynamic dependencies, refactoring-candidate extraction rules are defined, and a maintainability evaluation function is established. Then, refactoring candidates are extracted and assessed using the defined rules and the evaluation function, respectively. The best refactoring (i.e., that which most improves maintainability) is selected from among refactoring candidates, then refactoring candidate extraction and assessment are re-performed to select the next refactoring, and the refactoring identification process is iterated until no more refactoring candidates for improving maintainability are found.
Results: We evaluate our proposed approach in three open-source projects. The first results show that 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. The second results show that dynamic information is helpful in extracting refactoring candidates in the classes where real changes had occurred; in addition, the results also offer the promising support for the contention that using dynamic information helps to extracting refactoring candidates from highly-ranked frequently changed classes.
Conclusion: Our proposed approach helps to identify cost-effective refactorings and supports an automated refactoring identification process.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Object-oriented software undergoes continuous changes with various maintenance activities such as the addition of new functionalities, correction of bugs, and adaptation to new environments. 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* [1] or *software decay* [2]. Thus, refactoring can serve to restructure the design of object-oriented software without altering its external behavior [2] to improve maintainability, which in turn reduces maintenance costs and shortens time-to-market.

Given time, resource, or budget limitations for doing refactoring, a method for identifying cost-effective refactorings is needed. Cost-effectiveness of applied refactorings can be explained as maintainability improvement over invested refactoring effort

* Corresponding author. Tel.: +82 42 350 7739; fax: +82 42 350 8488.
E-mail addresses: arhan@se.kaist.ac.kr (A.-R. Han), bae@se.kaist.ac.kr (D.-H. Bae).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.12.002

(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. In our approach, we aim for making software accommodate changes more easily by performing cost-effective refactorings.

Previous studies have shown that the data capturing how the system is utilized is an important factor for estimating changes. Robbes et al. [3] 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 studies [4,5] 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, which aimed to perform cost-effective refactoring, 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. For this reason, to identify the cost-effective refactorings, the entities are identified based on the dynamic information of how the users utilize the software (e.g., user scenario and operational profile [6]); and within these entities, refactoring candidates need to be extracted. 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.

In this paper, we provide an automated approach to identifying cost-effective refactorings using dynamic information in object-oriented software. 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. We use the dynamic profiling technique [7] to obtain the dependencies of entities based on dynamic method calls by executing programs based on user scenarios or operational profiles. We define those dynamic dependencies as DMCs, and in this paper, dynamic information indicates the dynamic dependencies of entities (i.e., DMCs) that are obtained by using the dynamic profiling. Based on the DMCs, (1) the rules for reducing dependencies of entities are defined (for refactoring candidate extraction) and (2) the maintainability evaluation function is established (for refactoring candidate assessment).

The brief procedure of the proposed systematic approach for cost-effective refactoring identification is as follows. Refactoring candidates are extracted using the refactoring candidate-extraction rules. Then, refactoring candidates are assessed using the maintainability evaluation function. Those refactoring candidates are sorted in the order of their expected degree of improvement on maintainability, and the best refactoring—that most improves maintainability—is selected in a greedy way and applied. Refactoring-candidate extraction and assessment are re-performed to select the next refactoring, and the refactoring identification process is iterated until no more refactoring candidates for improving maintainability are found. Finally, a sequence of refactorings is generated by logging the results of refactoring selection.

The dynamic-profiling-based approach—refactoring identification using dynamic information—has been evaluated on three open-source projects: jEdit [8], Columba [9], and JGIT [10]. Two tests are performed. The first test investigates the overall usefulness of the approach for the refactoring identification, while the second test investigates the capability of the approach in extracting refactoring candidates. For the first test, we evaluate whether dynamic information is helpful in identifying cost-effective refactorings that fast improve maintainability. The method of change simulation is used for assessing the capability of refactorings for maintainability improvement using the reduced number of propagated changes in regard to injected original changes. 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, (2) the approach using static information only, and (3) the combination of the two approaches. We use two indicators (the percentage of reduction for propagated changes and the rate of reduction for propagated changes) to show cost-effectiveness of the identified refactorings. The results show that 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. For the second test, we evaluate whether dynamic information is helpful in extracting refactoring candidates in the classes where real changes had frequently occurred. We compare (a) the classes of top *k*% most frequently changed during the development history 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. The results show that dynamic information is helpful in extracting refactoring candidates in the classes where real changes had occurred. In addition, 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 can be a great help for cost-effective refactoring identification.

The rest of the paper is organized as follows. Section 2 contains a discussion of related studies. Section 3 explains the basic information and the overview of our proposed approach for cost-effective refactoring identification. Section 4 explains the detailed procedure and methods of refactoring candidate extraction, assessment, and selection. Section 5 covers the implemented tool for applying our proposed approach. In Section 6, we present the experiment performed to evaluate the proposed approach and discuss the obtained results. Finally, we conclude and discuss future research in Section 7.

2. Related work

Much of the existing research on automated refactoring focuses on *refactoring application* [11–15], 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 [16–18], removing code clones [19–21], and improving code quality such as testability [22], as well as maintainability. (2) For evaluating the design of the refactored code, design quality

evaluation models such as QMOOD [19] and Maintainability Index (MI) [23], or a special metric such as historical volatility [24], are used. Distance measures [25,26] or weighted sums of metrics [27] 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 [28]. (3) The methods for scheduling of refactoring candidates also have been studied [29,30] to achieve the greatest effect of maintainability improvement. However, we still lack systematic approaches, clear guides, and automated tool support for identifying where to apply which refactorings and in what order. In addition, research has yet to consider methods for identifying cost-effective refactorings in terms of using dynamic information. In the following, studies are presented in the categories of metric-based refactoring and search-based refactoring.

2.1. Metric-based refactoring

Several studies have attempted to support for identifying refactorings using metrics as a means of detecting refactoring candidates or evaluating refactoring effects [31,25,32–34]. Tahvildari and Kontogiannis [33] 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. Du Bois et al. [31] 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 and Kontogiannis [33], 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. [25] 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, 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 and Chatzigeorgiou [26] 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 [25]. 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. Search-based refactoring

The literature has proposed methods of refactoring identification by using several search techniques. O'Keeffe and Cinnéide [35] 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. However, they do not say where to apply which refactorings because extraction of refactoring candidates depends on random choices. Moreover, as mentioned in their paper, search-based refactoring techniques have difficulties in computing time and memory use.

Lee et al. [19] and Seng et al. [27] use genetic algorithms to produce a sequence of refactorings to apply to reach an optimal system in terms of the employed fitness function. However, Seng et al. [27] 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. Lee et al. [19] 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.

