Information and Software Technology 59 (2015) 53–66

Contents lists available at ScienceDirect

Information and Software Technology

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

An efficient approach to identify multiple and independent Move Method refactoring candidates

Ah-Rim Han [a], Doo-Hwan Bae [b], Sungdeok Cha [a,*]

[a] Department of Computer Science, Korea University, Anam-dong Sungbuk-gu, Seoul 136-701, South Korea
[b] Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, South Korea

ARTICLE INFO

Article history:
Received 18 March 2014
Received in revised form 27 October 2014
Accepted 28 October 2014
Available online 6 November 2014

Keywords:
Refactoring identification process
Refactoring selection
Multiple simultaneous refactorings
Maintainability improvement
Refactoring effect dependency

ABSTRACT

Context: Application of a refactoring operation creates a new set of dependency in the revised design as well as a new set of further refactoring candidates. In the studies of stepwise refactoring recommendation approaches, applying one refactoring at a time has been used, but is inefficient because the identification of the best candidate in each iteration of refactoring identification process is computation-intensive. Therefore, it is desirable to accurately identify multiple and independent candidates to enhance efficiency of refactoring process.
Objective: We propose an automated approach to identify multiple refactorings that can be applied simultaneously to maximize the maintainability improvement of software. Our approach can attain the same degree of maintainability enhancement as the method of the refactoring identification of the single best one, but in fewer iterations (lower computation cost).
Method: The concept of maximal independent set (MIS) enables us to identify multiple refactoring operations that can be applied simultaneously. Each MIS contains a group of refactoring candidates that neither affect (i.e., enable or disable) nor influence maintainability on each other. Refactoring effect delta table quantifies the degree of maintainability improvement each elementary candidate. For each iteration of the refactoring identification process, multiple refactorings that best improve maintainability are selected among sets of refactoring candidates (MISs).
Results: We demonstrate the effectiveness and efficiency of the proposed approach by simulating the refactoring operations on several large-scale open source projects such as jEdit, Columba, and JGit. The results show that our proposed approach can improve maintainability by the same degree or to a better extent than the competing method, choosing one refactoring candidate at a time, in a significantly smaller number of iterations. Thus, applying multiple refactorings at a time is both effective and efficient.
Conclusion: Our proposed approach helps improve the maintainability as well as the productivity of refactoring identification.

1. Introduction

While refactoring has been extensively researched and is frequently used in the industry, there still exists a gap between its theory and practice. In the industry, refactoring is largely a manual activity that relies heavily on a software developer's expertise. However, research community seeks to automate the refactoring process.

The major technical challenge in automated refactoring is the selection of the sequence of refactorings to perform. Each refactoring depends on the preceding applied refactoring. Thus, the application of a refactoring changes the system structure and may affect the applicability of other refactoring candidates or influence their effects on account of design quality factors, such as maintainability. Such phenomenon is known in evolutionary computation as epistasis [1,2]. Several approaches have been suggested to identify refactoring sequences to be applied. For example, aim of search-based techniques (e.g., [3–7]) is to determine the optimal refactoring sequence. Stepwise selection approach, on the other hand, attempts to find the most promising candidate in each iteration in greedy manner [8–10]. In the stepwise selection approach, the pool of refactoring candidates has to be re-computed at each iteration after the refactoring has been applied. However, it is inefficient and requires heavy computational load.

* Corresponding author. Tel.: +82 2 3290 4844; fax: +82 2 3290 3584.
E-mail addresses: arhan@korea.ac.kr (A.-R. Han), bae@se.kaist.ac.kr (D.-H. Bae), scha@korea.ac.kr (S. Cha).

http://dx.doi.org/10.1016/j.infsof.2014.10.007
0950-5849/© 2014 Elsevier B.V. All rights reserved.

In this paper, we propose an automated technique for identifying multiple refactorings that can be applied simultaneously by analyzing the dependencies of the refactorings and quantifying the degree of improvement in maintainability delivered by each elementary refactoring candidate. A brief procedure of the proposed approach is as follows. First, an object-oriented program is transformed into a design model that captures the entities (e.g., methods and attributes) and their dependencies. In order to evaluate the effect on maintainability of the operation of each elementary refactoring, a refactoring effect delta table is derived. In this table, the dependencies of entities across classes are used to quantify the degree of improvement in maintainability. Following this, using the concept of a maximal independent set (MIS) from graph theory [11–13], we group entities into MISs. Entities included in each set are independent of one another and are transformed into a group of refactoring candidates. Thus, refactoring operations can be applied simultaneously. Using the refactoring effect delta table, each group of refactoring candidates is assessed by accumulating the effect on maintainability of each elementary refactoring. Finally, the group of refactorings (i.e., multiple refactorings) that best improves maintainability is applied, the design model is updated and the refactoring effect delta table is calculated again. The process is repeated until no more refactoring candidates that improve maintainability of software can be found.

We apply our proposed approach to three open-source projects, jEdit [14], Columba [15], and JGit [16]. Experiments reveal that our approach helps increase the effectiveness and efficiency of the refactoring identification process by improving maintainability, while requiring smaller computational load than the approach selecting one refactoring candidate at a time.

Our proposed technique does not refactor an object-oriented program in a fully automated manner, but automatically identifies a set of refactoring candidates that can be safely applied simultaneously for delivering the maximum improvement on maintainability. The proposed method suggests refactoring candidates in ranked order of likelihood for maintainability improvement, and the software developer must make the final decision on whether or not to apply the suggested refactorings: even though the recommended refactorings are beneficial from a maintainability perspective, they might be rejected due to other factors.

We use Move Method refactoring to illustrate our approach. The application of each Move Method refactoring does not rule out the applicability of other refactoring candidates, but influences their effects on maintainability. The proposed method can be extended to big refactorings as well, once they are broken down to elementary-level refactorings (e.g., Move Method or Move Field refactorings) and duplications (e.g., conflicts) [17] and the dependencies among them are analyzed. In this paper, we do not consider Move Field refactoring, moving attributes (i.e., fields) from one class to another. We agree with the opinion [8] that fields are strongly conceptually bound to the classes in which they are initially placed, and are less likely to change than methods once assigned to a class.

The rest of the paper is organized as follows: Section 2 contains a discussion of related studies. Section 3 explains the overview of our proposed approach and the detailed methods for each procedure. In Section 4, we present the experiment to evaluate the proposed approach and discuss the results. Finally, we conclude and discuss avenues for future research in Section 5.

2. Related work

Many researchers have studied the methods for supporting automated refactoring process such as finding and evaluating refactoring candidates. Algorithms are developed to find refactoring candidates with the opportunities of applying design patterns [18–20], removing code clones [3,21,4,22] and improving code quality, such as testability [23] and maintainability. To evaluate the design of the refactored code, design quality evaluation models, such as the Quality Model for Object-Oriented Design (QMOOD) [3], and a special metric, such as historical volatility [24], are devised.

However, little research has been devoted to the selection procedure for refactorings to be applied from among the available candidates. In the following subsections, we present the studies that address issues related to refactoring selection.

2.1. Refactoring sequence determination

From the perspective of scheduling refactoring candidates and determining a sequence of refactorings to be applied, there have been two lines of study for automating the refactoring identification process: stepwise selection approach and search-based refactoring.

In refactoring identification using a stepwise selection method [8–10], the refactoring that best improves maintainability is selected in a stepwise manner among many of the extracted refactoring candidates. The entire refactoring identification process is repeated by re-extracting and re-assessing refactoring candidates. The stepwise approach offers the advantage of taking into account the changing system. Thus, complex dependencies among refactorings need not be considered, and newly created refactoring candidates can be taken into account. However, the stepwise approach can be inefficient as it selects *only one* refactoring for each iteration, even after assessing a large number of refactoring candidates.

Nonetheless, selecting multiple refactorings at a time by simulating the application of possible refactoring candidates that may be available after performing n more iterations (i.e., n-further steps) and by deferring the selection is difficult because the impact of a large number of refactoring candidates needs to be assessed. For instance, if the average number of available refactoring candidates (in each iteration of refactoring identification process) is m and the number of refactorings in a sequence is n (assuming that there is no repetition of refactoring candidates), the number of refactoring sequences that need to be examined is m^n. As the number of refactoring candidates increases, the number of possible refactoring sequences increases exponentially. Therefore, scheduling refactorings (i.e., selecting multiple refactorings at a time) by investigating all possible refactoring candidates exhaustively may become impossible (NP-hard).

In studies on search-based refactoring [25,6,3,26,5,4,7], researchers try to find an *optimal* (near-optimum) sequence of refactoring applications using search techniques. Lee et al. [3] and Seng et al. [26] use genetic algorithms to generate a sequence of refactorings to be applied to obtain an optimal system in terms of the employed fitness function. However, Seng et al. [26] do not take into account the fact that the application of a refactoring may create new refactoring candidates not originally present in the initial system. Moreover, some of the generated sequences of refactorings may not be feasible to be applied because of dependencies among refactoring candidates: applying one refactoring may conflict with the application of other refactorings. Seng et al. [26] do not perform a feasibility check on generated sequences of refactorings, and this may come at the additional cost of repairing. Lee et al. [3] try to resolve the refactoring-conflict problem by repairing infeasible sequences of refactorings that are randomly generated *without* regard for refactoring conflicts. However, it seems time consuming to reorder these sequences after generating them.

