Automatic Software Refactoring via Weighted Clustering in Method-Level Networks

Ying Wang, Hai Yu, Zhiliang Zhu, *Member, IEEE*, Wei Zhang, and Yuli Zhao

Abstract—In this study, we describe a system-level multiple refactoring algorithm, which can identify the move method, move field, and extract class refactoring opportunities automatically according to the principle of "high cohesion and low coupling." The algorithm works by merging and splitting related classes to obtain the optimal functionality distribution from the system-level. Furthermore, we present a weighted clustering algorithm for regrouping the entities in a system based on merged method-level networks. Using a series of preprocessing steps and preconditions, the "bad smells" introduced by cohesion and coupling problems can be removed from both the non-inheritance and inheritance hierarchies without changing the code behaviors. We rank the refactoring suggestions based on the anticipated benefits that they bring to the system. Based on comparisons with related research and assessing the refactoring results using quality metrics and empirical evaluation, we show that the proposed approach performs well in different systems and is beneficial from the perspective of the original developers. Finally, an open source tool is implemented to support the proposed approach.

Index Terms—Clustering analysis, cohesion, coupling, complex network, software refactoring

1 Introduction

In long-term development and maintenance processes, software inevitably undergoes continuous modifications due to new requirements, where these changes may cause the code to decay and drift from the software design. In addition, the complexity of the system increases over time and tight development schedules may force poor design decisions, which can exacerbate the problem. Thus, refactoring is an integral part of software development [1]. Refactoring operations can improve the understandability, maintainability, reusability, and flexibility of software by adjusting its internal structure without changing the external behaviors of the code [2]. Due to the expansion of software scales and the extension of maintenance cycles, developers need to perform refactoring operations continuously to improve the quality of the software system [3].

"High cohesion and low coupling" is one of the most important software design guidelines [4]. Cohesion is an indicator of the connection strength between the elements of a component and coupling measures the dependencies between different components [5]. At the system-level, remodularizing software components can produce a set of subsystems containing modules that cooperate to implement strongly related functionalities. To obtain insights into the system design and structure as well as a thorough understanding of its organization, software clustering algorithms are applied to create abstract structural views of the entities and relationships present in the source code. These abstract structural views are used to "navigate" the system and developers can use them to optimize software development and maintenance. The first step in the typical software clustering technique is representing the modules (e.g., classes, files, and packages) and module-level relationships as a module dependency graph. Clustering algorithms are then used to partition the graph, so the high level subsystem structure can be derived from component level relationships [6], [7]. These approaches use Move Class refactoring algorithms, which not only positively impact the internal cohesion of the package but also decrease the coupling between packages [8].

Cohesion and coupling problems may cause the following bad smells at the class level.

(1) Feature Envy: this represents a type of behavior that occurs when a method is more dependent on other classes than the class to which it belongs [9].
(2) Inappropriate Intimacy: this is a typical smell that occurs if the dependence relationships between two classes are too close [10].
(3) God Class: if a class encapsulates too many functions, its understandability, reusability, and extendibility will become poor [11].

We can move methods or fields to classes with more dependencies to remove the feature envy and inappropriate intimacy smells from the code, which is known as move method/field refactoring [12]. Moreover, the god class problem can be solved by extract class refactoring. According to the principle of "high cohesion and low coupling," strongly related methods and fields are extracted from the original class into the new class. This means that one god class should be decomposed into two or more new classes [13].

• *The authors are with the Software College, Northeastern University, Shenyang 110819, China. E-mail: {wangying8052, zhaoyl_neu}@163.com, yuhai@126.com, zzl@mail.neu.edu.cn, zhangwei@swc.neu.edu.cn.*

Manuscript received 17 Aug. 2016; revised 13 Feb. 2017; accepted 5 Mar. 2017. Date of publication 7 Mar. 2017; date of current version 21 Mar. 2018. Recommended for acceptance by M. Di Penta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2679752

Assigning class responsibilities based on human judgment and decision making is complicated and time-consuming. Thus, in this study, we propose an automatic system-level multiple refactoring algorithm based on complex network theory. All of the methods and attributes defined in the related classes are merged into an entity set and then regrouped by clustering analysis. Based on comparisons between the new classes and the original classes, we can identify multiple opportunities, including move method, move field, and extract class refactorings. The main contributions of this study are summarized as follows.

- A multi-relation network model for analyzing inheritance and non-inheritance relationships. Based on the model, we can reconstruct non-inheritance and inheritance hierarchies by setting a series of preconditions and preprocessing.
- A weighted clustering algorithm for regrouping entities based on a method for merging and splitting the related classes. From the perspective of the overall system, the result is considered to be the optimal class partition.
- A flexible strategy that allows developers to control the refactoring costs. The original class structure is treated as relatively reasonable suggestions made by developers, so if the clustering result is closer to the original design, developers well need to spend less on refactoring. In our study, a threshold is set to adjust the restructuring efforts based on the importance of the refactoring suggestions. In addition, we rank the refactoring suggestions based on the expected increases in the weighted modularity that they bring to the system. By refactoring at the expense of modifying and testing the codes, as well as using the suggestion order and threshold, developers can make a compromise between costs and benefits.
- A set of more general weights for different coupling relationships between the methods is used as the basis for redistributing the functionalities of classes. We consider the weighted summation of four types of coupling relationships between a method pair as their similarity, including their sharing attribute, invocation, semantic relevance, and functional coupling. A heuristic adjustment parameter scheme is proposed, which aims to obtain the coupling coefficients that apply to different software systems. We improve the parameter calibration algorithm proposed by Bavota et al. [14], [15]. Several related classes selected randomly from 50 well-designed systems based on GitHub[1] are merged to form the artificial god classes. Furthermore, to derive the values of coefficients, we repeat the operations for splitting god classes by clustering analysis, which yields a design where the results are close to the original partitions. Our experiment results confirmed that the refactoring accuracy based on the optimal coefficients obtained is stable across different software systems.
- A comprehensive comparison with previous studies from the perspective of clustering. We performed comparisons with previous studies bases on evaluation metrics containing cohesion and coupling measures, understandability, flexibility, reusability, and maintainability functions defined in the quality model for object-oriented design (QMOOD) and maintainability models.
- A further experimental evaluation was performed by professional software quality evaluators. According to questionnaires completed by 100 subjects and experimental data provided by Bavota et al. [15], we assessed the effectiveness of the proposed approach from the developer's perspective.
- A publicly-available tool[2, 3] and raw data to support further replication and research. We present "Refactoring solutions" (REsolution) as an automatic refactoring tool, which encapsulates the functions described above to allow developers to remove the code smells caused by cohesion and coupling problems.

The remainder of this paper is organized as follows. Section 2 discusses related research and Section 3 introduces the refactoring algorithm by using an example to illustrate the refactoring process. In Section 4, we discuss how to assign the four types of coefficients for edge weights in the method-level network. In Section 5, we provide a comparison with previous research. Section 6 presents a case study where five open source systems are used to perform automatic refactoring operations. Finally, we give our conclusion in Section 7.

2 Related Work

Three types of refactoring operations, i.e., move method, move field, and extract class refactorings, are used to re-assign the class responsibilities according to the principle of "high cohesion and low coupling." In this section, we review related research from three different areas.

2.1 Identification of Move Method/Field Refactoring Opportunities

To minimize the coupling and maximize the cohesion of a system, Czibula et al. proposed a class-level software refactoring scheme based on the *k-means* clustering algorithm [16]. They proposed the concept of a distance between the entity and class, where if the distance between the entity and the class to which it belongs is smaller than the distance between the entity and the other classes, then the entity should be considered as a move method/field refactoring candidate.

Bowman et al. combined a multi-objective genetic algorithm (MOGA) with complementary coupling and cohesion measurements to re-assign methods and attributes to classes in a class diagram [17]. MOGA performed far better compared with simpler alternative heuristics such as hill climbing and single objective GA, and the suboptimal moving methods/field refactoring suggestions could be

1. https://github.com/

2. Source code: https://github.com/wangying8052/REsolution

3. Runnable JAR: https://github.com/wangying8052/REsolution_runnable-JAR-File

corrected by the MOGA. Similar studies were performed by Lee et al. [18] and Seng et al. [19], who also used search techniques to generate a refactoring operation sequence, but the conflicts between the generated refactoring operations were not considered by Seng et al. [19]. Lee et al. [18] attempted to solve the refactoring conflict problem, but reordering the refactoring candidates after generating the sequence appeared to be time-consuming.

The aim of all search-based refactoring approaches is to identify the optimal refactoring sequence. Each refactoring step that is identified depends on the previous refactoring step. To identify multiple refactoring operations that can be applied simultaneously, Han et al. presented the concept of a maximal independent set (MIS) [20]. Based on the MIS, they automatically found the move method refactoring candidates that maximize the improvement in maintainability for software systems.

Tsantalis et al. presented the entity placement metric (EP) to evaluate the refactoring effect [21], which is defined as the ratio of the cohesion relative to coupling. If the *EP* value is lower, then the refactorings are more effective. This algorithm can obtain a set of envied classes where a method should be moved and then select that which benefits the system most from the perspective of the *EP* metric as the target class.