3. Overview of our approach

The following four subsections deal with the basic information of our approach. The first subsection presents a framework for the systematic refactoring identification to enable automated refactoring. In the second subsection, we explain the goal of refactoring and used refactoring in our approach. In the third subsection, the first subsection presents the procedure of dynamic profiling technique used in our approach, and the next subsection provides the definition and measurement of DMCs. Finally, the last subsection presents the overview of our proposed approach for cost-effective refactoring identification.

3.1. Framework for systematic refactoring identification

According to Mens and Tourwé [36], 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.

Table 1
Identified three phases by referencing refactoring process in [36].

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

6. Maintain the consistency between the refactored program code and other software artifacts such as documentation, design documents, and test cases.

Based on this process, we categorize the activities of the refactoring process into three phases (Table 1). ''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 this paper, we focus on refactoring identification and propose a framework for systematic refactoring identification (as in Fig. 1) to enable automated refactoring. In Section 4, we will explain the detailed methods of the proposed approach for cost-effective refactoring identification.

3.2. Goal of refactoring and used refactorings in our approach

Dependency refers to a relationship where the structure or behavior of an entity is dependent on another entity [37]. UML defines dependency as a relationship where a change to the influent modeling element may affect the dependent modeling element [38]. Object-oriented software involves structural and behavioral aspects of dependencies [39]. 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 [39]. 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 [2]), or a single class is modified by many different types of changes (e.g., Divergent Change [2]). 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 [2] 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 [26]. 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. Since the objective of our paper is to show the effect of using dynamic information for cost-effective refactoring identification, we focus on the two refactorings.

3.3. Dynamic profiling-based approach

3.3.1. 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

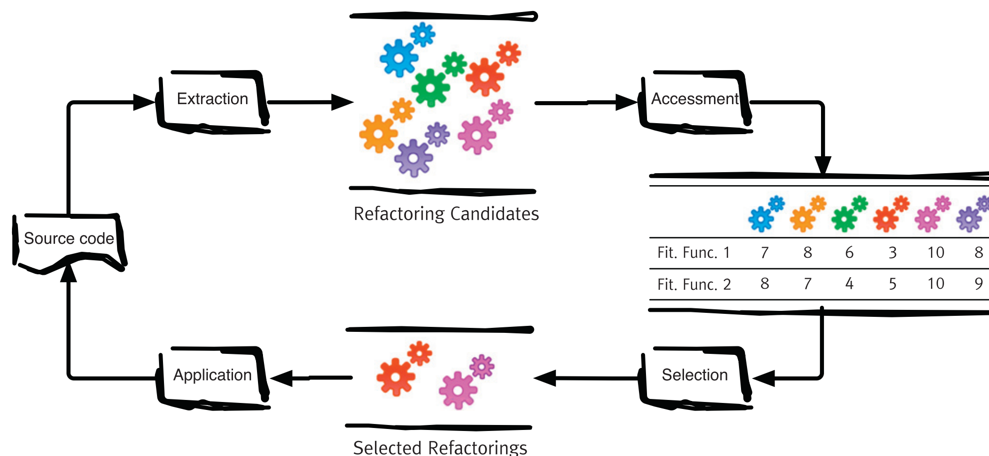


Fig. 1. A proposed framework for systematic refactoring identification.

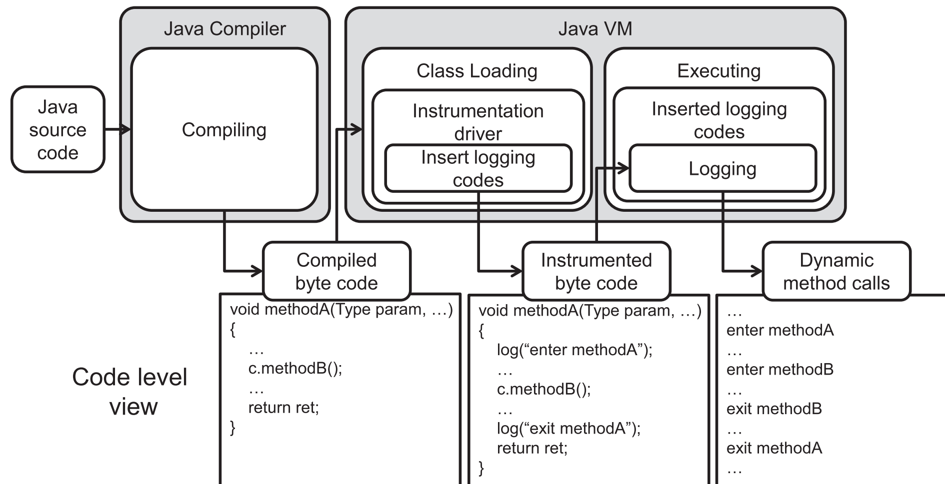


Fig. 2. Procedure for dynamic profiling.

Table 2
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

achieved by instrumenting either the program source code or its binary executable form. Dynamic profiling has been used by many researchers [40–42]. 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 [6] 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. 2 depicts the procedure of dynamic profiling used in this paper. The java instrumentation technique [7] 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.

3.3.2. 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 2. 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 $o_1 \vec{n}\, o_2$. 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 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.

3.4. Overview of our approach for cost-effective refactoring identification

Fig. 3 shows an overview of our proposed approach for cost-effective refactoring identification. The following briefly describes a procedure of the proposed cost-effective refactoring identification method. The input of refactoring identification is the source code. Using results obtained from static and dynamic profiling, an initial profiled model is constructed. Refactoring candidate extraction rules are defined and the maintainability evaluation function is established based on the information obtained from the profiled model (see Sections 4.1 and 4.2, respectively). Refactoring identification consists of three main activities: refactoring-candidate extraction, refactoring-candidate assessment, and refactoring selection. Refactoring candidates are extracted using the refactoring-candidate extraction rules. Then, refactoring candidates are applied to produce the tentatively refactored models and assessed by evaluating those refactored models using the