To generate a feasible sequence of refactorings considering conflicts and dependencies among refactoring candidates, Zibran and Roy [5,4,7] apply constraint programming (CP) techniques to the scheduling of code clone refactorings. First, they introduce an effort model for estimating developer's effort required to refactor code clones in procedural or object-oriented programs. Then, taking into account the effort model and a wide variety of possible hard and soft constraints, they formulate the scheduling of code clone refactorings as a constraint satisfaction optimization problem (CSOP) and solve it by applying CP techniques that aim to maximize benefits (measured in terms of changes in code/design quality metrics) while minimizing refactoring efforts. They also address the limitations of using genetic algorithms in optimization problems. They point out that genetic algorithms (1) do not seem to work well for CSOPs, since the core operations of genetic algorithms are based on random selection, which does not guarantee constraint satisfaction or optimization and (2) they are by nature time consuming and memory-intensive.

Our goal in this paper is *not* to find an optimal sequence of refactorings. In our approach, the sequence of multiple refactorings is generated after the termination of iterating the refactoring identification process. For instance, for each iteration, multiple refactorings that can be applied simultaneously to maximally improve maintainability are selected *in a stepwise manner*, then the entire process is repeated by re-identifying and re-evaluating refactoring candidates. By using this method, our approach takes the advantage of changing system by using the stepwise selection approach; and at the same time, compared to the approach of selecting only the best refactoring [8–10], our approach is more efficient in computation.

2.2. Refactoring dependency analysis

When selecting refactorings from available candidates, refactoring dependencies among them need to be considered to maximize maintainability. Mens et al. [17] represent refactorings as graph transformations and propose the techniques of critical pair analysis and sequential dependency analysis to detect dependencies among refactorings. Using the results of this analysis can help the developer make an informed decision about the most suitable refactoring in a given context. In a similar manner, Qayum and Heckel [27] represent the system using a graph model and the refactoring steps as graph transformation rules. The dependency information, which is derived from the analysis of graph transformations, is then used to express the problem as an instance of the optimization problem. Zibran and Roy [4] argue that the application of a subset of refactoring from a set of applicable refactoring activities may result in distinguishable impact on overall code quality. Moreover, there may be sequential dependencies and conflicts among the refactoring activities. Hence, they insist, it is necessary that a subset of non-conflicting refactoring activities from all refactoring candidates be selected and ordered (for application) such that the quality of the code base is maximized while the required effort is minimized.

In considering refactoring dependencies, the above-mentioned studies focus on applicability analyses or potential conflict analyses to determine whether the application of a refactoring changes or deletes the elements required for the application of other refactoring candidates. However, even though refactorings do not directly affect one another, the application of one refactoring may influence the degree to which another can subsequently improve maintainability of the software. The refactoring candidates are grouped into MISs in our proposed method according to the effect of their mutual dependencies on the system's maintainability. Thus, we can be assured that refactoring candidates grouped into the same MIS do not influence one another's effect on the maintainability of the software. The application of one Move Method refactoring, which is used in our paper, does not affect that of other Move Method refactoring candidates. Hence, we do not perform an applicability analysis.

3. Proposed approach

Fig. 1 shows our proposed refactoring identification process for identifying multiple refactorings that can be applied simultaneously. The input is the source code of an objected-oriented program.

First, the object-oriented program is transformed into an abstracted initial design model. In the abstracted design model, important entities (methods and attributes) and their dependencies are captured. To evaluate the effect of each elementary refactoring on maintainability, a refactoring effect delta table is derived from the design model (Section 3.1).

Following this, entities that are *not* interdependent are grouped into the same MIS (Section 3.2). By referring to the refactoring effect delta table, the methods involved in each MIS are then transformed into a group of refactoring candidates. Then, using the refactoring effect delta table, the accumulated values of the delta of maintainability for each group of refactoring candidates are sorted in order of expected degree of maintainability improvement.

Finally, a group of independent refactorings that best improve maintainability is applied followed by updates on design model as well as calculation of refactoring effect delta table. The preconditions that should be satisfied for a Move Method refactoring are formulated using rules reported in [8,28,29]. They include the followings: method to be moved should have one-to-one relationship with the target class; target class should not inherit a method having the same signature as the moved method; method to be moved should not override an inherited method in its original class; and method to be moved should not reference member variables of its original class. Before selecting a refactoring to be applied, the preconditions are checked to ensure behavior preservation.

The identification procedure for multiple refactorings is repeated until no more refactoring candidates that can improve maintainability can be found (Section 3.3). The output is a sequence of groups of refactorings, which are the selected and logged results obtained from each refactoring identification process.

3.1. Obtaining design model and refactoring effect delta table

Our goal for refactoring is to improve maintainability for the design of the object-oriented software. We measure maintainability based on the following concept. In object-oriented software, high cohesion and low coupling have been accepted as important factors for good software design quality in terms of maintenance [30]. *Cohesion* corresponds to the degree to which entities of a class belong together, and *coupling* refers to the strength of dependency from one class to another. For this reason, the number of intra-dependencies that entities belonging to a class (inner entities) have with the ones of the class itself should be as large as possible (high cohesion). At the same time, the number of inter-dependencies that entities of a given class have with the ones of other classes (outer entities) should be as small as possible (low coupling).

To this end, maintainability is quantified based on the *number of inter-dependencies* that exist between entities across classes in a system. Let n be an entity (method or attribute) of a system, c be a class of the system, and D denotes the set of dependencies existing in the system; and the number of inter-dependencies of entities is represented as follows:

$$\sum_{d \in D} (n_i, n_j) \in D, \quad where \quad n_i \in c_x, \quad n_j \in c_y, \quad and \quad c_x \neq c_y.$$

This number naturally represents the *lack* of degree of intra-dependency among entities of the same class (lack of cohesion) and, at the same time, the degree of inter-dependency among entities of different classes (coupling). Thus, it can be said that as the number of

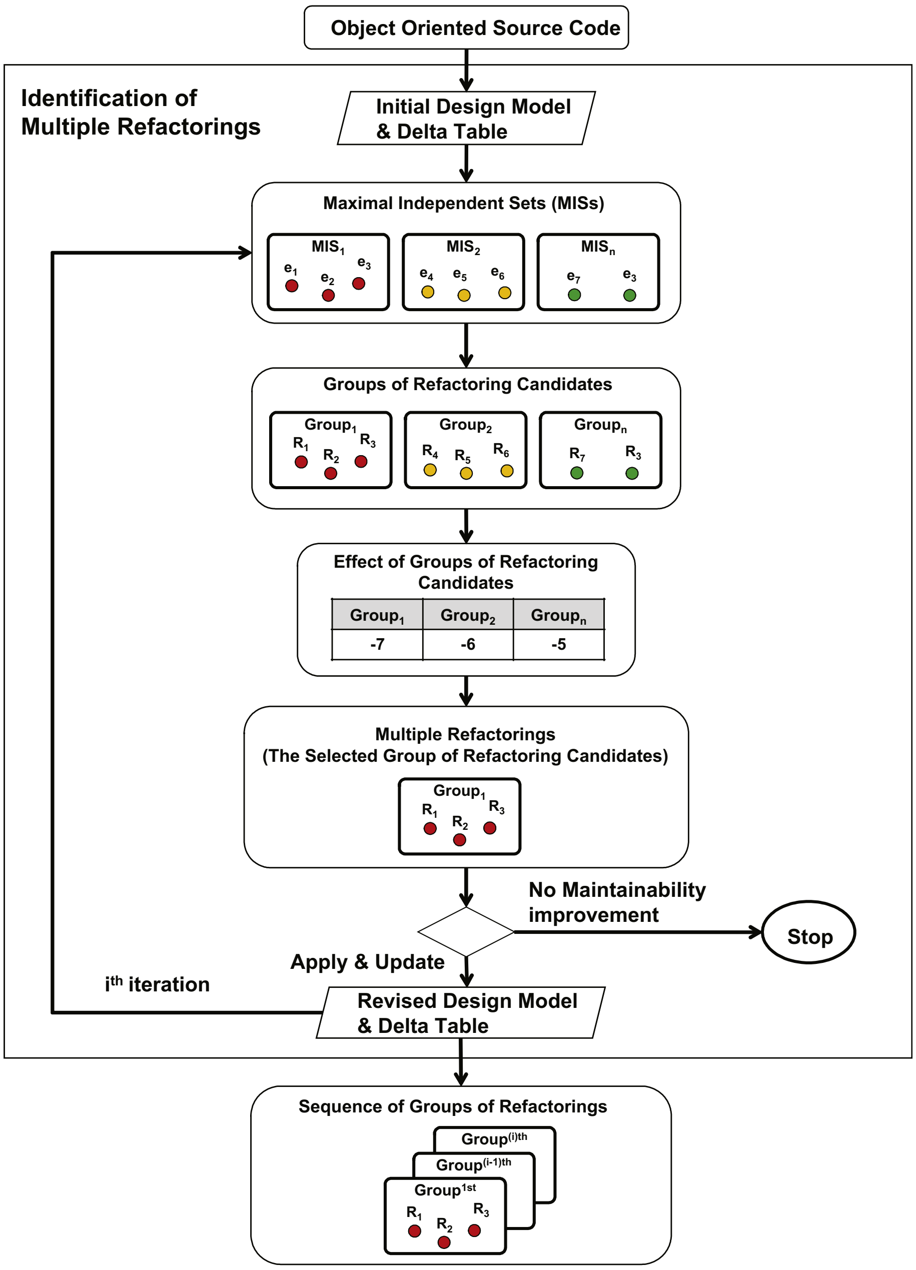

Object Oriented Source Code
Identification of Multiple Refactorings
Initial Design Model & Delta Table
Maximal Independent Sets (MISs)
MIS1 e1 e2 e3
MIS2 e4 e5 e6
MISn e7 e3
Groups of Refactoring Candidates
Group1 R1 R2 R3
Group2 R4 R5 R6
Groupn R7 R3
Effect of Groups of Refactoring Candidates
Group1 -7
Group2 -6
Groupn -5
Multiple Refactorings (The Selected Group of Refactoring Candidates)
Group1 R1 R2 R3
No Maintainability improvement
Stop
Apply & Update
ith iteration
Revised Design Model & Delta Table
Sequence of Groups of Refactorings
Group(i)th
Group(i-1)th
Group1st R1 R2 R3

Fig. 1. Overall approach of the proposed refactoring identification process for identifying multiple refactorings that can be applied simultaneously.

inter-dependencies gets decreases, the maintainability of the system improves. Consequently, the fitness function (used for refactoring selection criteria) is designed to be minimized by *reducing* this number for improving maintainability.