Recently, a novel approach called Methodbook was proposed by Bavota et al. for removing the feature envy code smells [22]. Unlike the refactoring algorithms mentioned above, semantic and structural measurements are both considered to evaluate the similarity between methods. Methodbook uses relational topic models to analyze the "friendships" of each method pair in the system, and the confidence level concept is employed to identify the target class where each method should be moved.

2.2 Identifying Extract Class Refactoring Opportunities

Fokaefs et al. proposed an extract class refactoring approach and a tool called JDeodorant based on the hierarchical clustering algorithm [23]. JDeodorant also integrates Tsantalis's approach [21] so it can perform both move method and extract class refactoring. The *Jaccard* distance is combined with structural measures to calculate the similarity of the entities in the god class, before merging the entities with the highest similarity, and thus the god class can be split into more than two new classes. The stop condition for hierarchical clustering requires that all the entities have been merged into one community and all the possible partitions of the original class are obtained by observing the tree diagram. Finally, it is necessary to rank all the refactoring opportunities according to the expected improvement they will obtain for the system. However, the best partition that assigns the most appropriate class responsibilities is selected manually by the developer. To verify the validity of this algorithm, semantic, structural, and combined measures were used to evaluate the accuracy of the refactorings. The results confirmed that hierarchical clustering combined with structural measures delivered superior performance.

Another interesting study by Bavota et al. used the *Max-Flow-MinCut* algorithm to split the god classes [14]. This semi-automatic approach based on graph theory combines structural and semantic measures to evaluate the cohesion between methods. The names of the variables, methods, and comments are extracted for use as a vocabulary to analyze the conceptual similarity of method pairs. A set of strongly related methods can be moved automatically from the original class into a new class by the proposed extract class refactoring approach, but the moved set does not include the attribute elements. Thus, developers need to regulate the distribution of attributes based on the method clusters obtained. The original class can only be split into two new classes with higher cohesion using this approach, but the god class may encapsulate more than two new functions.

To overcome this limitation, Bavota et al. proposed an improved algorithm for decomposing a class into two or more new classes [15] and the refactoring method was implemented in the Automated Refactoring In EclipSe (ARIES) project [24]. In addition, the evaluation is expanded significantly in the proposed approach [15] by combining quality metrics with empirical evaluations performed by the original and external developers to evaluate the refactoring results. The performance of the proposed algorithm depends on the coefficients for the different types of coupling relationship measurements used to calculate the similarity of methods.

The extract class refactoring approaches mentioned above are not focused on the identification of god classes, so the god classes need to be selected by the designers. In particular, in the method proposed by [23], each class in the system can be analyzed by the algorithm as well-designed preconditions, but the refactoring opportunities need to be selected by developers according to the suggestions generated. In addition, the inputs used by the clustering algorithms in the approaches described above are classes, and thus the entire system cannot be analyzed comprehensively. In our method, all the connected classes are merged into one entity set and we then obtain several new classes after regrouping them. Consequently, the refactoring suggestions obtained by our algorithm are considered the optimal results from the system-level perspective.

2.3 Identifying Multiple Refactoring Opportunities

O'Keeffe et al. used search-based techniques by combining an appropriate representation of the system structure with a change effecting operator and fitness function to solve refactoring problems automatically [25], [26], [27]. The search-based refactoring algorithm can perform multiple refactorings, such as push up/down field/method, extract hierarchy, and collapse hierarchy, and it was implemented as a tool called CODe-Imp. An empirical comparison was made between the refactoring results obtained by simulated annealing, genetic, and multiple ascent hill-climbing search-based algorithms. QMOOD [28] was used in their approach to evaluate the changes in code quality, such as reusability, understandability, and flexibility, which is also employed our study.

Moore et al. developed a prototype tool called Guru for automatic inheritance hierarchy restructuring and method refactoring for Self programs [29]. Guru uses a collection of classes but they do not need to be related by inheritance and they need not comprise a complete inheritance hierarchy. Furthermore, these classes are restructured into a new inheritance hierarchy without duplicate methods, so the code behaviors are preserved. In the restructuring process

for the inheritance hierarchy, multiple refactoring operations are performed, including the push up/down method and extract superclass/subclass.

Streckenbach and Snelting also considered the problems of pushing down or pulling up class members as well as splitting large classes [30], where they proposed the KABA tool based on the Snelting/Tip algorithm to restructure class hierarchies. Code behaviors are preserved by combining program analysis, type constraints, and concept lattices. Using KABA, a refactoring proposal is generated automatically based on the usage of the hierarchy by the client programs. Thus, the classes that are not invoked directly by the given client programs cannot be refactored.

Similar to our method, the algorithms mentioned above restructure the software system by moving class members or decomposing the classes. However, in previous approaches, the methods/fields are pulled up or pushed down based on their inheritance relationships, and the classes are extracted as the subclasses or superclasses of the original classes; thus, the average inheritance depth and number of immediate subclasses increase. Empirical evidence suggests that when a class is deeper within the hierarchy, then it is likely to inherit more methods, so predicting its behavior is more complex [31]. Furthermore, if the classes have more children, it is more difficult to modify them. A change affects all the subclasses, which usually requires more testing. Therefore, these classes are more fault-prone [32], [33]. In our method, we split the inheritance tree under the condition of controlling the inheritance depth and the number of subclasses, and the extracted trees are invoked by the original trees based on delegation. In Sections 5 and 6, we provide a comparison of related research and we discuss various refactoring results.

3 Methodology

The dependency relationships between classes belonging to inheritance hierarchies are complicated, so we need to set many preconditions in the clustering process to retain the code behaviors. For example, if the super-classes are decomposed, then the effects on their subclasses will not be predictable. The proposed algorithm includes two steps: the refactoring operations are applied automatically to the classes not in inheritance hierarchies and the leaf nodes of the inheritance hierarchies; and we then remove the code smells in the classes belonging to the inheritance hierarchies.

3.1 Class-Level Refactoring Preprocessing

When viewed from various levels, software comprises a set of elements, such as classes and methods. The system functions are implemented via the interactions among the elements. If we consider the classes (or methods) as nodes and the dependency relationships between them as edges, then the software system can naturally be described as a complex network [34], which is denoted as $G = (V, E)$, where V represents the node set and E represents the edge set. Let $|V|$ and $|E|$ be the total number of nodes and edges, respectively. The types of dependency relationships between classes include inheritance and association, whereas those between methods include invocation and accessing attributes. Edges can be directed or undirected. Moreover, the edges can be assigned weights to represent the strength of the relationships.

If we consider classes as the nodes of G, then G is the class-level network, where $V = \{CL_1, CL_2, \ldots, CL_{|V|}\}$, $E = \{e_1, e_2, \ldots, e_{|E|}\}$, $CL_i \in V$ is any class in the system, and $e_j \in E$ is any relationship between classes, where $i \in \{1, 2, \ldots, |V|\}$, $j \in \{1, 2, \ldots, |E|\}$.

If we consider methods as the nodes of G, then G is the method-level network, where $V = \{m_1, m_2, \ldots, m_{|V|}\}$, $E = \{e_1, e_2, \ldots, e_{|E|}\}$, $m_i \in V$ is any method in the class, and $e_j \in E$ is any relationship between methods, where $i \in \{1, 2, \ldots, |V|\}$, $j \in \{1, 2, \ldots, |E|\}$.

3.1.1 Class-Level Multi-Relation Directed Network (CMDN)

To distinguish the classes belonging to the inheritance and non-inheritance hierarchies, we propose the concept of a class-level multi-relation directed network.

Definition 1 (CMDN). *Suppose that $G = (V, E)$ is a class-level directed network, where E is a set of dependency relationships and the dependencies can be divided into n types, such as inheritance, association, and aggregation, then it follows that $E_1 \bigcup E_2 \bigcup \cdots \bigcup E_n = E$, and $E_1 \bigcap E_2 \bigcap \cdots \bigcap E_n = \varnothing$. If $\exists \forall E_i \in E$, we have $G_1 = (V, E_1)$, $G_2 = (V, E_2)$, . . . , $G_n = (V, E_n)$, and thus G is referred to as the CMDN. The properties of a CMDN are described as follows.*

Property 1. *Suppose that E_t is a type of dependency relationship set between nodes. If class $CL_a, CL_b \in V$, and class CL_a depends on class CL_b, which is denoted by $CL_a \xrightarrow{e_{ab}} CL_b$, then an edge $e_{ab} \in E_t$ exists from CL_a to CL_b.*

Property 2. *If we only consider the inheritance and non-inheritance relationships, then the CMDN can be viewed as comprising the directed networks G_1 and G_2. The CMDN is represented as $G = G_1 \bigcup G_2$, where $G_1 = (V, E_1)$, $G_2 = (V, E_2)$, E_1 is the dependency relationship set except inheritance, and E_2 is the inheritance relationship set. The nodes of G_1 correspond one-to-one with the nodes of G_2.*

Consider a system comprising six classes, where class $DrawingPanel$ is the superclass of $PerPanel$ and $SVGPanel$, class $NetPanel$ inherits $PerPanel$, $UndoRedoManager$ and $RestoreDataEdit$ are the classes of non-inheritance hierarchies, and class $UndoRedoManager$ is depended upon by class $NetPanel$ and $PerPanel$. The corresponding CMDN for this example is shown in Fig. 1.