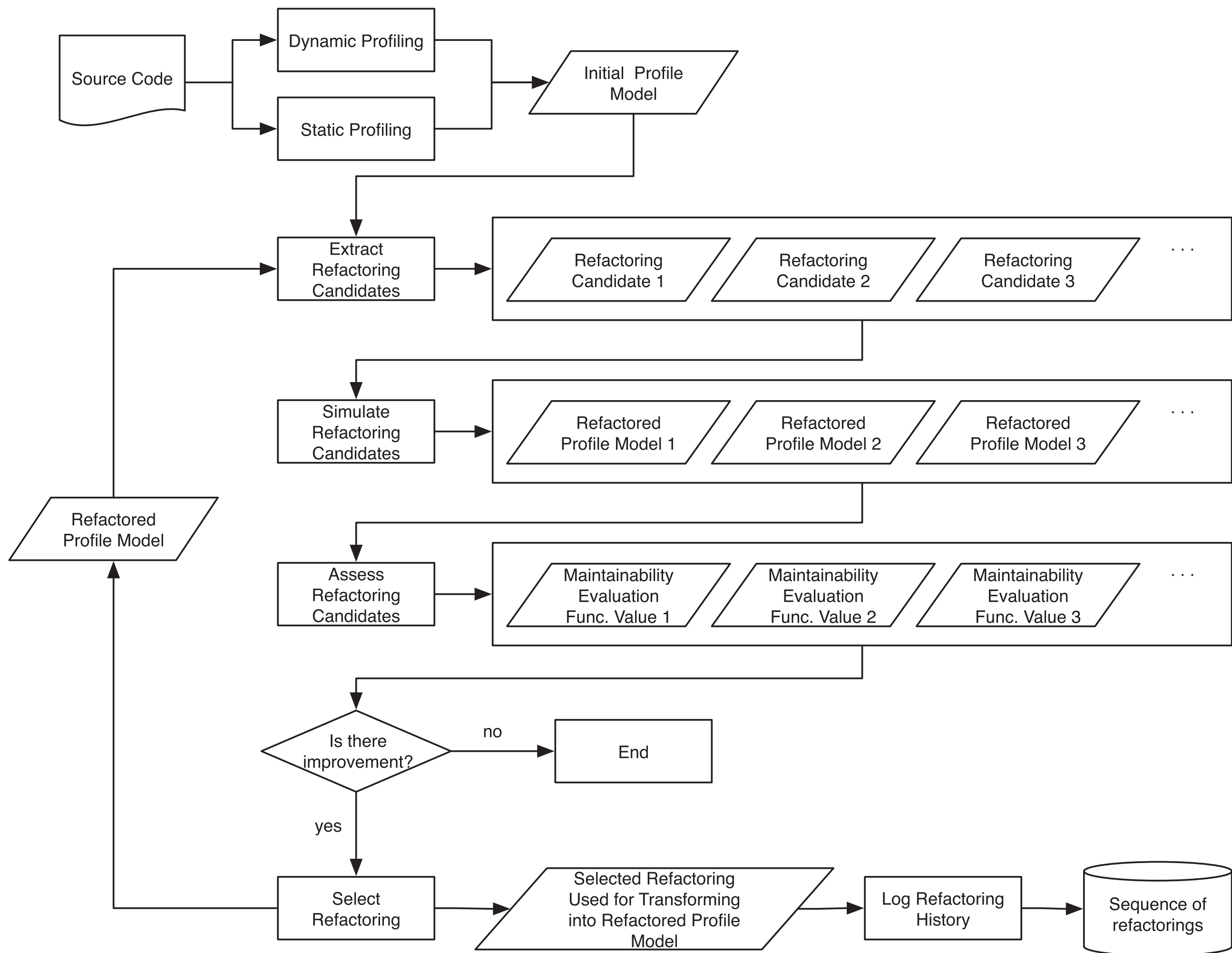


Fig. 3. Overview of our proposed approach for cost-effective refactoring identification.

maintainability evaluation function. Those refactoring candidates are sorted in the order of expected degree of improvement on maintainability. The best refactoring—that which most improves maintainability—is selected in a greedy way and applied; then the profiled model is updated. The procedure of refactoring identification is iterated until no more refactoring candidates for improving maintainability are found. The output of refactoring identification is a sequence of refactorings, which is comprised of the logged results of refactoring selection. The next section will explain the detailed procedure and methods of the three main activities of the refactoring identification.

4. Refactoring candidate extraction, assessment, and selection

4.1. Refactoring candidate extraction

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.

4.1.1. 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:

4.1.1.1. 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.

4.1.1.2. 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.

4.1.1.3. Refactoring. As stated in subSection 3.2, we use two types of refactorings—Move Method and Collapse Class Hierarchy—and the operations of those refactorings are presented in Procedures 1 and 2, respectively. We formulate pre-and post-conditions referring to [26,43,44] and check before and after refactoring applications. We do not specify these conditions in this paper.

Procedure 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})
  end for
  $C_{merged}.ancestor \leftarrow C_{merged}.ancestor \cup C_{merging}.ancestor$
  $C_{merged}.descendant \leftarrow C_{merged}.descendant \cup C_{merging}.descendant$
  $C_{merged}.field \leftarrow C_{merged}.field \cup C_{merging}.field$
  remove C_{merged}

Procedure 2. Move Method

Require: C: a target class to which a method is moved
Require: M: a method to be moved
  M.overridingMethod ← findingOverriding (C, M)
  /∗ findingOverriding *function is specified in* Procedure 3.∗/
  C.method ← *C.method* ∪ {M}

Procedure 3. findingOverriding

Require: c: a target class to which a method is moved
Require: m: a moving method
/∗ *procedure* findingOverriding *returns the method by which movingMethod is overrided.* ∗/
queue← ancestor classes of c
visitedClass ← ∅
while *queue* ≠ ∅ **do**
  tmpClass ← remove one element of class from *queue*
  add *tmpClass* to *visitedClass*
  for all *tmpMethod*∈ methods in c **do**
    if *tmpMethod* = m **then**
      return *tmpMethod*
    end if
  end for
  for all *ancClass* ∈ ancestor classes of *tmpClass* **do**
    if *visitedClass* does not contain *ancClass* **then**
      add *ancClass* to *queue*
    end if
  end for
end while
return *null*

4.1.2. 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, 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 paper, 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.

Procedure.
Procedure 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 *N*Diff_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 *N*Diff_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)
- $\mathbf{R}_{3(N=2)}, \mathbf{R}_{6(N=3)}, \mathbf{R}_{9(N=4)}, \mathbf{R}_{12(N=5)}, \mathbf{R}_{15(N=6)}: {}^{\forall}(m_i, m_j) \in max(N\text{Diff_M}, cutline) \rightarrow$ Move Method (owner class of m_j, m_i)

Procedure 4. get*N*Diff_M_and_*N*Diff_C (N = 2, 3, 4, 5, 6)

```
for all c ∈ classes in the system, m ∈ methods in c do
  diffClass ← ∅   |*a set for saving different callee classes*|
  for all dmc∈DMC (m,≺) do
    if dmc.c_callee≠c then
      add dmc.c_callee to diffClass
    end if
  end for
  if diffClass.size ⩾ N then
    for all dmc1,dmc2 ∈ DMC(m,≺) do
      if dmc1 ≠dmc2 then
        c1← dmc1.c_callee,c2←dmc2.c_callee
        if c1≠c2 & & c1≠c & & c2≠c then
          NDiff_C ({c1,c2})← NDiff_C ({c1,c2}) + 1
        end if
        m1←dmc1.m_callee,m2←dmc2.m_callee
        if m1≠m2 & & m1≠m & & m2≠m then
          NDiff_M ({m1,m2})← NDiff_M ({m1,m2}) + 1
        end if
      end if
    end for
  end if
end for
```

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.
Procedure 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.