3.1.1. Design model

The design model captures important entities and their dependencies affecting maintainability, and is in the form of a graph. The initial design model $G_R = (N_R, E_R)$ is defined as follows.

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

The nodes (N_R) indicate the entities of methods and attributes. The node contains class membership information. The edges (E_R) indicate the dependencies between entities. By moving methods, we aim to improve maintainability, which reduces the dependencies between the entities (in terms of low coupling and high cohesion). Thus, the dependencies are captured when the entities of those dependencies are preferably located in the same class (referred from the object-oriented design heuristics [31]) in order to improve maintainability.

To this end, an edge is added between the entities when (1) a method calls the other method (method_calls), (2) a method accesses an attribute (attribute_accesses$_1$), and (3) two methods access the same attribute (attribute_accesses$_2$). The direction of the edges is not differentiated because maintainability is measured in terms of the number of inter-dependencies among entities across classes in a system. Thus, for the edges of attribute_accesses$_2$(method m_1, method m_2), an edge is added in any direction between two methods.

3.1.2. Refactoring effect delta table

A refactoring effect delta table is derived to quantify the degree of maintainability improvement after the application for each elementary refactoring candidate. In this table, the row elements indicate the moving methods and attributes while the column elements indicate the target classes. Each cell in the refactoring effect delta table indicates a Move Method refactoring candidate (row: moving method, column: target class). The value of each cell of the table indicates the effect of a refactoring candidate on maintainability (i.e., delta (Δ) of maintainability for the refactoring candidate), and it can be obtained by the difference of the number of the inter-dependencies after applying the refactoring candidate from the current design model (i.e., before applying the refactoring candidate). The refactoring candidate that reduces the number of the inter-dependencies at the greatest degree is the one that best improves maintainability.

The refactoring effect delta table is calculated based on the design model as follows. The value of each cell is obtained by adding the number of potentially increasing dependencies (which were intra-dependencies within classes) and the number of decreasing dependencies (which were inter-dependencies across classes) when moving an entity (row) to a class (column).

For example, in design model *n* of Fig. 2(a), the system consists of four classes and each class contains methods A = {m_2, m_6, m_7 m_8}, B = {m_1, m_3}, C = {m_4}, and D = {m_5 m_9}. Based on the design model, the refactoring effect delta table is calculated as in Fig. 2(c). Let MM(method *m*, class *c*) denote a Move Method refactoring (moving method *m* to target class *c*) in each cell. Let D_n[MM(method *m*, class *c*)] denote the delta of maintainability of the refactoring (i.e., moving method *m* to class *c*) for design model *n*. By using these notations, the value of the refactoring, for example MM(m_3, A), in the refactoring effect delta table is as follows: D_n[MM(m_3, A)] = −3. In other words, the refactoring carried out in moving method m_3 (located in class B) to target class A reduces the number of system's inter-dependencies by as much as −3. This change in the value of the systems maintainability is calculated by adding the number of decreasing dependencies (−4) and the number of potentially increasing dependencies (+1) across class A and class B. The decreasing dependencies are (m_7, m_3), (m_6, m_3), (m_8, m_3) and (m_2, m_3), whereas the increasing dependency is (m_3, m_1).

Application of a chosen refactoring may alter the delta value on maintainability of other refactoring candidates even though they remain enabled. For example, after MM(m_3, A) is applied, the values of the delta of maintainability change from Fig. 2(c) to (d). The shaded cells in Fig. 2(d) represent the refactorings of which the values of the delta of maintainability are changed from Fig. 2(c). Thus, for example, the expected effect of MM(m_3, C) is changed from zero to three. On the other hand, the same operation has no effect on Move Method refactorings of m_5 and m_9 because they are independent to m_3. Such property allows simultaneous application of multiple refactorings.

3.2. Calculating MISs

The definition of an MIS is explained as follows. Given a graph $G = (V, E)$, an independent set is a set of vertices $S \subseteq V$ such that if $u,\ v \in S$, then $(u, v) \notin E$. In short, an independent set is a set of vertices in *G* such that no two vertices in the set are adjacent (i.e., connected by an edge). An MIS is an independent set to which no more vertices can be added without violating the independence property. In short, an MIS is an independent set that is not a subset of any other independent set. A maximum independent set is an independent set with the *maximum* cardinality among all independent sets of *G*. Finding an MIS is trivial in a sequential algorithm. For a graph G, randomly pick a node and add it to the independent set (*I*), and remove the node and the associated nodes with their edges. This process is repeated until all nodes are removed, and the resulting set (*I*) will be an MIS of G. On the contrary, computing a maximum independent set is a notoriously difficult problem. It is equivalent to a maximum clique on a complementary graph. Both problems are NP-hard. Therefore, finding all existing MISs, the same problem for finding the maximum independent set, is also an NP-hard problem.

For the reason stated above, we use a heuristic to find MISs that is scalable to programs of large sizes. We try to find the MISs, each of which has as many independent entities as possible (which are later transformed into elementary refactorings). This is because the more entities there are in an MIS (i.e., the more refactorings that are transformed from among the entities of the MIS), the larger the expected maintainability improvement. For this, we first decompose the entities of the design model into independent sets by removing the nodes and the edges associated with them in order of the number of dependencies of each. Each removed node becomes an independent set, and nodes that come to have no edges (by the removal of the node and its associated edges) are added to an independent set as well. We then combine the independent sets and obtain MISs. In this way, it is highly likely that we will find independent sets that already have a large number of entities. By reducing the number of candidate independent sets (in short, by converting the unit of the independent sets from entities to groups of entities), we can obtain MISs *more quickly* than the method of finding MISs from entities.

Algorithm 1. getIndependentSet

```
var $N$ = a set of nodes, $E$ = a set of edges
  such that $(n_1, n_2) \in E \rightarrow n_1, n_2 \in N$
  begin
   while $N \neq \emptyset$ do
        $M := $ findMaxDependencyNode($N$, $E$)
        $E := \{(n_1, n_2)\ |n_1 \neq M,\ n_2 \neq M,\ (n_1, n_2) \in E\}$
        $N := N \setminus \{O :=$ findDanglingNodes($N, E$)
        $S := S\ \cup\ \{\{M\}, O\}$
   od
   print $\boldsymbol{S}$
where
funct findMaxDependencyNode($N$, $E$) ≡
    MaxDepCount := $-1$
    $M$ = undefined
    for n $\in N$ do
    $D := \{(n_1, n_2) \mid n_1 = n$ or $n_2 = n, (n_1, n_2) \in E\}$
    if $|D|$ > MaxDepCount then MaxDepCount := $|D|$, $M := n$ fi
  od
  return $M$
end
funct findDanglingNodes($N$, $E$) ≡
   $O$ := {}
  for $n \in N$ do
    $D := \{(n_1, n_2)\ |n_1$ = n or $n_2$ = n, $(n_1, n_2) \in E\}$
    if $|\boldsymbol{D}|$== 0 then $O := O \cup \{n\}$ fi
  od
  return $O$
end
```

Algorithm 2. calculateMaximalIndependentSet

```
var $S$ = a set of IS
  begin
    $S_{MIS}$ = $\emptyset$
    while $S \neq \emptyset$ do
      $IS_{max}$ = maxDep($S$)
      $MIS_{cur}$ = $\{IS_{max}\}$
      for $IS_1 \in S$ do
        if $IS_{max}$ does not have dependencies with $IS_1$
          then $MIS_{cur} := MIS_{cur} \cup \{IS_1\}$
        fi
      od
      $S_{MIS} := S_{MIS} \cup \{MIS_{cur}\}$
    od
    print $S_{MIS}$
  where
  funct maxDep($S$) $\equiv$
    MaxDepCount $:= -1$
    $M$ = undefined
    for $IS_1 \in S$ do
      $D := 0$
      for $IS_2 \in S$ do
        if $IS_1$ has dependencies with $IS_2$ then $D := D + 1$ fi
      od
      if $|D|$ > MaxDepCount then MaxDepCount $:= |D|$, $M := n$ fi
    od
    return $M$
  end
```

We calculate MISs by applying the above-mentioned heuristic. The algorithms for decomposing into independent sets and computing MISs are specified in Algorithm 1 and 2, respectively. The complexity of the algorithms are O(n_3).