Property 3. *Any class CL_i of the system can be considered as an entity set defined in the class itself. The entity set contains two types of data: methods and attributes. If $\mathcal{M}_i = \{m_1, m_2, \ldots, m_n\}$, and $\mathcal{A}_i = \{a_1, a_2, \ldots, a_p\}$, where $m_k \in \mathcal{M}_i$ represents any method defined in class CL_i and $a_t \in \mathcal{A}_i$ denotes any attribute defined in class CL_i, then it follows that $CL_i = \mathcal{M}_i \bigcup \mathcal{A}_i$.*

3.1.2 Refactoring Preprocessing Operations for the Non-Inheritance Hierarchies

To remove the code smells caused by cohesion and coupling problems to the greatest extent, we merge the entities of each class component in the non-inheritance hierarchies and the leaf nodes in the inheritance hierarchies into an

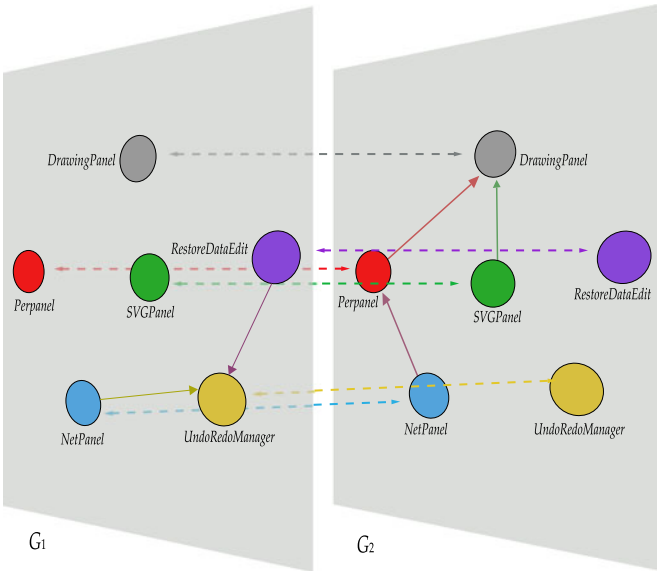


Fig. 1. Illustration of multi-relation directed networks.

entity set, and then regroup them. In the inheritance hierarchies, if the non-leaf nodes are decomposed in the clustering process, then the code behaviors will be changed. For example, compile errors will be caused by movements of the entities inherited by the corresponding subclasses and the methods that override the concrete methods or those containing any super-method invocations. The leaf nodes have no subclasses, so their entities can be moved out of the inheritance hierarchies according to the principle of "high cohesion and low coupling" and various appropriate preconditions.

Based on the definition given above and the properties of the CMDN, the refactoring preprocessing operations can be described as follows.

(1) Construct network $G = G_1 \bigcup G_2$ by analyzing the abstract syntax tree of the source code, where $G_1 = (V, E_1)$, $G_2 = (V, E_2)$, G_1 is the non-inheritance relationship network, and G_2 is the inheritance relationship network.

(2) Traverse the nodes in set V of G_2. If the in-degree of node $CL_i \in V$ is greater than 0, then CL_i is a non-leaf node in the inheritance hierarchies. If we let V_D be the node set to be deleted, then it follows that CL_i will be added to V_D.

(3) After deleting all the nodes that belong to V_D and the edges connected to them, we obtain the residual network denoted as $\overline{G_1} = (\overline{V}, \overline{E_1})$, where $\overline{V} = V \backslash V_D$, and $\overline{E_1} = E_1 \backslash E_D$; E_D is the edge set connected to the nodes in V_D.

(4) All of the connected components will be found in the residual network $\overline{G_1}$. Let CC represent the connected component set in $\overline{G_1}$, and $CC = \{cc_1, cc_2, \ldots, cc_{|CC|}\}$, where $|CC|$ is the total number of connected components in $\overline{G_1}$.

(5) If we merge all the classes in each connected component into an entity set Ω_i, for all $k \in [1, 2, \ldots, |CC|]$ and $m \in [1, 2, \ldots, |cc_k|]$, if $CL_m \in cc_k$, then we have $\Omega_i = \bigcup CL_m$, where $|cc_k|$ represents the number of classes in the connected component cc_k.

After the refactoring preprocessing, all the superclasses in the inheritance hierarchies are filtered out from the system. In this manner, the system can be denoted as several entity sets $\Psi = \{\Omega_1, \Omega_2, \ldots, \Omega_{|CC|}\}$. Moreover, all the entity sets in Ψ are treated as the objects to be restructured in the first step.

3.1.3 Refactoring Preprocessing Operations for the Inheritance Hierarchies

The number of classes and the relationships between them will actually change after regrouping the entities of the non-inheritance hierarchies. Therefore, before performing the refactoring operations for the inheritance hierarchies, we need to update the CMDN $G' = G_1' \bigcup G_2'$ based on the system structure obtained from the first refactoring step.

Let TR be the inheritance tree set and $TR = \{Tr_1, Tr_2, \ldots, Tr_{|TR|}\}$, where $|TR|$ is the total number of inheritance trees. In the inheritance relationship network G_2', we define the class with an out-degree of 0 and an in-degree greater than 0 as the root node in the inheritance tree, which is denoted as CL_k^{root}, $k \in [1, 2, \ldots, |TR|]$. If there is a directed path from node CL_m to CL_n, then we say that node CL_m can reach CL_n. The nodes in G_2' that can reach the root node CL_k^{root} comprise an inheritance tree Tr_k. Obviously, one root node CL_k^{root} corresponds to an inheritance tree Tr_k.

We need to remove the bad smells from each inheritance tree in turn. To preserve the code behaviors, the classes in the inheritance hierarchies should not be regrouped after being merged into an entity set. Hence, we decompose the classes from top to bottom according to the structure of the inheritance tree. The refactoring preprocessing operations for the inheritance hierarchies are as follows.

(1) Update CMDN $G' = G_1' \bigcup G_2'$ according to the refactoring results for the non-inheritance hierarchies, where $G_1' = (V', E_1')$, and $G_2' = (V', E_2')$. In network G_2', if one node does not belong to the inheritance hierarchies, then no edges are connected to it. We traverse the classes in set V of G_2, where all the interfaces comprising empty methods and the nodes with a degree equal to 0 will be added to the node set V_D to be deleted.

(2) After deleting all the nodes in V_D, we obtain a residual network represented as $\overline{G_2'} = (\overline{V'}, \overline{E_2'})$, where $\overline{V'} = V' \backslash V_D$, and $\overline{E_2'} = \overline{E_2} \backslash E_D$, and E_D is the edge set connected to the nodes in V_D.

(3) We find all the connected components in $\overline{G_2'}$ and, clearly, each of them denotes an inheritance tree.

3.2 Application of the Weighted Clustering Algorithm to Regroup Methods

3.2.1 Method-Level Weighted Undirected Network

Following the preprocessing operations described above, the refactoring objects can be decomposed into several entity sets and inheritance trees. Each entity set $\Omega_i \in \Psi$ or any class in the inheritance tree $Tr_k \in TR$, and the relationships between its elements can be described as a method-level weighted undirected network. To consider the attributes during the refactoring process, we treat all the attributes as their corresponding *Getter* or *Setter* methods. Thus, we equate the accessing of an attribute with the invocation of its corresponding *Getter* or *Setter* method.

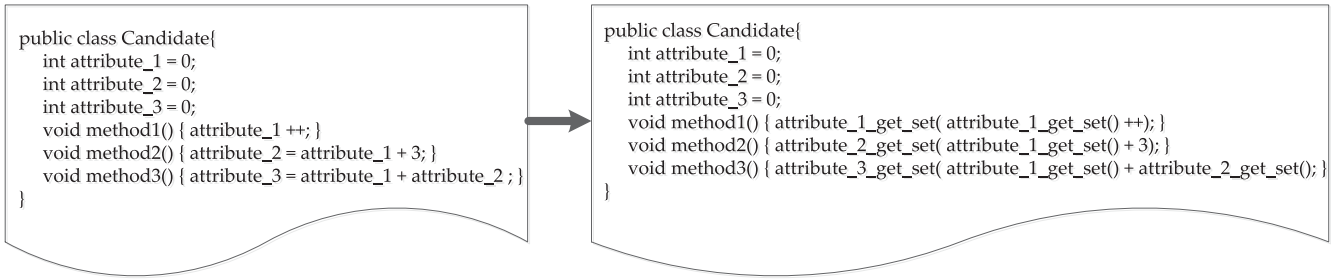


Fig. 2. Illustration of the conversion of attributes into their corresponding $Getter$ or $Setter$ methods.