- $\mathbf{R}_{16}$:$^{\forall}(c_i, c_j) \in max$(I_ C,*cutline*) → Collapse Class Hierarchy (c_i, c_j)
- $\mathbf{R}_{17}$:$^{\forall}(m_i, m_j) \in max$(I_M,*cutline*) → Move Method (owner class of m_i, m_j)
- $\mathbf{R}_{18}$:$^{\forall}(m_i, m_j) \in max$(I_M,*cutline*) → Move Method (owner class of m_j, m_i)

Procedure 5. getI_C_and_I_M

```
for all c ∈ classes in the system, m ∈ methods in c do
  if m ≠null then
    for all dmc1 ∈ DMC(m,≺) do
      c1 ← dmc1.c_caller,c2 ← dmc1.c_callee
      if c1 ≠c2 & & c1 ≠c & & c2 ≠c then
        I_C({c1,c2}) ← I_C({c1,c2}) + 1
      end if
      m1 ← dmc1.m_caller,m2 ← dmc1.m_callee
      if m1 ≠m2 & & m1 ≠m & & m2 ≠m then
        I_M({m1,m2}) ← I_M({m1,m2}) + 1
      end if
    end for
  end if
end for
```

4.2. Refactoring candidate assessment

Before making a decision on which refactorings to apply, we need to assess the extracted refactoring candidates. Each of the extracted refactoring candidates is assessed as follows. The refactoring candidate is applied to the current version of the profiled model, and the tentatively refactored model is produced. Then, the design quality of maintainability for the refactored model is evaluated. To evaluate maintainability of the refactored model, the maintainability evaluation function is established. By using the maintainability evaluation function, the extracted refactoring candidates are assessed and ranked in the order of their expected degree of improvement on maintainability.

4.2.1. Maintainability evaluation function

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 [45], because less propagation of changes to other parts of the system or side effects would occur [46,47]. *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. 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 [48,49], or 2) Pareto optimality [28], are used. We adopt the former approach for conflating two objectives of metrics 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. 4 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.

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

Fig. 4. Maintainability evaluation function for producing *fitness* value.

In the following, each metric—constituting the maintainability evaluation function—is explained. For cohesion, the Method Similarity Cohesion (*MSC*) [50] 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) [51] 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) [52], 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 eight types of combination: (dynamic method call vs. static method call) × (import direction vs. export direction) × (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) [53] counts static method calls for all invoked methods in the import direction, and Request For a Class (RFC) [51] counts static method calls for distinct methods in the import direction, while Coupling Between Objects (CBO) [51] counts static method calls for distinct methods in both directions. The coarse-grained metrics, such as Coupling Factor (CF) [54], 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.

4.3. Refactoring selection

The application of a refactoring may (1) change or delete elements necessary for other refactorings, and thus disable these refactorings and (2) create new available refactoring candidates to apply. As a result, the application order of refactorings determines the total effect on maintainability improvement. Therefore, when selecting refactorings to apply among extracted refactoring candidates, the applying order needs to be considered to achieve the greatest effect of maintainability improvement. Note the difficulty in selecting multiple refactorings, because refactorings need to be scheduled by considering issues such as 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 [19]. Furthermore, the cost of evaluating the effect of every refactoring candidate is large, since the refactoring needs to be actually applied as stated in Section 4.2. We are working to devise a technique for multiple selections and leave this for future work. Hence in our approach, among refactoring candidates, we select the best refactoring in a one-by-one—greedy manner—by re-performing the process of refactoring identification.

For selecting the best refactoring, refactoring candidates are prioritized in the descending order of fitness values of the maintainability evaluation function, and the refactoring with the largest fitness value is selected. It is important that before selecting a refactoring, for the refactored model, we check the *specialization ratio* (S) [55], which is a measure used to prevent merging of too many classes together and getting the class hierarchy wider. S is formulated as follows: (♯ of classes − ♯ of root classes)/(♯ of classes − ♯ 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.