First, the entities of the design model are decomposed into independent sets (*ISs*). Fig. 3 provides an illustrative example of the decomposition of entities of the design model into independent sets. In the initial design model (Fig. 3(a)), since m_3 has the largest number of dependencies, it is removed first and added into IS_1 (Fig. 3(b)). The edges associated with m_3 are removed as well (Fig. 3(c)). Therefore, the methods (m_1, m_4, m_6, m_7, and m_8) have no edges. These methods are added into IS_2 (Fig. 3(d)). By repeating this procedure, we obtain the following: $IS_1 = \{m_3\}$, $IS_2 = \{m_1, m_4, m_6, m_7, m_8\}$, $IS_3 = \{m_5\}$, $IS_4 = \{m_9\}$, and $IS_5 = \{m_2\}$.

The dependency relations among these independent sets are then obtained. Based on the dependencies among entities in each set, the dependencies between and among independent sets can be obtained. The dependency among entities indicates a relation on the design model $G_R = (N_R, E_R)$, such that two nodes (entities) $n_1, n_2 \in N_R$ and the edge (dependency) $(n_1, n_2) \in E_R$. We denote the dependency relation between entities, say n_1 and n_2, as $n_1 \Longleftrightarrow n_2$. Then, there exists a dependency between two independent sets, say IS_x and IS_y, such that $n_1 \in IS_x$, $n_2 \in IS_y$, and $n_1 \Longleftrightarrow n_2$. We denote the dependency relation between the independent sets as $IS_x \leftrightarrow IS_y$. For example, a dependency relation exists between independent sets IS_1 and IS_2 in Fig. 3 because an entity in each independent set ($m_3 \in IS_1$ and $m_4 \in IS_2$) has the dependency relation $m_3 \Longleftrightarrow m_4$. The results are as follows: $IS_1 \Longleftrightarrow IS_2$, $IS_1 \Longleftrightarrow IS_5$, $IS_3 \Longleftrightarrow IS_4$, $IS_3 \Longleftrightarrow IS_5$, and $IS_4 \Longleftrightarrow IS_5$.

Following this, each pair of independent sets is combined (united), unless the sets in question do not have a dependency relation. For instance, out of ${}_5C_2 = 5 \times 4/2 = 10$ combinations of pairs of independent sets, five pairs have dependency relations. Thus, we combine the remaining five pairs of independent sets as follows: $IS_6 = IS_1 \cup IS_3$, $IS_7 = IS_1 \cup IS_4$, $IS_8 = IS_2 \cup IS_3$, $IS_9 = IS_2 \cup IS_4$, and $IS_{10} = IS_2 \cup IS_5$. The combined independent sets can be united with other combined independent sets unless they do not have any dependency relations. The combined independent sets in this example cannot be united any further. Consequently, the MISs of entities are obtained as follows: $MIS_1 = \{m_3, m_5\}$, $MIS_2 = \{m_3, m_9\}$, $MIS_3 = \{m_1, m_4, m_6, m_7, m_8, m_5\}$, $MIS_4 = \{m_1, m_4, m_6, m_7, m_8, m_9\}$, and $MIS_5 = \{m_1, m_4, m_6, m_7, m_8, m_2\}$.

Since we consider only Move Method refactoring, as opposed to Move Field refactoring, MISs include only the methods. Excluding the attributes from the MISs does not mean that the attributes are disregarded for checking the preconditions to ensure behavior preservation, constructing the design model, or calculating the refactoring effect delta table.

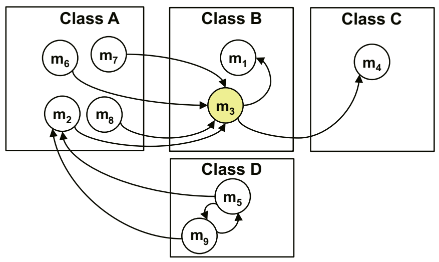
3.3. Selecting multiple refactorings

In the first step, a method in each MIS is instantiated into the specific refactoring designating the class to which the method moves. Using the refactoring effect delta table, each method *m* in an MIS is mapped into the Move Method refactoring that has the largest maintainability improvement among all available Move Method refactorings (method *m*, class *c*), where $c \neq$ owner class of method *m* and $c \in$ classes in the system. If there are no Move Method refactorings that improve the system's maintainability (i.e., that reduce the number of inter-dependencies), we do not assign the refactoring and let the method stay in the same class (i.e., we do not move it). In short, only moving methods that improve the systems maintainability are valid.

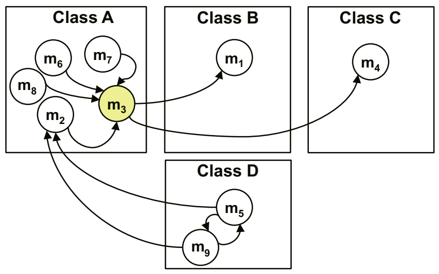
For example, in Fig. 4, m_3 in MIS_2 is mapped to MM(m_3, A). This is because among four refactorings (i.e., MM(m_3, A) = −3, MM(m_3, B) = '–', MM(m_3, C) = 0, and MM(m_3, D) = 1), MM(m_3, A) reduces the number of inter-dependencies to the greatest degree. The value of the delta of maintainability of MM(m_3, C) is zero because moving m_3, which was located in class B, to class C reduces the dependency (between methods m_3 and m_4) while increasing the dependency (between methods m_3 and m_1). In the refactoring effect delta table, '–' denotes the not-applicable refactoring (i.e., when moving a method to its owner class is meaningless); the value of the delta of maintainability for this not-applicable refactoring is zero because it does not change. When the values of the delta maintainability for two refactorings are smallest and are equal, then the refactoring is randomly assigned among them. The results of the transformed refactorings are represented in Fig. 4.

Following this, the effect on maintainability of each group of elementary refactorings is assessed by adding the values of the cells (represented as the delta of maintainability) of the corresponding elementary refactorings in the refactoring effect delta table. For example, in Fig. 4, the accumulated values of the delta of maintainability for each group are $Group_3 = -6$, $Group_4 = -4$, $Group_5 = -4$, $Group_1 = -3$, and $Group_2 = -3$.

The accumulated values of the groups of elementary refactorings are prioritized in descending order, and the group of elementary refactorings with the largest value is selected. We then check the stopping condition whether or not there is any maintainability improvement. If maintainability has improved, the selected refactorings are applied, the design model is updated and the refactoring effect delta table is recalculated. For example, among the five refactoring groups shown in Fig. 4, the refactorings in $Group_3 = -6$ reduce the number of inter-dependencies by the largest value thus are selected and applied. The identification procedure for multiple refactorings is repeated until no more refactoring candidates that can improve maintainability can be found. The selection procedure for multiple refactorings is then stopped, and the resulting sequence of groups of refactorings are generated.



(a) Example design model n.



(b) Design model (n + 1): After moving method m3 to class A from design model n of Fig. 2(a).

Moving Method

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

(c) Delta table D_n: Δ maintainability for each Move Method refactoring for the design model n of Fig. 2(a).

Moving Method

	Target Class			
D_{n+1}	A	B	C	D
m1	-1	-	0	0
m2	-	1	1	-1
m3	-	3	3	2
m4	-1	0	-	0
m5	1	2	2	-
m6	-	1	1	1
m7	-	1	1	1
m8	-	1	1	1
m9	1	2	2	-

(d) Delta table D_{n+1}: Δ maintainability for each Move Method refactoring for the design model (n + 1) of Fig. 2(b).

Fig. 2. Example of the design model and the refactoring effect delta table. (Δ maintainability represents the difference of the number of the inter-dependencies after applying each Move Method refactoring candidate, therefore, the more the number of the inter-dependencies reduces, the more maintainability improves.)

We need to ensure that proper constraint is applied to prevent our algorithm from merging all classes (e.g., God Class). Such action would have negative impact on maintainability. Specialization ratio S, whose value is computed by dividing the number of root classes divided by that of all classes, indicates the portion of clustered group of classes. If the S of the refactored model exceeds the threshold, the identification process is stopped. We used 0.45 (e.g., 45%) as a threshold value based on research reported in [32,33], but other values can be substituted if desired.

4. Evaluation

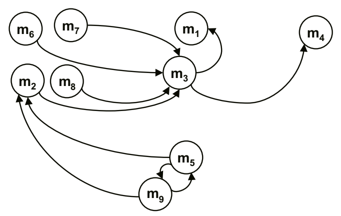
We evaluate the effectiveness and efficiency of our approach for identifying the multiple refactorings that can be applied simultaneously in terms of maintainability improvement and cost reduction. The research questions for our experiment are as follows.

- **RQ1. (Effectiveness):** Does the simultaneous application of multiple refactorings help improve maintainability?
- **RQ2. (Efficiency):** By how much does this approach reduce the computation cost of to achieve the same degree of maintainability?

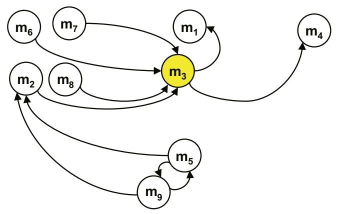
Three projects are chosen as experimental subjects: jEdit [14], Columba [15], and JGit [16]. A number of reasons led us to select these as subjects:

- They contain a relatively large number of classes.
- They are written in Java, and our proposed method applies to object-oriented software.
- They are widely used as experimental subjects in literature in the areas.