As shown in Fig. 2, the methods $attribute\_1\_get\_set()$, $attribute\_2\_get\_set()$ and $attribute\_3\_get\_set()$, which are added by the proposed approach virtually, are the same as the $Getter$ or $Setter$ method functions for attributes $attribute\_1$, $attribute\_2$ and $attribute\_3$, respectively. If we consider the method $a_{k\_}\ get\_set()$ as the node of the network instead of its corresponding attribute $a_k \in CL_i$, then it follows that after the clustering analysis, the cluster that contains the method $a_{k\_}\ get\_set()$ can be seen as the new class to which the corresponding attribute a_k belongs. The advantages of the proposed method described above are as follows.

(1) The two types of nodes in the system can be converted into one type of node.
(2) All the methods that access attribute a_k have a sharing attribute, but also an invocation relationship with the $Getter$ or $Setter$ method for a_k. In this manner, the relationships between method pairs that share the same attributes are assigned a high weight, and thus it follows that the related methods and attributes tend to be assigned to the same cluster. Therefore, the attributes largely avoid being accessed by the methods defined in the other classes, which balances the tradeoff between improving the cohesion of a class and increasing the field security. Increasing the value of attribute invisibility would imply less complexity as well as greater understandability and a higher degree of maintainability, thereby improving the quality of the software system [35], [36].

Three types of relationships between methods, i.e., attribute sharing, invocation, and functional coupling, are considered when we construct a method-level undirected network. Moreover, we assign weights to all the edges according to the strengths of the relationships.

If attribute a_t is accessed by both methods m_i and m_j, then a sharing attribute relationship exists between m_i and m_j. The sharing attribute weight (SAW) of m_i and m_j is calculated by

$$SAW(m_i, m_j) = \begin{cases} \frac{|\mathcal{M}_i \bigcap \mathcal{M}_j|}{|\mathcal{M}_i \bigcup \mathcal{M}_j|} & \text{if } |\mathcal{M}_i \bigcup \mathcal{M}_j| \neq 0 \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

where $\mathcal{M}_i$ and $\mathcal{M}_j$ represent the attribute sets accessed by m_i and m_j, respectively.

Let $I(m_i, m_j)$ be the number of invocations performed by method m_i on m_j, and n is the total number of methods in the system. As shown by Eq. (2), if m_j is invoked by the other methods, then MIW_{ij} is the ratio of $I(m_i, m_j)$ relative to the total number of times that m_j is called; otherwise, MIW_{ij} is equal to 0. Furthermore, the method invocation weight (MIW) between m_i and m_j is denoted as the maximum value of MIW_{ij} and MIW_{ji}

$$MIW_{ij} = \begin{cases} \frac{I(m_i, m_j)}{\sum_{k=1}^{n} I(m_k, m_j)} & \text{if } \sum_{k=1}^{n} I(m_k, m_j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$MIW(m_i, m_j) = \max(MIW_{ij}, MIW_{ij}). \quad (3)$$

Typically, the refactored classes are optimal in the sense that every class object contains only the methods or fields that it actually accesses [30]. From this perspective, if more entities defined in a class are accessed by the instance object to which it belongs, then there is greater improvement in the functional cohesion. In our approach, each method is considered as a functional domain. If methods m_i and m_j are invoked by the same functional domain, then we say that m_i and m_j have functional coupling relationship. Consequently, methods that frequently appear in the same functional domain have tight functional coupling relationships with each other and a high probability of being assigned to the same class. The functional coupling weight (FCW) is computed by

$$FCW(m_i, m_j) = \begin{cases} \frac{ET_{ij}}{ET_i + ET_j} & \text{if } ET_i + ET_j \neq 0 \\ 0 & \text{otherwise,} \end{cases} \quad (4)$$

where ET_{ij} represents the number of functional domains in which methods m_i and m_j appear together, and ET_i and ET_j denote the numbers of functional domains by which m_i and m_j are invoked, respectively.

Similar to [12], we use the latent semantic indexing (LSI) technique to define the semantic similarity weight (SSW) between method pair [14]. The source code under analysis is converted into a text corpus, so we can extract the variable names, code comments, etc. from each method. In this corpus, each method is considered as a document and using LSI, we map each document onto a vector in a multi-dimensional space determined by the terms [37]. Thus, it follows that methods m_i and m_j can be represented as the vectors $\overrightarrow{m_i}$ and $\overrightarrow{m_j}$, respectively. If we let $||\overrightarrow{m_k}||$ be the Euclidean norm of vector $\overrightarrow{m_k}$, then $SSW(m_i, m_j)$ is defined as the cosine of the angle between the vectors corresponding to methods m_i and m_j

$$SSW(m_i, m_j) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{||\overrightarrow{m_i}|| \cdot ||\overrightarrow{m_j}||}. \quad (5)$$

```
public class DrawingPanel {

    private static double scaleFactor;
    private static Point2D.Double translate;
    private static JLabel emptyDrawingLabel;
    private static Constrainer constrainer;
    static JToolBar bar;

    private void addCreationButtonsTo(...) { ... }
    public void addDefaultCreationButtonsTo(...) { ... }
    public void addAttributesButtonsTo(...) { ... }
    public void addZoomButtonsTo(...) { ... }
    public AbstractButton createZoomButton(...) { ... }
    public void setEmptyDrawingMessage(...) { ... }
    public String getEmptyDrawingMessage() { ... }
    protected void drawDrawing(...) { ... }
    public Point2D.Double viewToDrawing(...) { ... }
    public Rectangle drawingToView(...){ ... }

    }

public class UndoRedoManager {

    private UndoAction undoAction;
    private RedoAction redoAction;
    private ResourceBundleUtil labelsUndo;
    private ResourceBundleUtil labelsRedo;
    public boolean undoOrRedoInProgress;

    public ResourceBundleUtil getlabelsRedo() { ... }
    public ResourceBundleUtil getlabelsUndo() { ... }
    public Action getUndoAction()  { ... }
    public Action getRedoAction() { ... }
    public void undoProgress(...)  { ... }
    public void redoProgress(...)  { ... }
    public boolean undoOrRedo(...)  { ... }
    public void UndoactionPerformed(...)  { ... }
    public void RedoactionPerformed(...)  { ... }
    public void updateActions(...)  { ... }

}

public class SVGPanel extends DrawingPanel {

    static double scaleFactorSVG;
    static Point2D.Double translateSVG;
    AffineTransform t;

    public Point2D.Double viewToDrawing(...) { ... }
    public static Rectangle drawingToView(...) { ... }
    public AffineTransform getDrawingToViewTransform() { ... }
    protected void setHasUnsavedChanges(...) { ... }

    }

public class PertPanel extends DrawingPanel {

    static JPopupButton createPopupButton;
    static JPopupButton FontButton;
    static ResourceBundleUtil labels;
    static ResourceBundleUtil labelsDrawing;
    double scaleFactorPert;
    Point2D.Double translatePert;

    private static void addCreationButtonsTo(...) { ... }
    public void addDefaultCreationButtonsTo(...) { ... }
    public static void addFontButtonsTo(...) { ... }
    public static void addColorButtonTo(...) { ... }
    public void setEmptyDrawingMessage(...) { ... }
    public String getEmptyDrawingMessage() { ... }
    protected void drawBackground(...) { ... }
    protected void drawGrid(...) { ... }

    }

public class NetPanel extends PertPanel {

    static JPopupButton strokeDecorationPopupButton;
    static JPopupButton strokeWidthPopupButton;
    static JToolBar UndoDrawing;

    public void NetPanel(...) { ... }
    private void addCreationButtonsTo(...) { ... }
    public void addDefaultCreationButtonsTo(...) { ... }
    public void addStrokeDecorationButtonTo(...) { ... }
    public void addStrokePlacementButtonTo(...) { ... }
    public void undoDrawing(...) { ... }
    public static void undoData(...) { ... }

    }

public class RestoreDataEdit {

    Figure figure;
    Object oldRestoreData;
    Object newRestoreData;

    public RestoreDataEdit(...) { ... }
    public String getPresentationName() { ... }
    public boolean addEdit(...) { ... }
    public boolean replaceEdit(...) { ... }
    public void redoEdit(...) { ... }
    public void redoData(...) { ... }

    }
```

Fig. 3. Code snippet used as an example to understand the proposed algorithm.

Thus, $SSW(m_i, m_j)$ depends on the word usage similarity in different code snippets. If methods m_i and m_j are conceptually related, then the value of $SSW(m_i, m_j)$ is greater than 0. However, [14] indicated that nearly all method pairs have semantic relationships with each other, so the method-level semantic network can be considered as a complete graph. We use the threshold Th_1 to remove the pseudo-semantic relationships with weights lower than Th_1.