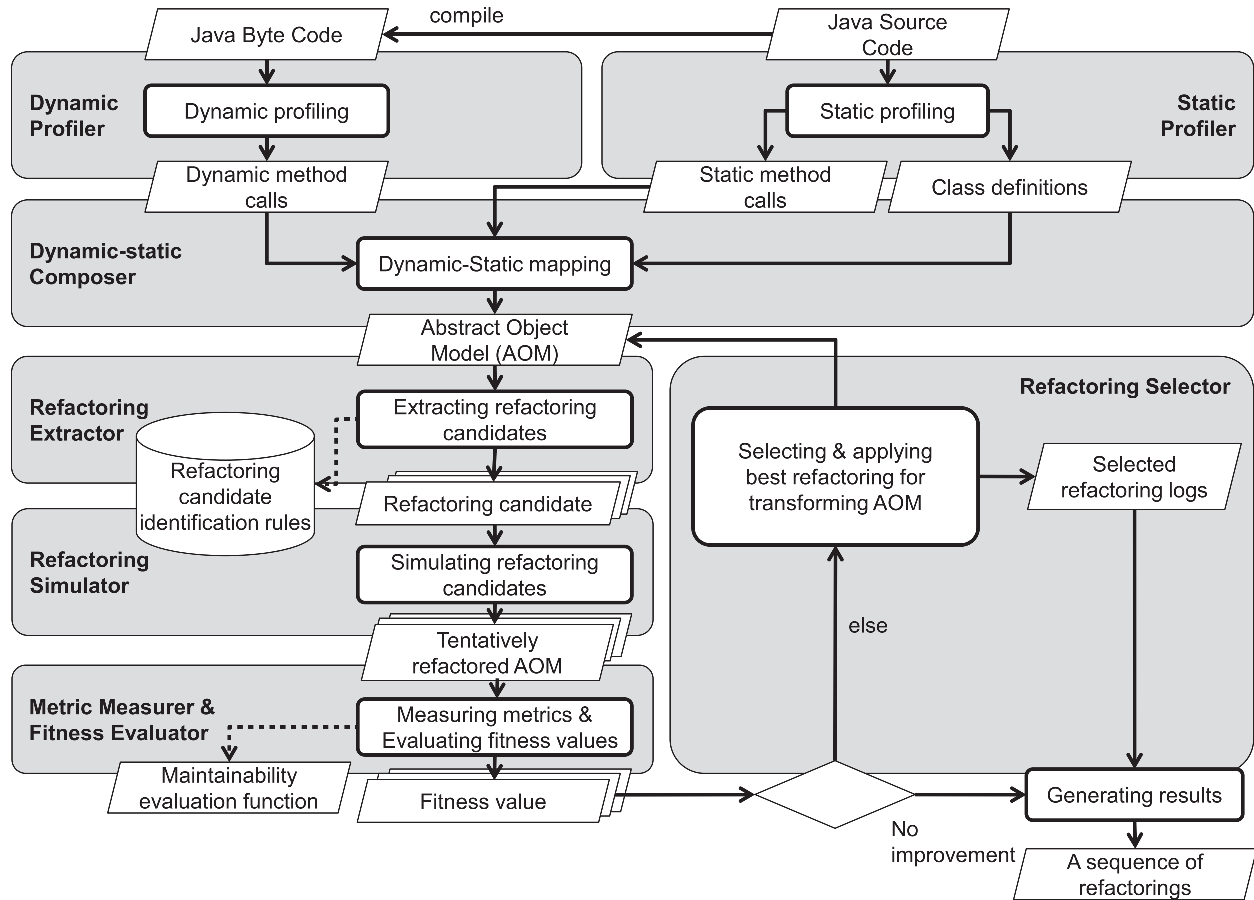


Fig. 5. Overall tool architecture.

After selecting the best refactoring, the profiled model is updated into the selected refactored model, and the selected refactoring is logged. Refactoring-candidate extraction and assessment are re-performed to select the next refactoring, and the refactoring identification process is iterated until there are no more improvements of fitness values for the extracted refactoring candidates. When no more refactoring candidates for improving maintainability are found, the refactoring identification procedure is stopped, and the sequence of logged refactorings is generated.

5. Tool implementation

5.1. Overall tool architecture

The proposed method has been implemented [56] using Java with Eclipse environment. Fig. 5 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 Section 5.2. 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 shows a snapshot of the tool operation.

5.2. Abstract Object Model (AOM)

AOM is the profiled model; it is specifically designed in our tool for saving the dynamically and statically profiled information and simulating the effect of refactorings without actually applying refactorings on source codes. Fig. 7 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

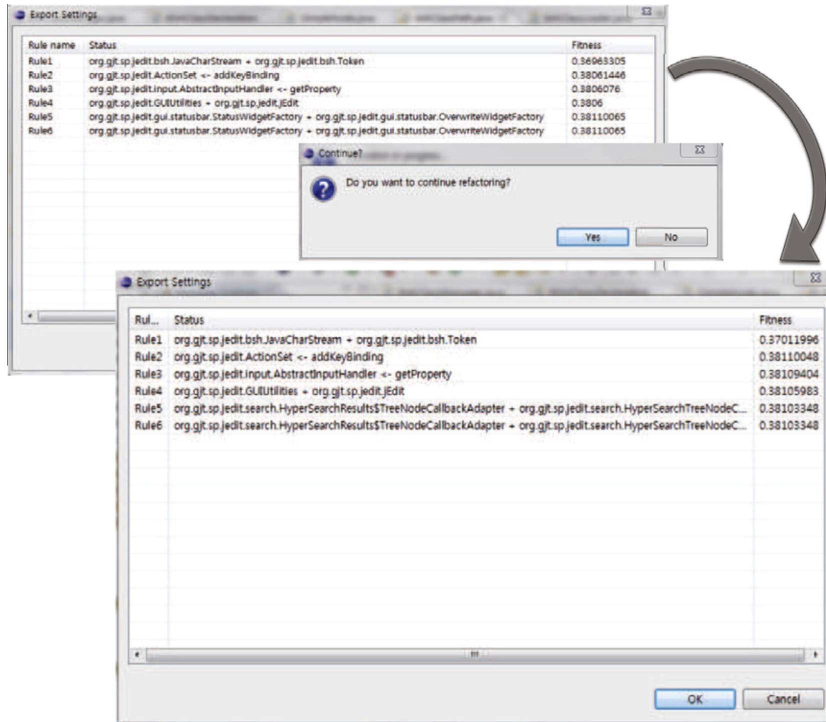


Fig. 6. A snapshot of tool operation.

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.

The AOM is beneficial in two ways. First, the cost of computing metrics and fitness value for every trial of refactoring is rather high. Since not all the information of the source codes is necessary, computing costs can be saved by manipulating abstract source code models. Second, 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. As mentioned in subSection 5.1, in the dynamic-static composer, the DMCs are *mapped* into the corresponding SMCs from which those DMCs are instantiated. Therefore, for each trial 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.

6. Evaluation

We evaluate our approach—refactoring identification using dynamic information—to check its usefulness for identifying cost-effective refactorings. The hypotheses for our experiment are as follows:

H1. Dynamic information is helpful in identifying cost-effective refactorings that fast improve maintainability (i.e., that lead to a high rate of maintainability improvement).

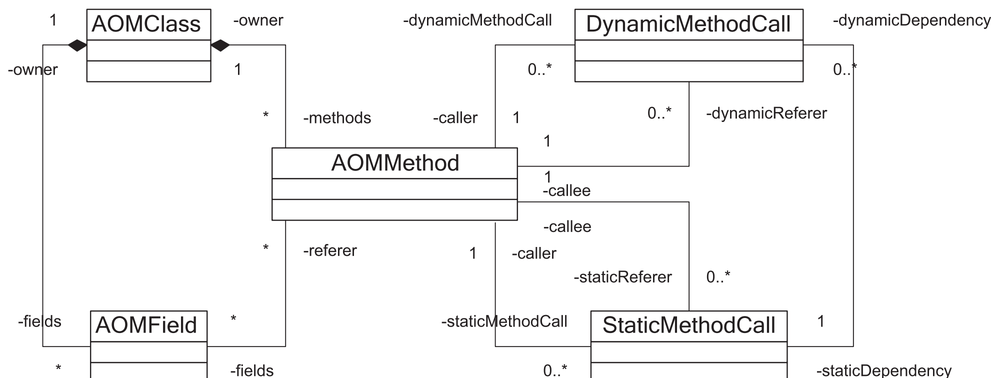


Fig. 7. A metamodel of the AOM used in this paper.

H2. Dynamic information is helpful in extracting refactoring candidates in the classes where real changes had frequently occurred.

The first hypothesis is intended to test the overall usefulness of the approach for the refactoring identification, while the second hypothesis is to test the capability of the approach in extracting refactoring candidates. In the first subsection, the experimental subjects and data processing method for those subjects are explained. In the second and third subsections, the evaluation methods are explained, and the results are presented for each hypothesis, respectively. The final subsections end with threats to validity and discussion.

6.1. Evaluation subjects and data processing

Three projects are chosen for experimental subjects: jEdit [8], Columba [9], and JGIT [10]. 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.
- They are written in Java; our proposed method applies to object-oriented software.
- Their development histories are well-managed in version-control systems.

Table 3 summarizes characteristics and development histories of each subject.