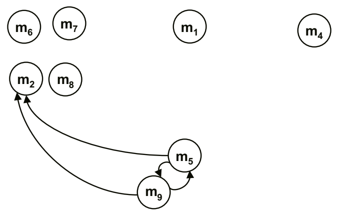
Table 1 summarizes characteristics of each subject.



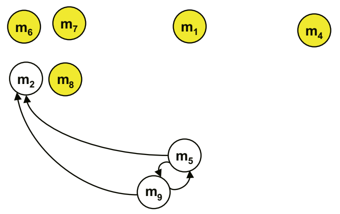
(a) Initial design model (same design model in Fig. 2(a)).



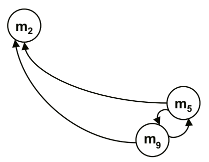
(b) Pick node m$_3$ and add it to set IS_1.



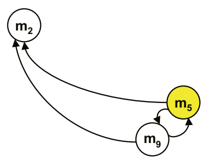
(c) Remove m$_3$ and the edges associated with it.



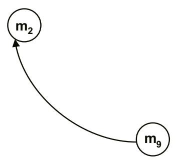
(d) Pick nodes m$_1$, m$_4$, m$_6$, m$_7$, m$_8$ (that have no edges) and add them to set IS_2.



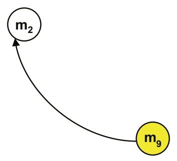
(e) Remove m$_1$, m$_4$, m$_6$, m$_7$, m$_8$.



(f) Pick node m$_5$ and add it to set IS_3.



(g) Remove m$_5$ and the edges associate with it.



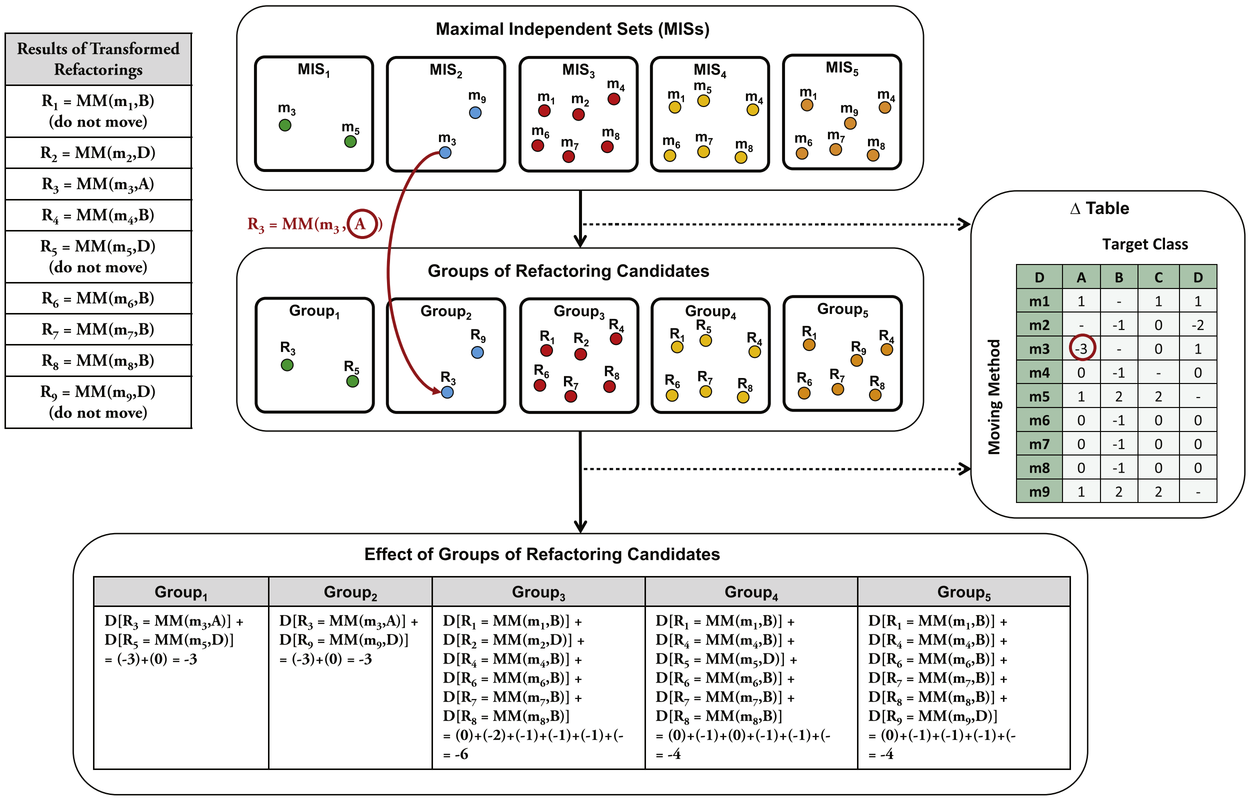
(h) Pick node m$_9$ and add it to set IS_4.

(i) Remove m$_9$ and the edges associate with it.

(j) Pick node m$_2$ and add it to set IS_5.

Fig. 3. Illustrative example of decomposing entities of the design model (in Fig. 2(a)) into independent sets (*IS*) for the design in Fig. 2(a). The results are IS_1 = {m$_3$}, IS_2 = {m$_1$, m$_4$, m$_6$, m$_7$, m$_8$}, IS_3 = {m$_5$}, IS_4 = {m$_9$}, and IS_5 = {m$_2$}.

Our approach considers all Move Method refactorings available in the system; thus, all the Move Method refactorings are regarded as refactoring candidates. A group of refactoring candidates obtained from an MIS is the unit of the refactoring application because those refactoring candidates in the group (set) are independent to each other and can be applied simultaneously.


Maximal Independent Sets (MISs)
MIS_1: m_3, m_5
MIS_2: m_9, m_3
MIS_3: m_1, m_2, m_4, m_6, m_7, m_8
MIS_4: m_1, m_5, m_4, m_6, m_7, m_8
MIS_5: m_1, m_9, m_4, m_6, m_7, m_8
R_3 = MM(m_3, $\bigcirc$A)
Groups of Refactoring Candidates
$Group_1$: R_3, R_5
$Group_2$: R_9, R_3
$Group_3$: R_1, R_2, R_4, R_6, R_7, R_8
$Group_4$: R_1, R_5, R_4, R_6, R_7, R_8
$Group_5$: R_1, R_9, R_4, R_6, R_7, R_8
Δ Table
Target Class
Moving Method
Effect of Groups of Refactoring Candidates

Results of Transformed Refactorings
R_1 = MM(m_1,B) (do not move)
R_2 = MM(m_2,D)
R_3 = MM(m_3,A)
R_4 = MM(m_4,B)
R_5 = MM(m_5,D) (do not move)
R_6 = MM(m_6,B)
R_7 = MM(m_7,B)
R_8 = MM(m_8,B)
R_9 = MM(m_9,D) (do not move)

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

$Group_1$	$Group_2$	$Group_3$	$Group_4$	$Group_5$
D[R_3 = MM(m_3,A)] + D[R_5 = MM(m_5,D)] = (-3)+(0) = -3	D[R_3 = MM(m_3,A)] + D[R_9 = MM(m_9,D)] = (-3)+(0) = -3	D[R_1 = MM(m_1,B)] + D[R_2 = MM(m_2,D)] + D[R_4 = MM(m_4,B)] + D[R_6 = MM(m_6,B)] + D[R_7 = MM(m_7,B)] + D[R_8 = MM(m_8,B)] = (0)+(-2)+(-1)+(-1)+(-1)+(- = -6	D[R_1 = MM(m_1,B)] + D[R_4 = MM(m_4,B)] + D[R_5 = MM(m_5,D)] + D[R_6 = MM(m_6,B)] + D[R_7 = MM(m_7,B)] + D[R_8 = MM(m_8,B)] = (0)+(-1)+(0)+(-1)+(-1)+(- = -4	D[R_1 = MM(m_1,B)] + D[R_4 = MM(m_4,B)] + D[R_6 = MM(m_6,B)] + D[R_7 = MM(m_7,B)] + D[R_8 = MM(m_8,B)] + D[R_9 = MM(m_9,D)] = (0)+(-1)+(-1)+(-1)+(- = -4

Fig. 4. Example of transforming methods involved in each MIS into a group of elementary refactorings and assessing the effect of the groups of refactorings.

Table 1
Characteristics for each subject.

Name (Version)	jEdit (jEdit-4.3)	Columba (Columba-1.4)	JGit (JGit-1.1.0)
Type	Text editor	Email client	Distributed source version control system
Class ♯	952	1506	689
Method ♯	6487	8745	5334
Attribute ♯	3523	3967	2989
MIS ♯	1272	1199	1190

Therefore, MISs can be regarded as the refactoring opportunities to be applied, and we add the number of the MISs obtained in each of the subject systems in the last row of Table 1.