The edge weight of the proposed method-level network model is described by Eq. (6), where $\alpha + \beta + \gamma + \eta = 1$. Given that $SAW(m_i, m_j)$, $MIW(m_i, m_j)$, $FCW(m_i, m_j)$, $SSW(m_i, m_j) \in [0, 1]$, we have $W_e(m_i, m_j) \in [0, 1]$

$$W_e(m_i, m_j) = \alpha \times SAW(m_i, m_j) + \beta \times MIW(m_i, m_j) + \gamma \times FCW(m_i, m_j) + \eta \times SSW(m_i, m_j). \quad (6)$$

An example code[4] is shown in Fig. 3. Clearly, according to the refactoring preprocessing operations described in

4. https://github.com/wangying8052/Example-Code

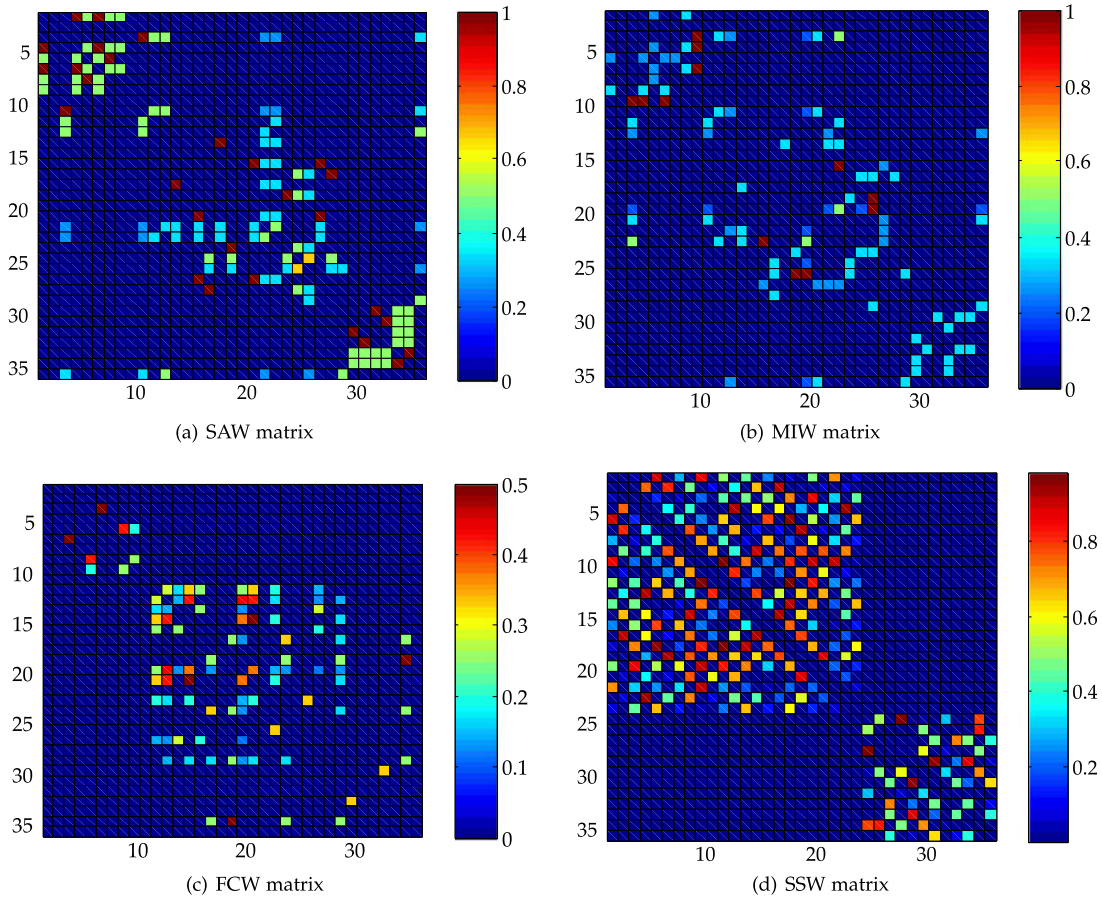


Fig. 4. Four types of weighted coupling matrices for the method-level undirected network.

Section 3.1.2, after merging the classes of the non-inheritance hierarchies and the leaf nodes in the inheritance hierarchies, we can obtain the residual network $\overline{G_1}$ by deleting superclasses $DrawingPanel$ and $PertPanel$. Furthermore, the connected components cc_1 and cc_2 can be found in $\overline{G_1}$, where $cc_1 = \{RestoreDataEdit, NetPanel, UndoRedoManager\}$, and $cc_2 = \{SVGPanel\}$. Obviously, there are 10 attributes and 23 methods in entity set Ω_1, which is obtained by merging the three classes of cc_1. For example, if we consider component cc_1, we convert each attribute a_i into the method $a_i\_get\_set()$ as proposed in our approach. Figs. 4a, 4b, 4c, and 4d show the four types of weighted coupling matrix for the method-level undirected network. Obviously, more and higher coupling relationships can be captured by semantic measurement. Lots of semantic similarities can be identified between the method pairs which even have no types of structural coupling relationships. This can be considered as a reason why we need to set the threshold Th_1 to eliminate the pseudo-semantic relationships.

3.2.2 Weighted Clustering Algorithm

To improve the class structure, the hierarchical clustering algorithm employed to regroup the entities needs to satisfy the following conditions.

(1) There is no need to assign the number of clusters. The goal of extract class refactoring is to split the god class according to the "high cohesion and low coupling" principle, but the number of new classes into which the god class should be decomposed cannot be specified.

(2) There is no need to set thresholds for the stopping condition for cluster partitioning. The structure and behaviors are difficult to control because of the complexity of the software, and thus it follows that it is difficult to set thresholds that are applicable to all software systems.

We use the community detection algorithm proposed by Clauset, Newman, and Moore to regroup the entities [38], which is a type of agglomerative method that uses the modularity to measure the quality of division. The modularity represents the difference between the divided network and its corresponding null model [39], [40]. For unweighted undirected networks, the modularity metric is calculated by

$$Q' = \frac{1}{2M}\sum_{ij}\left(a_{ij} - \frac{k_i \times k_j}{2M}\right) \times \delta(CN_i, CN_j), \tag{7}$$

where M is the number of the edges in the network, $A = (z_{ij})$ is the adjacency matrix of the network, k_i and k_j are the degrees of nodes nd_i and nd_j, respectively, and CN_i and CN_j are the communities to which nodes nd_i and nd_j belong, respectively. If nodes nd_i and nd_j belong to the same community, then $\delta(CN_i, CN_j)$ is 1; otherwise, $\delta(CN_i, CN_j)$ is equal to 0. We propose a weighted modularity Q metric to evaluate the quality of the partition, which is defined as

Algorithm 1. The weighted community detection algorithm

Input: Sharing attribute adjacency matrix $A_{n\times n}$;
Method invocation adjacency matrix $B_{n\times n}$;
Simultaneous execution adjacency matrix $C_{n\times n}$;
Semantic relevance adjacency matrix $D_{n\times n}$.
Output: Community set C; Weighted modularity Q.

1. Initially, we obtain the adjacency matrix A' of the method-level network $G = (V, E)$ based on the equation $A' = \alpha \times A_{n\times n} + \beta \times B_{n\times n} + \gamma \times C_{n\times n} + \eta \times D_{n\times n}$. The methods that should not be split are bound into one community, and each of the other nodes $nd_i \in V$ is considered as a community. We assume that network G is divided into $|C|$ communities, and then the community set can be described as $C = \{CN_1, CN_2, \ldots, CN_{|CN|}\}$. Let the weighted modularity $Q = 0$.
2. According to Equation (9), the element ΔQ_{ij} of the initial modularity increment matrix A_Q is calculated as follows:

```
for ( i = 1; i<= |C|; i ++){
  for ( j = 1; j <= |C|; j++){
    if A'ij ≠ 0
      ΔQij = (Sin ( CNi, CNj) - Si × Sj)/2W;
    else
      ΔQij = 0;                    }
                                  }
```

3. We find the maximum element ΔQ_{mn} in matrix A_Q, without considering the modularity increments whose corresponding community pair contains the entities defined in different original classes. Merge communities CN_m and CN_n, and denote the merged community as CN_n. After deleting the elements in row m and column m, the elements in row n and column n are updated as follows:

$$\Delta Q_{nk}' = \Delta Q_{kn}' = \begin{cases} \Delta Q_{mk} + \Delta Q_{nk}, & \text{if } CN_k \text{ connects with } CN_m \text{ and } CN_n \\ \Delta Q_{mk} - S_n \times S_k / 2W^2, & \text{if } CN_k \text{ connects with } CN_m \text{ and not connects with } CN_n \\ \Delta Q_{nk} - S_m \times S_k / 2W^2, & \text{if } CN_k \text{ connects with } CN_n \text{ and not connects with } CN_m \end{cases}$$

At the same time, we update the weighted modularity by the equation $Q = Q + \Delta Q_{mn}$. Repeat the process until the maximum modularity increment of merging the communities whose entities belong to the same original class is lower than 0.
4. Continue to congregate the community pair containing the entities of different classes whose corresponding ΔQ_{mn} is the greatest one in matrix A_Q. Update the elements of A_Q in the way of step (3). When the maximum element of A_Q is less than or equal to 0, we stop clustering.