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 3). 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.

6.1.1. 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 [8] 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 [9] 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 [10] 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

Table 3
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 (♯ of classes in the version)	jEdit-4.3 (953 classes)	Columba-1.4 (1506 classes)	v1.1.0.201109151100-r (494 classes)

Table 4
Examined range of revisions.

jEdit	Columba	JGIT
18,000–19,000	300–450	1–1616

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.

6.1.2. Extracting changes

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 4) 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) → 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) → changed_line_number

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

6.2. Evaluation method

6.2.1. Hypothesis 1: Effect of dynamic information for cost-effective refactoring identification

To test Hypothesis 1, 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.

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 6.1.2) 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

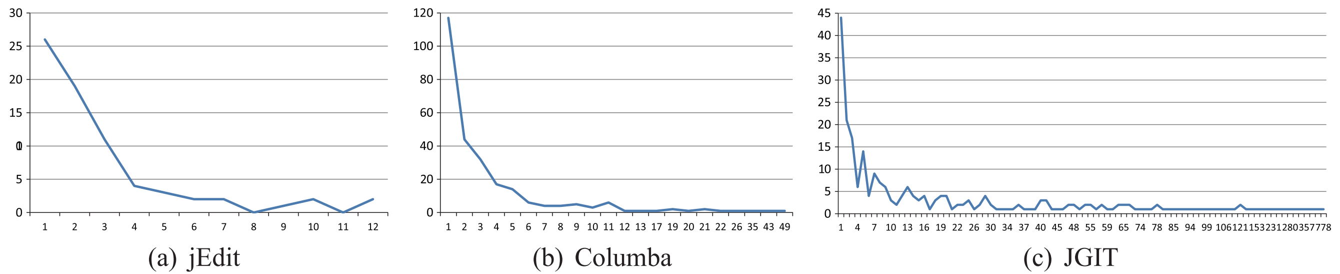


Fig. 8. Change distribution graph (*X*-axis: ♯ of occurred changes for each class, *Y*-axis: ♯ of corresponding classes).

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.

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. 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 nth 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 $r_1, \ldots, r_n$. 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.

6.2.2. Hypothesis 2: 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 Hypothesis 2, 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 6.1.2) 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. 8). 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 ≈ 70%, 788 out of 993 ≈ 80%, and 8754 out of 9773 ≈ 90%, for jEdit, Columba, and JGIT, respectively. Note that, as with Hypothesis 1, 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.

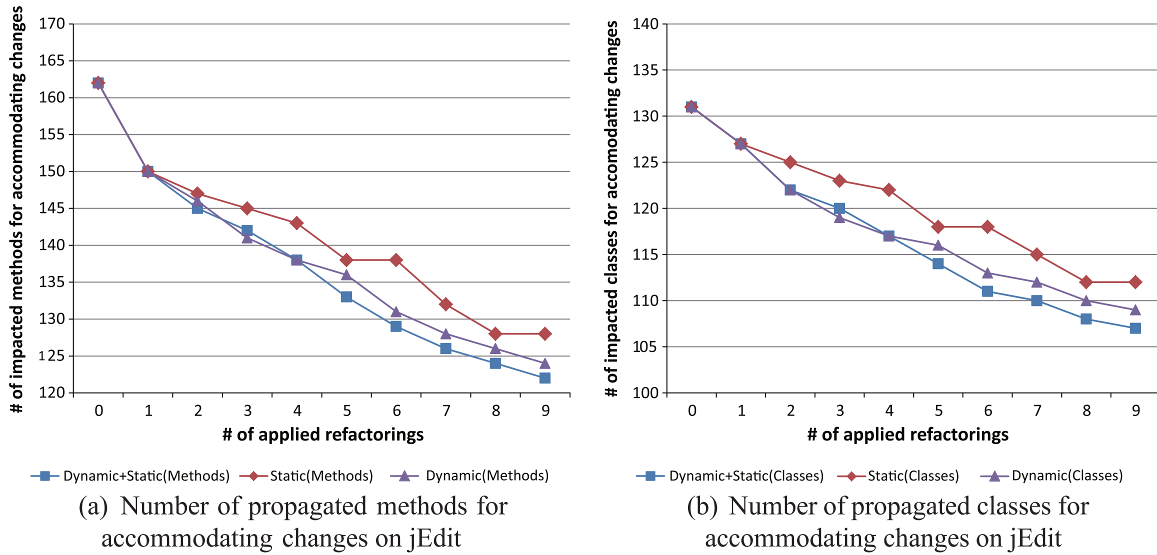
We use the classes of refactoring candidates that are obtained from group 1 (i.e., the approach using dynamic information only) and group 2 (i.e., the approach using static information only) in Hypothesis 1. The lists of the extracted classes as refactoring candidates for each approach and the list of 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 [57]), which are the measures for comparing similarity of two top *k* lists.

6.3. Results

6.3.1. Hypothesis 1: Effect of dynamic information for cost-effective refactoring identification

The results are represented in Figs. 9–11 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 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.

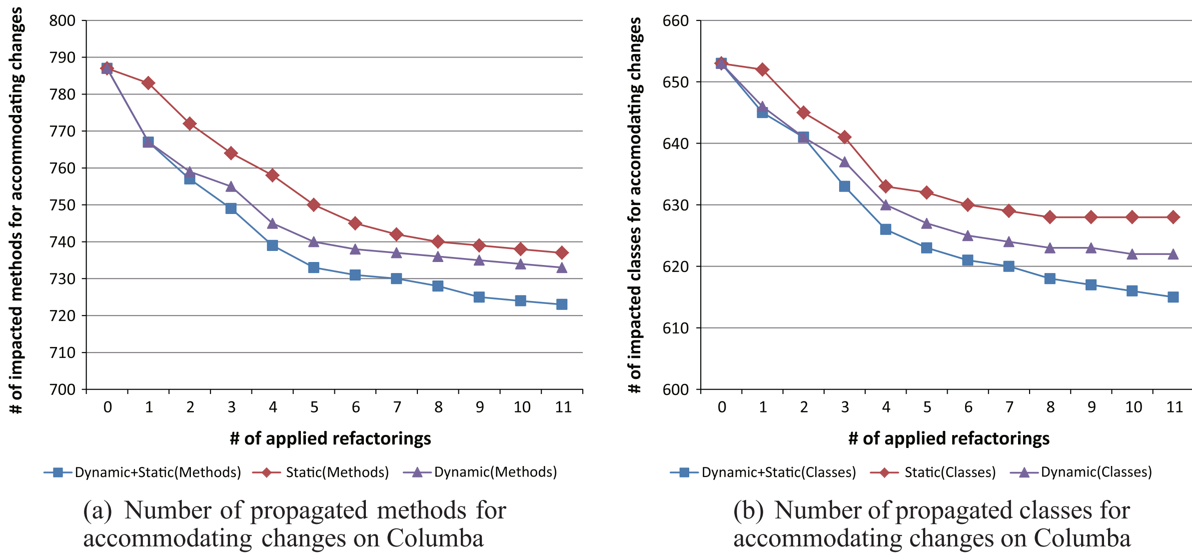
For jEdit, as in Fig. 9a and 9b, the same number of propagated changes is reduced for all approaches in the first applied refactoring. However, from the next applied refactorings, the approaches



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

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

Fig. 9. Change simulation for jEdit.