4.1. Experimental design

We compare the approach of identifying multiple refactorings (our approach) with that of identifying a single refactoring in each iteration of the refactoring identification process. The approach for identifying a single refactoring follows the basic structure of the refactoring identification process illustrated in Fig. 1. In the single refactoring approach, refactoring candidates that are all Move Method refactorings available in the system are assessed using the refactoring effect delta table, and the refactoring candidate that best improves maintainability is selected and applied. Then, the entire refactoring identification process is iterated. Both of the approaches select refactoring(s) that best improve maintainability repeatedly in a stepwise manner by iterating the refactoring identification process. The final outputs are a sequence of groups of refactorings in our approach, while a sequence of refactorings in the single refactoring approach.

To answer the question about the *effectiveness* of our approach of identifying multiple refactorings (RQ1), we show that the refactored design that applies refactorings identified by our approach contributes to improving the maintainability of the system. The groups of refactorings are identified until the *final solution* is reached, where no more refactorings that improve maintainability are found, by repeating the refactoring identification process, and the sets of multiple refactorings are applied in the identified sequence to the original design.

For presenting maintainability improvement of the refactored design of the code, the *maintainability* is measured using a maintainability evaluation function [10], which was used to assess the contribution to improving maintainability of extracted refactoring candidates in our previous paper. It is designed as $\frac{cohesion}{coupling}$ to produce larger values as the system becomes more maintainable (with higher cohesion and lower coupling). The maintainability evaluation function of this design determines the merging of unrelated units of codes to be bad because this reduces coupling but lowers cohesion. Several metrics related to cohesion and coupling are normalized, weighted, and added up in this function. In addition to the maintainability evaluation function values, we also present two metrics, Method Similarity Cohesion (MSC) [34] and Message Passing Coupling (MPC) [35]. In MSC, the similarity among all pairs of methods is integrated and normalized to measure the cohesiveness of the class. MSC is different from other cohesion metrics in that it considers the degree of similarity between a pair of methods in a class. MPC is a commonly used metric for representing coupling and is appropriate to capture small code changes, such as moving a method to a class. MPC for a class *c* indicates the number of static method calls for all invoked imported methods.

On the other hand, to investigate the *efficiency* of multiple refactorings (RQ2), we compare the cost of the two approaches to reach the final solution and to obtain the same degree of maintainability improvement. We can show that our method is more efficient than the single refactoring selection approach in the following ways: (1) our approach can achieve a greater degree of maintainability improvement for the same cost and (2) our approach can attain the same degree of maintainability improvement at a lower cost. We choose the second method, which is easier to understand on account of simpler numbers.

The total computation cost (i.e., that which is required to generate the final solution, where no more refactorings that improve maintainability are found) is obtained by accumulating the computation cost required for each iteration of the refactoring identification process. The computation cost includes calculating MISs, calculating the refactoring effect delta table for assessing the refactoring candidates in each MIS, and applying the selected multiple refactorings by updating the design model. The computation cost is measured in terms of the number of iterations and the elapsed time (i.e., run-time). The elapsed time can be affected by environmental (external) factor; however, it is proportional to the total number of iterations and can therefore be an alternative indicator representing the computation cost. The elapsed time (s) is measured under the following conditions: processor 1.8 GHz Intel Core i5, Memory 8G 1600 MHz DDR3, Graphic Intel HD Graphics 4000 512 MB, and Software OS X 10.8.2.

4.2. Results

4.2.1. RQ1: Effectiveness

Table 2 summarizes the results of maintainability (maintainability evaluation function values, MSC (cohesion metric) and MPC (coupling metric)), where each approach has arrived at the final solution for jEdit, Columba, and JGit, respectively. The graphs in Fig. 5 shows the visual results. In all three projects, the maintainability evaluation function values of our approach are greater than those of the original design. In addition, the MSC value increases while the MPC value decreases. Therefore, our approach contributes to the improvement of maintainability.

As shown in Table 2, the values of the maintainability evaluation function obtained from our approach are higher than those of the single refactoring approach (e.g., 0.0317 vs 0.0303 for jEdit). On the case studies performed on three real-world examples, our approach outperformed the single refactoring approches. However, application on other examples may yield different performance. Goal of this research is not to prove that our approach will always outperform alternative approaches. Rather, we convincingly demonstrated that our approach will deliver effective improvement on maintainability.

The final sequence of applied refactorings are different in the two approaches, thus the final achievement for maintainability improvement become different. This is because a method of selection for each iteration results in determining which refactoring(s) be applied first, and subsequently, next available refactoring candidates are changed. More specifically, according to the selection method, the preceding applied refactoring(s) is/are determined. This situation subsequently changes the most beneficial (or available) refactoring candidates at the time of the next selection. When terminating the process, a selected sequence of refactorings is generated and it becomes the suggested sequence of refactorings to be applied. As a result, according to the selection method, the applied refactorings become different.

4.2.2. RQ2: Efficiency

Table 3 summarizes the results of the required costs (in terms of the number of iterations) to reach the final solution (see Table 2) and the number of selected refactorings per iteration of the two competing approaches for jEdit, Columba, and JGit, respectively. The total number of iterations required to reach the final solution using our approach is much smaller than that required by the method of selecting a single refactoring: jEdit ($26 < 1586$), Columba ($39 < 2290$), and JGit ($74 < 620$).

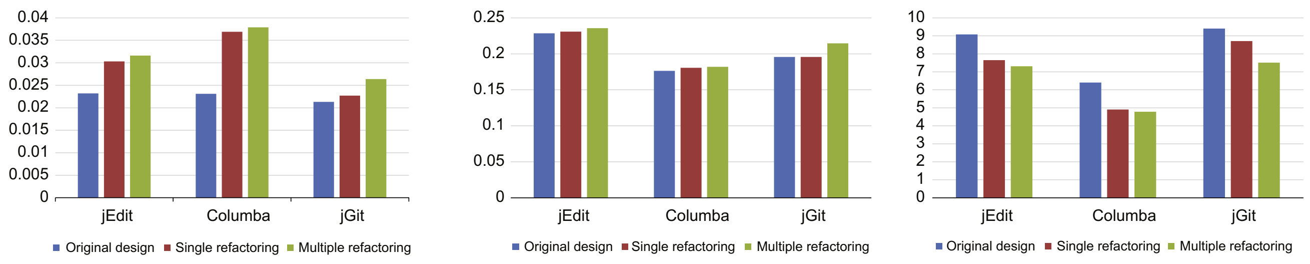
In addition, Fig. 6 shows the results of the required costs (in terms of the number of iterations) to accomplish the same degree of maintainability improvement. We compare the number of iterations required to accomplish n ($n = 25$, 50, 75, 100) percent(%) of maintainability improvement. The maintainability improvement indicates the difference between maintainability evaluation function values of the final solution and the original design. To fairly compare the efficiency of two competing approaches, we set the smaller degree of maintainability improvement attained between the two competing approaches to serve as the baseline for comparison of efficiency in achieving the same degree of maintainability improvement. Thus, the improvement achieved by the single approach becomes the baseline for comparison because our approach improves maintainability by a greater degree than the single approach for all projects (as shown in Table 2). When calculating the number of iterations required for a specific degree of maintainability improvement, the number is rounded off to the nearest whole number. As the final outcome, our approach reaches the same degree of maintainability with a much smaller number of iterations than the competing approach of selecting a single refactoring.

For further analysis, we take a close look in Fig. 7 at the graphs of required costs (in terms of elapsed time) to reach the final solution for jEdit, Columba, and JGit, respectively. The x-axis shows the elapsed time, and the y-axis shows the maintainability evaluation function values. In both jEdit and Columba (Fig. 7(a) and (b),

Table 2
Results of the effectiveness of multiple refactorings.

Subject	Competitors	Maintainability		
		Maintainability evaluation fn. [10]	MSC [34]	MPC [35]
jEdit	Original design	0.0232	0.2286	9.0830
	Single refactoring	0.0303	0.2309	7.6513
	Multiple refactoring (our approach)	0.0317	0.2357	7.3088
Columba	Original design	0.0231	0.1763	6.4031
	Single refactoring	0.0369	0.1807	4.9037
	Multiple refactoring (our approach)	0.0379	0.1819	4.7822
JGit	Original design	0.0213	0.1956	9.4021
	Single refactoring	0.0227	0.1957	8.7097
	Multiple refactoring (Our approach)	0.0264	0.2147	7.5137

⁎ Original design: initial design of software (before applying refactorings).
⁎ Single refactoring: approach of selecting a single refactoring when reaching to the final solution†.
⁎ Our approach: approach of selecting multiple refactorings when reaching to the final solution†.
⁎ † Final solution: no more refactorings that improve maintainability are found by repeating refactoring identification process.



(a) Maintainability evaluation fn. [10] (b) MSC [34] (c) MPC [35]

Fig. 5. The graphs for visualizing the effect of multiple refactorings.

Table 3
Results of the efficiency of multiple refactorings—the required number of iterations to reach the final solution.

Subject	Competitors	Computation cost1	Reference			
		♯ of iterations	♯ of applied refactorings (per iteration)			
		Final solution[†]	Avg.	Max.	Min.	Std dev.
jEdit	Single refactoring	1586	1	1	1	0
	Multiple refactoring (Our approach)	26	60.5	454	3	109.3
Columba	Single refactoring	2290	1	1	1	0
	Multiple refactoring (Our approach)	39	76.8	629	2	81.4
JGit	Single refactoring	620	1	1	1	0
	Multiple refactoring (Our approach)	74	22.5	569	2	73.2

⁎ Single refactoring: approach of selecting a single refactoring when reaching to the final solution.
⁎ Our approach: approach of selecting multiple refactorings when reaching to the final solution.
[†] Final solution: no more refactorings that improve maintainability are found by repeating refactoring identification process. (see Table 2).