Fig. 5. Description of the weighted clustering algorithm.

$$\begin{aligned} Q &= \frac{1}{2W}\sum_{ij}\left(w_{ij} - \frac{s_i \times s_j}{2W}\right) \times \delta(CN_i, CN_j) \\ &= \sum_{k=1}^{nc}\left[\frac{W_k}{W} - \left(\frac{S_k}{2W}\right)^2\right], \end{aligned} \tag{8}$$

where W is the sum of all the edge weights in the network, s_i is the sum of all the weights of the edges connected to node nd_i, w_{ij} is the weight of the edge between nodes nd_i and nd_j, n_c is the number of communities in the network, W_k is the sum of all the edge weights within community CN_k, and S_k is the sum of all the weights of the edges connected to the nodes in community CN_k. Based on Eq. (8), the weighted modularity increment for the overall system after merging communities CN_i and CN_j is given by

$$\Delta Q_{ij} = \begin{cases} \frac{S_{\text{in}}(CN_i, CN_j) - S_i \times S_j}{2W} & \text{if } CN_i \text{ connects with } CN_j \\ 0 & \text{otherwise,} \end{cases} \tag{9}$$

where $S_{\text{in}}(CN_i, CN_j)$ is the sum of all the weights of the edges connecting communities CN_i and CN_j.

We use the modularity increment matrix $A_Q = (\Delta Q_{ij})$ to calculate and update the modularity increment before merging each community pair in the network. To preserve the code behaviors, the methods that should not be split according to the syntax rules and preconditions are bound together into one community at the beginning of community detection, where each of the other nodes in the network is considered an independent community. Furthermore, we calculate each element ΔQ_{ij} of the modularity increment matrix A_Q. If ΔQ_{mn} is the maximum element in A_Q, then communities CN_m and CN_n are merged, and A_Q is updated at the same time. In this manner, we agglomerate the communities step by step until the maximum element of A_Q is 0, and when this occurs, the community structure is the optimal division and the modularity of the system reaches its peak. The detailed community detection algorithm is described in Fig. 5. Obviously, move method/field refactoring occurs when two communities containing entities from the different original classes are merged. After clustering ends, the extract class refactoring opportunity is identified if the methods defined in the same class are distributed in more than one community. Refactoring is a type of test-driven operation, so there is a cost due to the workloads and a risk of modifying the source code. Considering the practical value of this approach, we respect the original system structure and avoid big-bang software refactoring. Therefore, we improve the clustering algorithm as follows:

(1) We prioritize merging the community pair with entities that are all defined in the same original class, thereby maintaining the original design as much as possible. The groups obtained are considered to be the reasonable functionality distributions suggested by the original developers.

(2) Next, if the modularity increment obtained by aggregating the community pair containing the entities from different classes satisfies $0 \leq \Delta Q_{mn} \leq Th_2$, then we merge CN_m and CN_n, where Th_2 represents the threshold used to identify more important move method/field opportunities. In the GUI of the proposed tool, three values can be assigned to Th_2 to allow developers to control the restructuring costs, i.e., 0, avg_0, and max_0, where avg_0 and max_0 denote the average and the maximum of the modularity increment obtained by merging the clusters containing the entities that belong to the same original class, respectively. In other words, avg_0 and max_0 are considered as two types of coupling strengths designed by original developers for the original system structure. If Th_2 is higher, then the clustering division is closer to the original design, and thus the refactoring expense is reduced.

3.2.3 Identification of the Extract Class and Move Method/Field Refactoring Opportunities

The move method, move field, and extract class refactoring opportunities can be identified simultaneously by analyzing the differences between the cluster partition and the original class structure. To limit the execution time for the algorithm, we narrow the scope of the comparisons to one connected component in the residual network $\overline{G_1}$ rather than the entire system. Thus, we compare each community obtained with each original class in the same connected component. If we let C_t be the community set obtained by clustering analysis and V_t is the original class set, where they satisfy $C_t, V_t \subseteq cc_t$, and $t \in [1, 2, \ldots, |CC|]$, then for all $m, i, j \in [1, 2, \ldots, |V_t|]$, $n, p, q \in [1, 2, \ldots, |C_t|]$, $i \neq j$, and $p \neq q$, they satisfy $V_t = \bigcup CL_m$, $C_t = \bigcup CN_n$, $CL_i \bigcap CL_j = \varnothing$, and $CN_p \bigcap CN_q = \varnothing$, where $|V_t|$ and $|C_t|$ are the total number of original classes and new classes for component cc_x, respectively. We can make the following conclusions.

(1) If $CN_k \subset CL_p$, $CN_k \bigcap CL_q = \varnothing$ for all $p, q \in [1, 2, \ldots, |V_t|]$, $p \neq q, k \in [1, 2, \ldots, N_p]$, and $N_p \geq 2$, then the original class CL_p is suggested as an opportunity for the extract class operation. Clearly, CN_k is the new class extracted from CL_p and N_p is the number of groups into which the original class CL_p is decomposed.

(2) If $CL_p \subset CN_k$, $CN_k \bigcap CL_q = T \neq \varnothing$ for all $p, q \in [1, 2, \ldots, |V_t|]$, $p \neq q, k = 1$, then the move method/field refactoring candidates are identified in the original class CL_q. It is suggested that the entity set T is moved from class CL_q to CL_p.

(3) Suppose that $CL_p = \bigcup_{i=1}^{N_p} CL_{pi}$, $CL_q = \bigcup_{j=1}^{N_q} CL_{qj}$. If $\exists CN_k = CL_{pi} \bigcup CL_{qi}$, then

① if $\left|CN_k \bigcap CL_p\right| > \left|CN_k \bigcap CL_q\right|$, then the original class CL_p should undergo the extract class refactoring operation and it is suggested that the entity set $CN_k \bigcap CL_p$ is extracted from CL_p to form a new class CL_p^{new}; furthermore, the move method/field refactorings are applied for class CL_q and it is suggested that the entity set $CN_k \bigcap CL_q$ is moved from class CL_q to CL_p^{new};

② if $\left|CN_k \bigcap CL_p\right| < \left|CN_k \bigcap CL_q\right|$, then the move method/ field refactoring operations are performed on the original class CL_p and obviously, the entity set $CN_k \bigcap CL_p$ should be moved from CL_p to CL_q.

It should be noted that the classes do not have to be split by the proposed clustering algorithm. If a class has a good design and satisfies the principle of "high cohesion and low coupling," then it will not be restructured after the clustering analysis. Finally, we sort all the refactoring suggestions based on the expected modularity increments that they bring to the system in descending order, which helps developers to make an appropriate decision to obtain more benefits with lower costs.

3.3 Automatic Refactoring Algorithm Framework

3.3.1 Preconditions for Fully Automated Refactoring

The preconditions for automatic refactoring can effectively ensure that the external behaviors of the code will not be changed [21], [23], [41]. The proposed approach is designed for automatically restructuring the entire system, so we set the the following preconditions besides those put forward in Refs. [23] and [21].

(1) All the methods that are synchronized or contain synchronized blocks, as well as the attributes accessed by them, should not be decomposed, so we bind them into a community at the beginning of the clustering analysis. In the synchronization mechanism for Java, if a synchronized method defined in an object is executed by one thread, then all the other threads that invoke synchronized methods for the same object will be suspended until the executed thread has finished its tasks. Thus, concurrent problems will be caused if the synchronized methods and the attributes accessed by them are split.

(2) If a method overrides an actual or abstract method for the superclass CL_i or contains any super-method or attribute invocations, then it should not be moved out of the subclass of CL_i because the behaviors of the original class and the classes that invoke the moved method will be changed if the method that overrides an actual inherited method is moved out of the subclass of CL_i. Similarly, if a method that overrides an abstract method is moved, this would cause compilation errors because the abstract method cannot be implemented. During the refactoring steps for the non-inheritance hierarchies, all the classes of the non-inheritance hierarchies and the leaf nodes in the inheritance hierarchies are merged into an entity set for regrouping by clustering analysis. To avoid the problems mentioned above, the methods that override the methods for their corresponding superclasses or that invoke super-methods are bound into

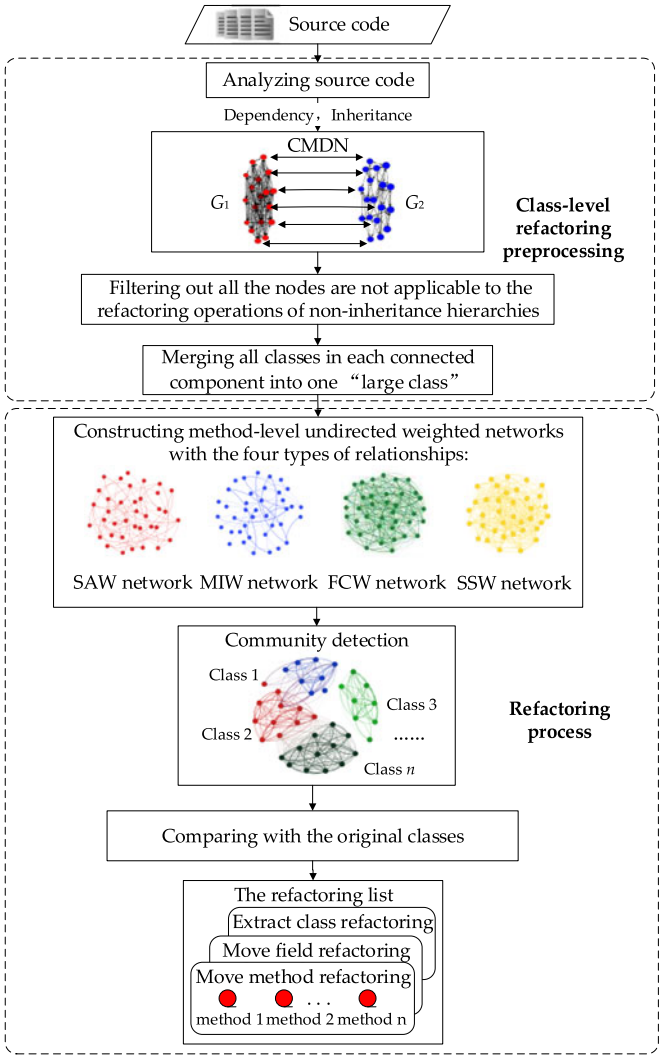


Fig. 6. Refactoring process for the non-inheritance hierarchies.

a community at the beginning of the clustering analysis. Finally, the cluster that contains these bound methods will inherit the corresponding original superclass.