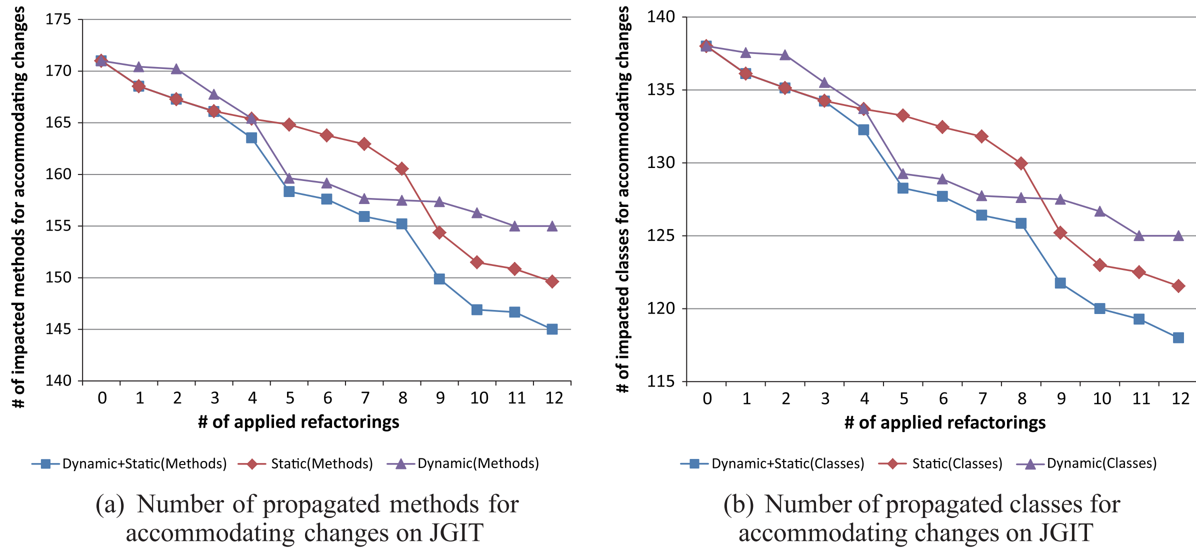
(a) Number of propagated methods for accommodating changes on Columba

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

Fig. 10. Change simulation for Columba.

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 72–75%—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. 10a and 10b, 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 75–76%—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,



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

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

Fig. 11. Change simulation for JGIT.

Table 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.

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
jEdit							
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							
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							
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

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 3), 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. Further-

Table 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.

Changed			Static ∩ Changed		Dynamic ∩ Changed	
Top %	Class ♯	Change ♯	Class ♯	Change ♯	Class ♯	Change ♯
jEdit						
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						
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						
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

Note: 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).

more, 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. 11a and 11b, 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 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.

6.3.2. Hypothesis 2: 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 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 [57]) of the approach using static information and the approach using dynamic information are represented in Table 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

Table 7
Top k ranking distance measures (K: Kendall's tau; F: Spearman's footrule [57]) between the real changed classes and the extracted classes as refactoring candidates for each approach using static information and approach using dynamic information.

Changed	jEdit				Columba				JGIT			
	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

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

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 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.

6.4. 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.

6.5. Discussion

Some of the researches addressed the method of estimating refactoring cost; for example, Zibran and Roy [21] 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 3.1). 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.

7. Conclusion and future work

In this paper, we provide the automated approach to identifying cost-effective refactorings using dynamic information. The dynamic profiling technique is used to obtain dynamic dependencies of entities (i.e., DMCs). Based on the DMCs, (1) the rules for reducing dependencies of entities are defined (for refactoring candidate extraction) and (2) the maintainability evaluation function is established (for refactoring candidate assessment). We have proposed the framework for systematic refactoring identification to enable automated refactoring. Three main activities are performed to identify cost-effective refactorings as follows: (1) extracting refactoring candidates in a way that reduces dependencies of entities of methods and classes using the refactoring candidate-extraction rules; (2) assessing those extracted refactoring candidates using the maintainability evaluation function; and, finally, (3) selecting the most cost-effective refactoring among the extracted refactoring candidates. The procedure of refactoring identification is iterated until no more refactoring candidates for improving maintainability are found, and it then generates the sequence of refactorings.

In the experiment, our approach has been evaluated on three open-source projects—jEdit, Columba, and JGIT. In the first test, we investigate whether dynamic information is helpful in identifying cost-effective refactorings that fast improve maintainability. The results show that 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. In the second test, we investigate whether dynamic information is helpful in extracting refactoring candidates in the classes where real changes had frequently occurred, and the results show that dynamic information is helpful in extracting refactoring candidates in the classes where real changes had occurred. In addition, 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 support the arguments that using dynamic information can be a great help for cost-effective refactoring identification. From all these results, we have come up with the conclusion that dynamic information plays an important role (i.e., becomes a very good factor) in identifying cost-effective refactorings, especially for the system having intensive user interactions (such as jEdit or Columba). For JGIT, which is algorithmic-based and has rather less interactions with users, additional predictors other than dynamic information may help to better identify cost-effective refactorings.

As for future work, we plan to consider other types of refactorings such as Extract Class. Furthermore, we plan to develop a method for selecting multiple refactorings. This is needed because it is inefficient to select just the best refactoring in a one-by-one of greedy manner for each iteration of refactoring identification, after producing the tentatively refactored models by applying all the extracted refactoring candidates and evaluating those refactored models. Therefore, we need to devise algorithms for (1) reducing the search space by identifying independent sets of refactorings and (2) searching the refactoring candidates effectively considering dependency of application among them.