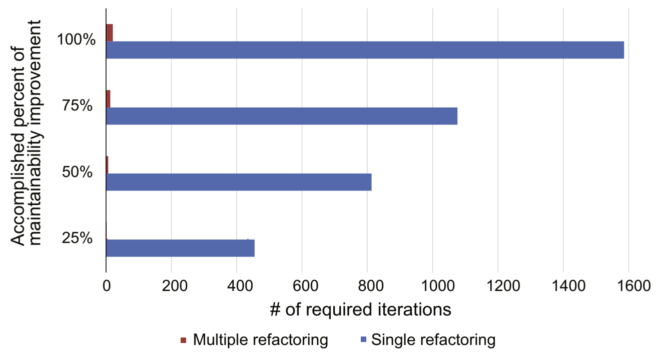
respectively), maintainability evaluation function values of our approach increase rapidly, while those of the single refactoring approach increase comparatively slowly. Even though there is an overhead to compute MISs in the first step for the selection of multiple refactorings (denoted as "Preprocessing Time" in the graphs in Fig. 7), our approach can attain the same degree of maintainability improvement at a much lower cost (i.e., time or the number of iterations). Thus, the benefit of net reduced time more than outweighs this necessary overhead.

As shown in Table 3 and Fig. 7(c), performance difference on JGit is substantially greater than that of jEdit or Columba. Maintainability improvement on single refactoring approach became almost non-existent due to local optimum problem. By analyzing the applied refactorings identified from the single refactoring approach, we observed that it often selected ones that methods in Command classes be moved to `Revwalk` or `Treewalk` classes. This is because the Command classes are reified classes in which methods do not often have references with attributes; and methods in the Command classes had a large number of dependencies with the `Revwalk` and `Treewalk` classes. Accordingly, the next (or the second) most promising refactorings are rarely selected, even though the differences between the degree of maintainability improvement of those refactorings and that of the best refactorings are subtle and negligible. Applying some of the next most promising refactorings (e.g., moving methods in Command classes, such as `MergeCommand`, `RebaseCommand`, and `PullCommand`, to one another) would contribute more to increasing the maintainability at the end. Such pattern is repeated until no improvements are found, thus falling into the trap of local optimum. Our technique, on the other hand, allows methods that do not have any dependencies be moved simultaneously. Therefore, in some cases, selecting multiple refactorings makes the effects of performing the simulated annealing (a search technique for finding an actual best solution by exploring possible solutions accepting better solutions as well as worse solutions), which prevents getting stuck in the local optimum.

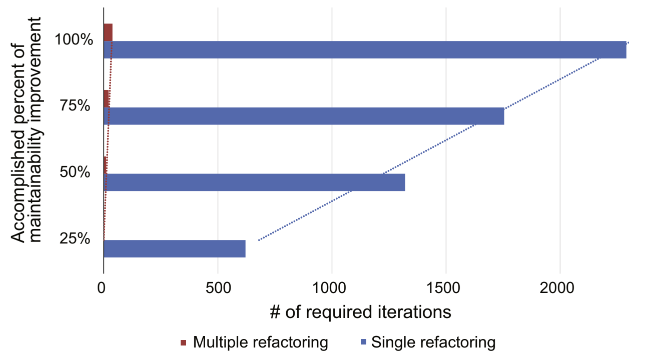
4.2.3. Discussion and conclusion

Our approach of selecting multiple refactorings could be more complex and thus harder to use than the approach of selecting a single one at each step. Yet, in our approach, the smallest unit of refactoring application is the group of Move Method refactorings that can be applied at the same time. The entire refactoring process is repeated by identifying and assessing refactoring candidates. On the other hand, big refactorings (consisting of small refactorings having complex dependencies) are very hard to safely execute because they could affect a larger portion of the source code and thus degrade understandability. Therefore, compared to big refactorings, the required application cost of relocating codes or retesting affected portions of codes would be smaller using the multiple refactorings identified in our approach.

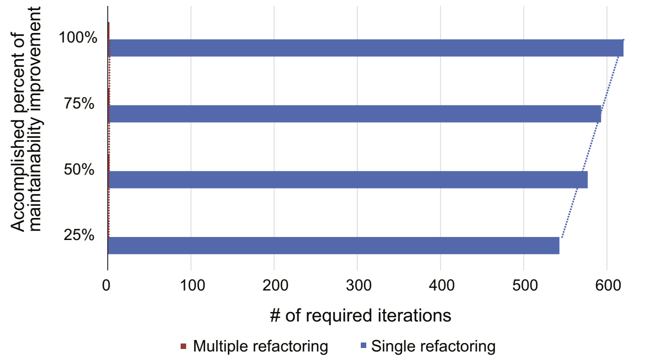
From the results, we can conclude that our approach helps enhance the effectiveness and efficiency of the refactoring identification process. Compared to the selection method involving single refactoring, our approach selects refactorings that improve the maintainability of the software design at lower computation costs (in terms of smaller iterations or shorter elapsed time). Even though our method requires an overhead to compute MISs at the beginning of the refactoring identification process, the benefit of net reduced time more than outweighs this necessary overhead. Furthermore, the single refactoring approach may face the local optimum problem. In this experiment, our approach tends to perform better in avoiding local optima by selecting refactorings evenly throughout the software.



(a) jEdit

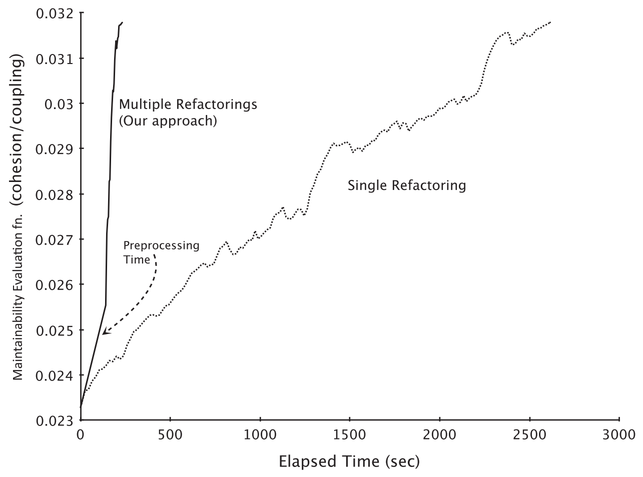


(b) Columba

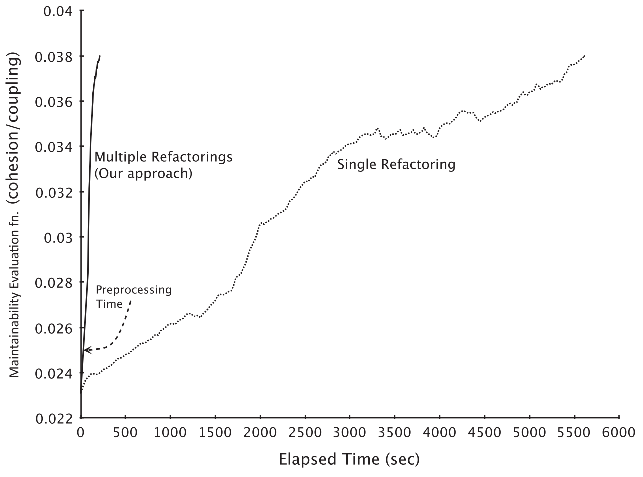


(c) JGit

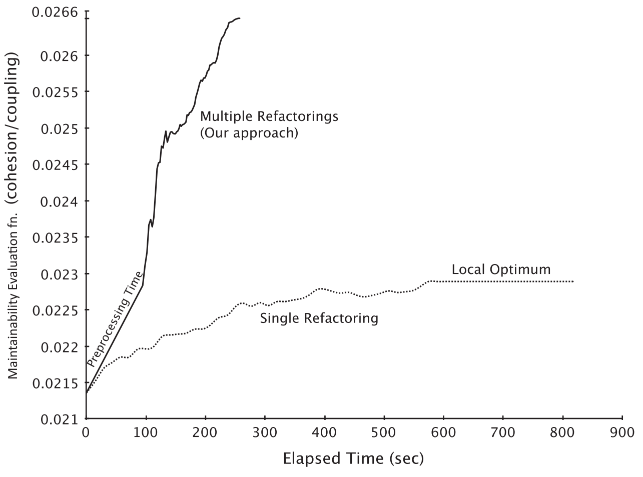
Fig. 6. Results of the efficiency of multiple refactorings—the number of iterations required to accomplish the same degree of maintainability improvement. Note that we set the smaller degree of maintainability improvement to serve as the *baseline* for comparison.



(a) jEdit



(b) Columba



(c) JGit

Fig. 7. Results of the effect of multiple refactorings on time.

4.3. Threats to validity

In this section, we address the possible threats to validity in the four perspectives: construct validity, internal validity, external validity, and reliability.

4.3.1. Construct validity

In [10], we defined the maintainability evaluation function as a fitness function by combining cohesion (numerator) and coupling (denominator) into one numerical value. For this reason, values are generally very small and the significance on the performance difference may appear negligible. As supplementary values, we also use cohesion and coupling metrics (e.g., MSC and MPC) as other studies [8,36] did.