(3) To preserve the code behaviors, if superclass CL_i is decomposed into several new classes based on the principle of "high cohesion and low coupling," then its subclasses should be split based on the structure of the new classes. The decomposition of the subclass should satisfy a requirement where methods that override or invoke the super-methods of the new class CL_k extracted from CL_i should be bound into the class that inherits CL_k.

(4) The interfaces only contain abstract methods, so they cannot be considered as target classes for the actual methods that need to be moved. Thus, when we preprocess the inheritance hierarchies, we filter out all the interfaces from the inheritance network.

(5) If the total number of methods in the new class extracted from the original class is less than NL, then we consider that the new class has too few functions and it follows that all of the extracted methods should be moved back. In our method, we let $NL = 3$.

3.3.2 Refactoring Process for the Non-Inheritance Hierarchies

We implement the refactoring preprocessing operations based on the CMDN, where all the classes of the non-inheritance hierarchies and the leaf nodes in the inheritance hierarchies are merged into several entity sets according to the network structure, and each entity set can be considered as a method-level weighted undirected network. Furthermore, the clustering algorithm described in Section 3.2 is used to regroup the entities and each cluster represents a new class. Specifically, each indecomposable method set needs to be merged into a cluster at the beginning of the clustering analysis. According to the preconditions, there are two types of indecomposable method set: 1) the methods defined in a class that are synchronized or contain the synchronized blocks, as well as the attributes accessed by them; and 2) the methods defined in a leaf node of the inheritance hierarchies that override or invoke the super-methods. After the clustering analysis, the new class that contains the indecomposable methods of the leaf node should inherit the corresponding original superclass. Finally, the move method, move field, and extract class refactoring suggestions are obtained by comparing the new classes with the original classes. The overall process of the class-level automatic refactoring algorithm is described in Fig. 6.

3.3.3 Refactoring Process for the Inheritance Hierarchies

If there are too many functions in an inheritance hierarchy, then the inheritance hierarchy needs to be teased apart by creating more inheritance hierarchies according to the principle of "high cohesion and low coupling," and one inheritance hierarchy can invoke the others by delegation [2].

The relationships are more complicated in the inheritance hierarchy than the non-inheritance hierarchy. If we merge all the classes into an entity set and then regroup them, it would be too difficult to preserve the code behaviors. To solve this problem, the clustering algorithm is used to regroup each class in the inheritance tree. Using refactoring preprocessing, all of the non-inheritance classes are filtered out of the network G_2. Thus, we traverse the inheritance trees in turn and each inheritance tree is decomposed from the top down. If new classes are extracted from the original superclass after the clustering analysis, then we need to split the corresponding subclasses based on the structure of the decomposed superclass to improve its cohesion. The refactoring algorithm is complete when all the leaf nodes have been processed. The detailed refactoring process is described in Figs. 7 and 8, where V_{root} represents the root node set of the inheritance hierarchies and $CL_k^{\text{root}} \in V_{\text{root}}$ represents any root node of an inheritance tree.

If root node CL_k^{root} is decomposed into class set V_k^{new}, then its direct subclass $CL_{km}^{\text{sub}} \in V_k^{\text{sub}}$ should be split according to the following operations, which are the further explanations of step 3.

(1) Create the new subclass set $V_k^{\text{sub}'}$ that corresponds to each new class extracted from the root node CL_k^{root}, and it follows that $|V_k^{\text{sub}'}| = |V_k^{\text{new}}|$. Initially, let $CL_{ki}^{\text{sub}'} = \mathcal{M}_i^{\text{o}} \bigcup \mathcal{M}_i^{\text{s}}$ for all $CL_{ki}^{\text{sub}'} \in V_k^{\text{sub}'}$, $i \in [1, 2, \ldots,$

Algorithm 2. The teasing apart inheritance algorithm

Input: Inheritance network G_2.
Output: Restructured inheritance network G_2^{R}.

1. All the entities defined in the root node $CL_k^{\mathrm{root}} \in V_{\mathrm{root}}$ and relations between them are considered as the method-level weighted undirected network, where $k \in [1, 2, ..., |V_{\mathrm{root}}|]$. On this basis, the clustering analysis is used to regroup class CL_k^{root}, then,
① If class CL_k^{root} was not decomposed, we should delete the node CL_k^{root} and all the edges connecting to it from network G_2. Here, we denote V_k^{sub} as the direct subclass set of class CL_k^{root}. It follows that the direct subclasses in set V_k^{sub} become the root nodes.
Let $V_{\mathrm{root}} = V_{\mathrm{root}} \setminus CL_k^{\mathrm{root}} \bigcup CL_k^{\mathrm{sub}}$, and repeat step 1.
② Class CL_k^{root} should be subjected to Extract Class if it was split by clustering analysis. The new class set extracted from CL_k^{root} is denoted as V_k^{new}, and the restructured root node is represented as CL_k^{R}. Obviously, each class of V_k^{new} is considered as a root of the extracted inheritance tree.
2. Create the instance variables of classes belonging to set V_k^{new} in class CL_k^{R}. Consequently, the extracted inheritance trees can be invoked by the main inheritance tree rooted at class CL_k^{R}.
3. We split the direct subclasses of CL_k^{root} according to the structure of the new extracted classes and the principle of "high cohesion and low coupling".
4. If $CL_{km}^{\mathrm{sub}} \in V_k^{\mathrm{sub}}$ has direct subclasses, then we repeat step 3, where $m \in [1, 2, ..., |V_k^{\mathrm{sub}}|]$. The process continues, and the refactoring operations stop when all the leaf nodes have been analyzed. Finally, we obtain the restructured inheritance network G_2^{R} by updating G_2.

Fig. 7. Description of the algorithm for teasing apart inheritance.

$|V_k^{\mathrm{new}}|]$ and $\mathcal{M}_i^{\mathrm{o}}, \mathcal{M}_i^{\mathrm{s}} \subseteq CL_{km}^{\mathrm{sub}}$, where $CL_{ki}^{\mathrm{sub}'}$ denotes the subclass of the new class $CL_{ki}^{\mathrm{new}} \in V_k^{\mathrm{new}}$ extracted from class CL_k^{root}, and $\mathcal{M}_i^{\mathrm{o}}$ and $\mathcal{M}_i^{\mathrm{s}}$ represent the method sets defined in class CL_{km}^{sub} that override and invoke the super-methods defined in class CL_{ki}^{new}, respectively, which satisfy $\mathcal{M}_i^{\mathrm{s}} \bigcap \mathcal{M}_j^{\mathrm{s}} = \varnothing$, $\mathcal{M}_i^{\mathrm{o}} \bigcap \mathcal{M}_j^{\mathrm{o}} = \varnothing, i, j \in [1, 2, \ldots, |V_k^{\mathrm{new}}|], i \neq j$.

(2) Let $CL_k^{\mathrm{SR}'}$ be the subclass of the restructured root node CL_k^{R}, $CL_k^{\mathrm{SR}'} = (\mathcal{M}_{\mathrm{root}}^{\mathrm{s}} \backslash \mathcal{M}_{\mathrm{union}}^{\mathrm{s}}) \bigcup (\mathcal{M}_{\mathrm{root}}^{\mathrm{o}} \backslash \mathcal{M}_{\mathrm{union}}^{\mathrm{o}})$, where $\mathcal{M}_{\mathrm{union}}^{\mathrm{s}} = \mathcal{M}_1^{\mathrm{s}} \bigcup \mathcal{M}_2^{\mathrm{s}} \bigcup \ldots \bigcup \mathcal{M}_{|V_k^{\mathrm{new}}|}^{\mathrm{s}}$, $\mathcal{M}_{\mathrm{union}}^{\mathrm{o}} = \mathcal{M}_1^{\mathrm{o}} \bigcup \mathcal{M}_2^{\mathrm{o}} \bigcup \ldots \bigcup \mathcal{M}_{|V_k^{\mathrm{new}}|}^{\mathrm{o}}$, $\mathcal{M}_{\mathrm{root}}^{\mathrm{s}}$ and $\mathcal{M}_{\mathrm{root}}^{\mathrm{o}}$ denote the method sets containing all the methods invoking and overriding the inherited methods, respectively, which satisfy $\mathcal{M}_{\mathrm{root}}^{\mathrm{s}}, \mathcal{M}_{\mathrm{root}}^{\mathrm{o}} \subseteq CL_{km}^{\mathrm{sub}}$.