Acknowledgement

This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

References

[1] D. Parnas, Software aging, in: Proceedings of The 16th International Conference on Software Engineering (ICSE94), IEEE Computer Society Press, 1994. pp. 279–287.
[2] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.
[3] R. Robbes, D. Pollet, M. Lanza, Replaying ide interactions to evaluate and improve change prediction approaches, in: 7th IEEE Working Conference on Mining Software Repositories (MSR), 2010, IEEE, pp. 161–170.
[4] E. Arisholm, L. Briand, A. Føyen, Dynamic coupling measurement for object-oriented software, IEEE Transactions on Software Engineering (2004) 491–506.
[5] A.-R. Han, S.-U. Jeon, D.-H. Bae, J.-E. Hong, Measuring behavioral dependency for improving change-proneness prediction in uml-based design models, The Journal of Systems & Software 83 (2010) 222–234.
[6] J. Musa, Operational profiles in software-reliability engineering,, IEEE Software 10 (1993) 14–32.
[7] M. Dmitriev, Selective profiling of java applications using dynamic bytecode instrumentation, in: 2004 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2004, pp. 141–150.
[8] jEdit, jEdit, 2011. <http://www.jedit.org/>.
[9] Columba, Columba, 2011. <http://sourceforge.net/projects/columba/>.
[10] JGIT, JGIT, 2011. <http://eclipse.org/jgit/>.
[11] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Urbana 51 (1997) 61801.
[12] S. Ducasse, M. Lanza, 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), Citeseer, 2000, pp. 1–7.
[13] C. Seguin, JRefactory, 2003. <http://jrefactory.sourceforge.net/csrefactory.html>.
[14] JetBrains, IntelliJ IDEA, 2012. <http://www.jetbrains.com/idea/>.
[15] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss, Emf model refactoring based on graph transformation concepts, Electronic Communications of the EASST 3 (2007).
[16] J. Kerievsky, Refactoring to Patterns, Pearson Education, 2005.
[17] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufman, 2002.
[18] S. Jeon, J. Lee, D. Bae, An automated refactoring approach to design pattern-based program transformations in java programs, in: Ninth Asia–Pacific Software Engineering Conference, 2002, IEEE, 2002, pp. 337–345.
[19] S. Lee, G. Bae, H.S. Chae, D.-H. Bae, Y.R. Kwon, Automated scheduling for clone-based refactoring using a competent *GA*, Software Practice & Experience 41 (2011) 521–550.
[20] Y. Higo, S. Kusumoto, 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 (2008) 435–461.
[21] M. Zibran, C. Roy, Conflict-aware optimal scheduling of code clone refactoring: a constraint programming approach, in: 2011 IEEE 19th International Conference on Program Comprehension (ICPC), IEEE, 2011, pp. 266–269.
[22] M. Harman, Refactoring as testability transformation, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2011, pp. 414–421.
[23] P. Oman, J. Hagemeister, Metrics for assessing a software system's maintainability, in: Proceerdings of Conference on Software Maintenance, 1992, pp. 337–344.
[24] N. Tsantalis, A. Chatzigeorgiou, Ranking refactoring suggestions based on historical volatility, in: 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2011, pp. 25–34.
[25] F. Simon, F. Steinbruckner, C. Lewerentz, Metrics based refactoring, in: Fifth European Conference on Software Maintenance and Reengineering, 2001. IEEE, 2001, pp. 30–38.
[26] N. Tsantalis, A. Chatzigeorgiou, Chatzigeorgiou, identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (2009) 347–367.
[27] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, 2006, p. 1916.

[28] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, ACM, 2007, pp. 1106–1113.
[29] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: a new way to save effort, IEEE Transactions on Software Engineering (2012). 1–1.
[30] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Software and Systems Modeling (2007).
[31] B. Du Bois, S. Demeyer, J. Verelst, Refactoring–improving coupling and cohesion of existing code, in: Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2004, pp. 144–151.
[32] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, in: Proceedings. International Conference on Software Maintenance, 2002, pp. 576–585.
[33] L. Tahvildari, K. Kontogiannis, A metric-based approach to enhance design quality through meta-pattern transformations, in: Proc. European Conf. Software Maintenance and Reeng., 2003, pp. 183–192.
[34] L. Zhao, J. 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, Citeseer, 2006, pp. 1–5.
[35] M. O'Keeffe, M.Ó. Cinnéide, Search-based refactoring for software maintenance, The Journal of Systems & Software 81 (2008) 502–516.
[36] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2004) 126–139.
[37] G. Arévalo, Understanding behavioral dependencies in class hierarchies using concept analysis, Proceedings of LMO 3 (2003) 47–59.
[38] OMG, UML 2.4 Superstructure Specification (formal/2010-05-05) edition, 2010. <http://www.omg.org/spec/UML/2.4/Superstructure/PDF/>.
[39] V. Garousi, L. Briand, Y. Labiche, Analysis and visualization of behavioral dependencies among distributed objects based on uml models, Model Driven Engineering Languages and Systems (2006) 365–379.
[40] W. Hwu, P. Chang, Achieving high instruction cache performance with an optimizing compiler, in: ACM SIGARCH Computer Architecture News, ACM, 1989. pp. 242–251.
[41] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, J. Whaley, The jalapeno dynamic optimizing compiler for java, in: Proceedings of the ACM 1999 conference on Java Grande, ACM, 1999. pp. 129–141.
[42] A. Srivastava, A. Eustace, Atom: A system for building customized program analysis tools, ACM SIGPLAN Notices 39 (2004) 528–539.
[43] P.V. Gorp, H. Stenten, T. Mens, S. Demeyer, Towards automating source-consistent uml refactorings, Lecture Notes in Computer Science (2003) 144–158.
[44] W.F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Framework, Ph.D. Thesis, University of Illinois at Urbana—Champaign, 1992.
[45] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, Prentice Hall PTR, 2002.
[46] B.D. Bois, S. Demeyer, J. Verelst, Refactoring–improving coupling and cohesion of existing code, in: Working Conference on Reverse Engineering, 2004, pp. 144–151.
[47] F. Bachmann, L. Bass, R. Nord, Modifiability Tactics, Technical Report, 2007.
[48] O. Seng, M. Bauer, M. Biehl, G. Pache, Search-based improvement of subsystem decompositions, in: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, ACM, pp. 1045–1051.
[49] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, E. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: 6th International Workshop on Program Comprehension, 1998, IWPC'98, IEEE, pp. 45–52.
[50] C. Bonja, E. Kidanmariam, Metrics for class cohesion and similarity between methods, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 91–95.
[51] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493.
[52] J. Bansiya, L. Etzkorn, C. Davis, W. Li, A class cohesion metric for object-oriented designs, journal of object-oriented program, Journal of Object-Oriented Program 11 (1999) 47–52.
[53] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems and Software 23 (1993) 111–122.
[54] L. Briand, J. Daly, J. Wust, A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on Software Engineering 25 (1999) 91–121.
[55] B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice-Hall Inc., Upper Saddle River, NJ, USA, 1996.
[56] A.-R. Han, ARTool, 2011. <http://github.com/igsong/ARTOOL>.
[57] R. Fagin, R. Kumar, D. Sivakumar, Comparing top k lists, in: Proceedings of the Fourteenth Annual ACM–SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2003, pp. 28–36.