In this paper, we do not make direct comparison on the quality of the identification of refactoring candidates [36,37,8], as the main contribution of our paper is to select multiple refactorings that can be applied simultaneously for every iteration of the refactoring identification process. The initial refactoring opportunities can be taken from any refactoring recommendation tool if desired as long as dependencies among candidates are established; then, the groups of multiple refactorings can be identified.

Basically, the single refactoring approach follows the same refactoring identification approach with ours but selects one refactoring candidate that best improves maintainability among all Move Method refactorings available in the system. With regard to the efficiency of the computation for refactoring identification, the single refactoring can be the main competitor because the aim of evaluation is to show that the application of multiple refactorings at a time significantly reduces the number of iterations required to perform (compared to the application of a single refactoring).

We do not make direct comparison on the memory consumption against the single refactoring approach. Both techniques are essentially stepwise selection approaches where refactoring candidates are repeatedly identified and evaluated at each iteration. Memory requirements of two approaches in each iteration are almost same. Furthermore, comparison on accumulated amount of memory consumption is meaningless because it is freed at the end of each iteration. On the other hand, if search-based refactoring techniques are applied, memory consumption would be substantially large due to the application of backtracking algorithm where candidates are incrementally built and kept until they are determined not to be valid (or final) solutions.

4.3.2. Internal validity

In this paper, we focus on relations established by method calling procedures when capturing dependencies among entities in the design model. The dependencies related to the methods that do not affect the behavior of the system (e.g., getter/setter methods and delegate methods) are excluded when constructing the design model. For dependencies that may affect maintainability, other types of dependencies caused by structural relations between classes (such as association, aggregation, composition, and inheritance) can be considered.

We assume that the application of Move Method refactorings does not delete or merge entities constituting the refactorings and, therefore, the nodes and edges of the initial design model G_R remain the same after the application of the selected refactorings. Therefore, in our paper, MISs do not need to be recalculated for every iteration of the refactoring identification process. We plan to consider other types of refactoring in our future work. For taking into these types of refactoring account, we need to develop a method to efficiently recalculate MISs to accommodate situations where the application of refactorings deletes or merges entities.

4.3.3. External validity

Controlled study performed on large and complex open source software convincingly demonstrated that proposed approach is highly effective compared to single refactoring approach. Assuming that chosen applications exhibit characteristics of object-oriented software that are subject to design refactoring, similar results are expected although the degree of maintainability improvement would vary from one application to another.

4.3.4. Reliability

The proposed method has been implemented using Python and the data used for the experiment is available online [38]. We use an efficient method for calculating the refactoring effect delta table using matrix computation. By only changing the link and membership matrices (the modeling configuration of the software design) and manipulating those matrices, the "delta of maintainability" for the application of a Move Method refactoring candidate on the design configuration can be easily obtained at once. The matrix computation is fast because there are various scientific and numerical techniques to accelerate the speed (e.g., we use SciPy [39] libraries implemented for Python). Assessing the effects of refactoring candidates is the most computation-intensive part, so this method helps to reduce the computation time.

5. Conclusion

In this paper, we provided an automated method for selecting multiple refactorings that can be applied simultaneously to improve the efficiency of the refactoring identification process. To determine the groups of refactorings that can be applied independently, the entities are grouped into MISs. By using a refactoring effect delta table, each group of refactorings is assessed and sorted in order of expected degree of improvement on maintainability. Finally, we select the multiple refactorings that best improves maintainability. Experimental results showed that the method of selecting multiple refactorings helps enhance the effectiveness and efficiency of the refactoring identification process by improving maintainability while reducing the computation cost. Our approach also showed better performance in avoiding local optima by selecting refactorings evenly throughout the software.

While the most promising refactoring candidates are identified automatically, the software developer makes the final decision with respect to applying the suggested candidates based on experience and design principles.

For future research, we plan to consider other types of refactoring. Our method of the refactoring effect delta table can support the extension of the refactoring process to other types of refactoring. This is feasible because our approach provides a method of assessing an elementary refactoring, and big refactorings (e.g., Collapse Hierarchy Class refactoring, Pull Up Method refactoring) comprise of those elementary refactorings (e.g., Move Method refactoring).

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054098). This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014-H0301-14-1023) supervised by the NIPA (National IT Industry Promotion Agency).

References

[1] Y. Davidor, Epistasis variance: a viewpoint on ga-hardness, in: Proceedings of the First Workshop on Foundations of Genetic Algorithms, Bloomington Campus, Indiana, USA, July 15–18, 1990, pp. 23–35.
[2] A.E. Eiben, P.-E. Raué, Z. Ruttkay, Solving constraint satisfaction problems using genetic algorithms, in: International Conference on Evolutionary Computation, 1994, pp. 542–547.
[3] S. Lee, G. Bae, H.S. Chae, D.-H. Bae, Y.R. Kwon, Automated scheduling for clone-based refactoring using a competent GA, Softw., Pract. Exper. 41 (2011) 521–550.
[4] M.F. Zibran, C.K. 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.
[5] M.F. Zibran, C.K. Roy, Conflict-aware optimal scheduling of prioritised code clone refactoring, IET Softw. 7 (2013).
[6] H. Liu, G. Li, Z. Ma, W. Shao, Conflict-aware schedule of software refactorings, IET Softw. 2 (2008) 446–460.
[7] M.F. Zibran, C.K. Roy, A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring, in: 2011 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2011, pp. 105–114.
[8] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (2009) 347–367.
[9] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Identification and application of extract class refactorings in object-oriented systems, J. Syst. Softw. (2012).
[10] A.-R. Han, D.-H. Bae, Dynamic profiling-based approach to identifying cost-effective refactorings, Inf. Softw. Technol. 55 (2013) 966–985.

[11] N. Alon, L. Babai, A. Itai, A fast and simple randomized parallel algorithm for the maximal independent set problem, J. Algorithms 7 (1986) 567–583.
[12] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM J. Comput. 15 (1986) 1036–1053.
[13] D.S. Johnson, M. Yannakakis, C.H. Papadimitriou, On generating all maximal independent sets, Inform. Process. Lett. 27 (1988) 119–123.
[14] jEdit, jEdit, 2011. <http://www.jedit.org/>.
[15] Columba, Columba, 2011. <http://sourceforge.net/projects/columba/>.
[16] JGIT, JGIT, 2011. <http://eclipse.org/jgit/>.
[17] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Softw. Syst. Model. (2007).
[18] J. Kerievsky, Refactoring to Patterns, Addison Wesley Pearson Education, 2005.
[19] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-oriented Reengineering Patterns, Morgan Kaufmann, 2002.
[20] S. Jeon, J. Lee, D. Bae, An automated refactoring approach to design pattern-based program transformations in java programs, in: Software Engineering Conference, 2002, Ninth Asia-Pacific, IEEE, 2002, pp. 337–345.
[21] Y. Higo, S. Kusumoto, K. Inoue, A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system, J. Softw. Maint. Evol.: Res. Pract. 20 (2008) 435–461.
[22] K. Hotta, Y. Higo, S. Kusumoto, Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph, in: 16th European Conference on Software Maintenance and Reengineering (CSMR'12), 2012, pp. 53–62.
[23] 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.
[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] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: a new way to save effort, IEEE Trans. Softw. Eng. 38 (2012) 220–235.
[26] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (2006) 1916.
[27] F. Qayum, R. Heckel, Search-based refactoring using unfolding of graph transformation systems, Electron. Commun. EASST 38 (2011).
[28] P.V. Gorp, H. Stenten, T. Mens, S. Demeyer, Towards automating source-consistent UML refactorings, Lect. Notes Comput. Sci. (2003) 144–158.
[29] W.F. Opdyke, Refactoring: A Program Restructuring Aid in Designing Object-oriented Application Framework, Ph.D. Thesis, University of Illinois at Urbana–Champaign, 1992.
[30] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering, Prentice Hall PTR, 2002.
[31] A.J. Riel, Object-oriented Design Heuristics, Addison-Wesley Publishing Company, 1996.
[32] S. Vaucher, F. Khomh, N. Moha, Y.-G. Guéhéneuc, Tracking design smells: lessons from a study of god classes, in: 16th Working Conference on Reverse Engineering, 2009, WCRE'09, IEEE, 2009, pp. 145–154.
[33] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: 9th International Conference on Quality Software, 2009, QSIC'09, IEEE, 2009, pp. 305–314.
[34] C. Bonja, E. Kidanmariam, Metrics for class cohesion and similarity between methods, Proceedings of the 44th Annual Southeast Regional Conference (2006) 91–95.
[35] W. Li, S. Henry, Object-oriented metrics that predict maintainability, J. Syst. Softw. 23 (1993) 111–122.
[36] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A.D. Lucia, Methodbook: recommending move method refactorings via relational topic models, IEEE Trans. Softw. Eng. 40 (2014) 671–694.
[37] V. Sales, R. Terra, L.F. Miranda, M.T. Valente, Recommending move method refactorings using dependency sets, in: 20th Working Conference on Reverse Engineering (WCRE), 2013, IEEE, 2013, pp. 232–241.
[38] A.-R. Han, Multiple Refactoring Identification Tool, 2014. <https://github.com/ahrimhan/mass-refactoring>.
[39] SciPy, SciPy, 2014. <http://www.scipy.org/>.