(3) Let $\overline{CL_{km}^{\mathrm{sub}}} = CL_{km}^{\mathrm{sub}} \backslash (CL_{k1}^{\mathrm{sub}'} \bigcup CL_{k2}^{\mathrm{sub}'} \bigcup \ldots \bigcup CL_{k|V_k^{\mathrm{new}}|}^{\mathrm{sub}'})$, and we then consider each element of $\overline{CL_{km}^{\mathrm{sub}}}$, class $CL_{ki}^{\mathrm{sub}'}$, and $CL_k^{\mathrm{SR}'}$ as $\left|\overline{CL_{km}^{\mathrm{sub}}}\right| + |V_k^{\mathrm{new}}| + 1$ communities at the beginning of the clustering analysis. According to the clustering algorithm described in Section 3.2, we need to compute the increments in the weighted modularity before merging each community pair in the network. To avoid aggregating the subclasses in set $V_k^{\mathrm{sub}'}$, for any communities CN_p and CN_q, if $CL_{ki}^{\mathrm{sub}'} \subseteq CN_p$ and $CL_{kj}^{\mathrm{sub}'} \subseteq CN_q$, then we let the modularity increments $\Delta Q_{pq} = 0$, where $p, q \in [1, 2, \ldots, |C|], i, j \in [1, 2, \ldots, |V_k^{\mathrm{new}}|], p \neq q, i \neq j$.

(4) After regrouping all the entities from class CL_{km}^{sub}, we obtain the community set $C = \{CN_1, CN_2, \ldots, CN_{|C|}\}$, and each community is considered as a new class extracted from CL_{km}^{sub}. For all $p, q \in [1, 2, \ldots, |C|], i \in [1, 2, \ldots, |V_k^{\mathrm{new}}|]$, the inheritance relationships between CN_p and CL_{ki}^{new} are described as follows.
①If $CL_{ki}^{\mathrm{sub}'} \subseteq CN_p$, then CN_p inherits class CL_{ki}^{new}.
②If $CL_k^{\mathrm{SR}'} \subseteq CN_q$, then CN_q inherits class CL_k^{R}.

(5) Create $|C| - 1$ instance variables for the new classes extracted from subclass CL_{km}^{sub} in class $CL_k^{\mathrm{SR}'}$.

4 An Illustrative Example

We employed the code snippet shown in Fig. 3 as an example to further demonstrate the effectiveness of the proposed algorithm. It should be noted that all the sample classes were selected from JHotDraw software and modified by moving methods, so "bad smells" were injected into the codes. The class *UndoRedoManager* contained "undo" and "redo" functionalities because it was created by merging two well-designed local inner classes: *UndoAction* and *RedoAction*. Furthermore, we moved parts of strongly related methods from class *DefaultDrawingView* to classes *DrawingPanel* and *PertPanel*, and thus the inheritance tree rooted at the class *DrawingPanel* encapsulated more than one functionality. In all the following case studies, Th_1 was the average SSW in the network and $Th_2 = avg_0$.

The values of α, β, γ, and η in Eq. (6) were assigned as 0.5, 0.2, 0.2, and 0.1. First, the refactoring operations described in Fig. 6 were applied to the classes of the non-inheritance hierarchies and the leaf nodes of the inheritance hierarchies. Furthermore, the multi-relation directed networks were updated according to the refactoring results. Finally, we restructured the classes of the inheritance hierarchies based on the process shown in Fig. 8. The cluster dendrogram for the connected component cc_1 obtained by the first step is shown in Fig. 9. The value of the weighted modularity Q was increased from 0 to 0.3876 by 25 community merging operations. After the cluster analysis, we obtained four new classes. Fig. 10 shows the refactoring suggestions displayed in a tooltip.

In Fig. 11a, the green, red, and blue nodes represent the entities in the original classes *RestoreDataEdit*, *NetPanel*, and *UndoRedoManager*, respectively. It was suggested that the class *UndoRedoManager* should be subjected to the extract class refactoring operation to improve its cohesion. As expected, the strongly related entities belonging to the original inner class *RedoAction*, were extracted to form a new class which is denoted as $UndoRedoManager_{new2}$. Furthermore, the entities had closer relationships with the *UndoRedo*

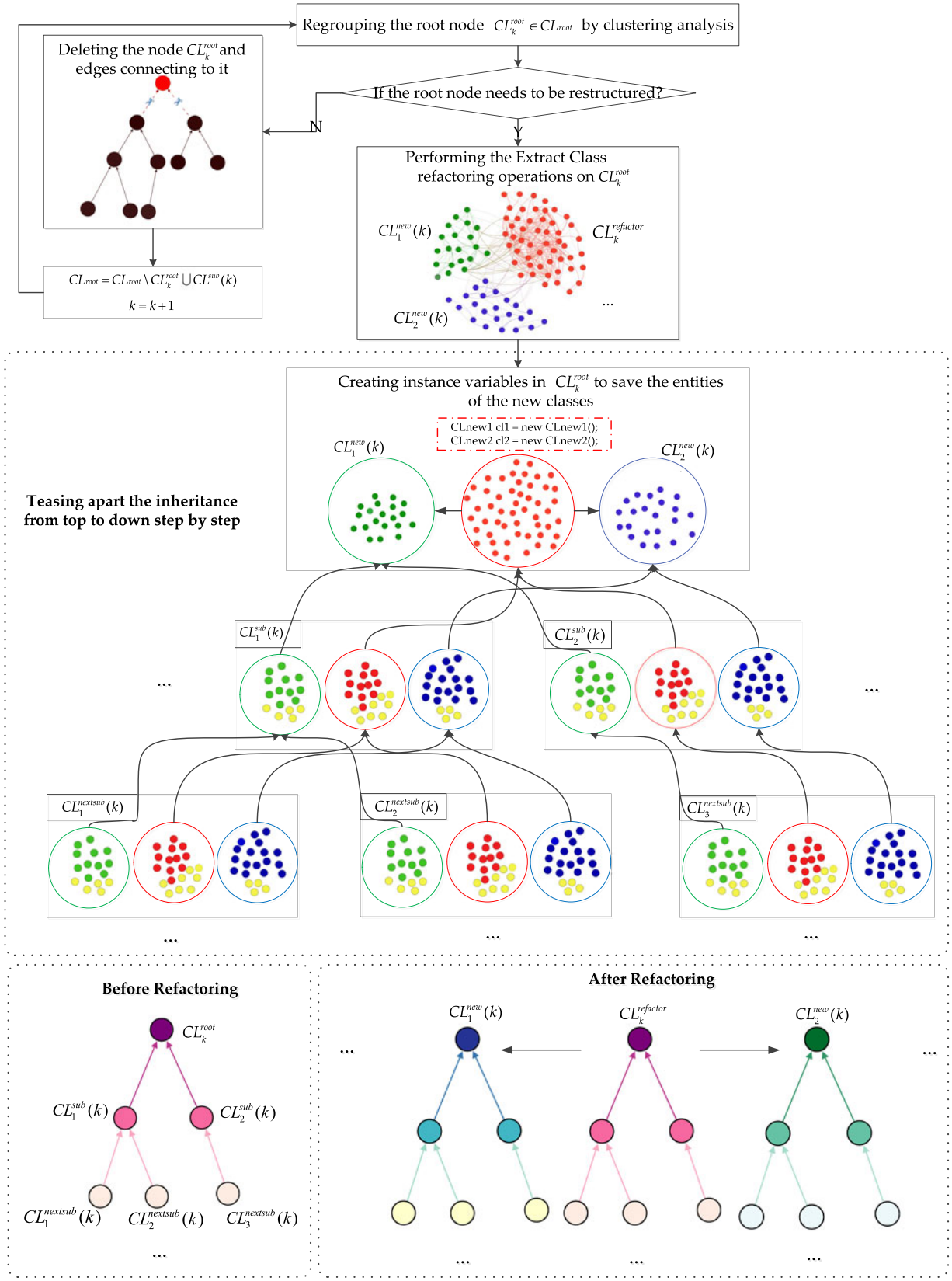


Fig. 8. Refactoring process for the inheritance hierarchies.

$Manager_{new1}$ and $UndoRedoManager_{new2}$ was suggested to be moved to the target classes to minimize the coupling.

It should be noted that the entities *UndoRedoManager.undoOrRedo*(...), *UndoRedoManager.undoOrRedoInProgress*, and *UndoRedoManager.updateActions*(...) had no sharing attribute or invocation relationships, however, they were executed together three times in the methods *NetPanel.undoDrawing*(...), *NetPanel.undoData*(...), and *RestoreDataEdit.redoEdit*(...), respectively. Consequently, these three methods were clustered together in the first few steps,

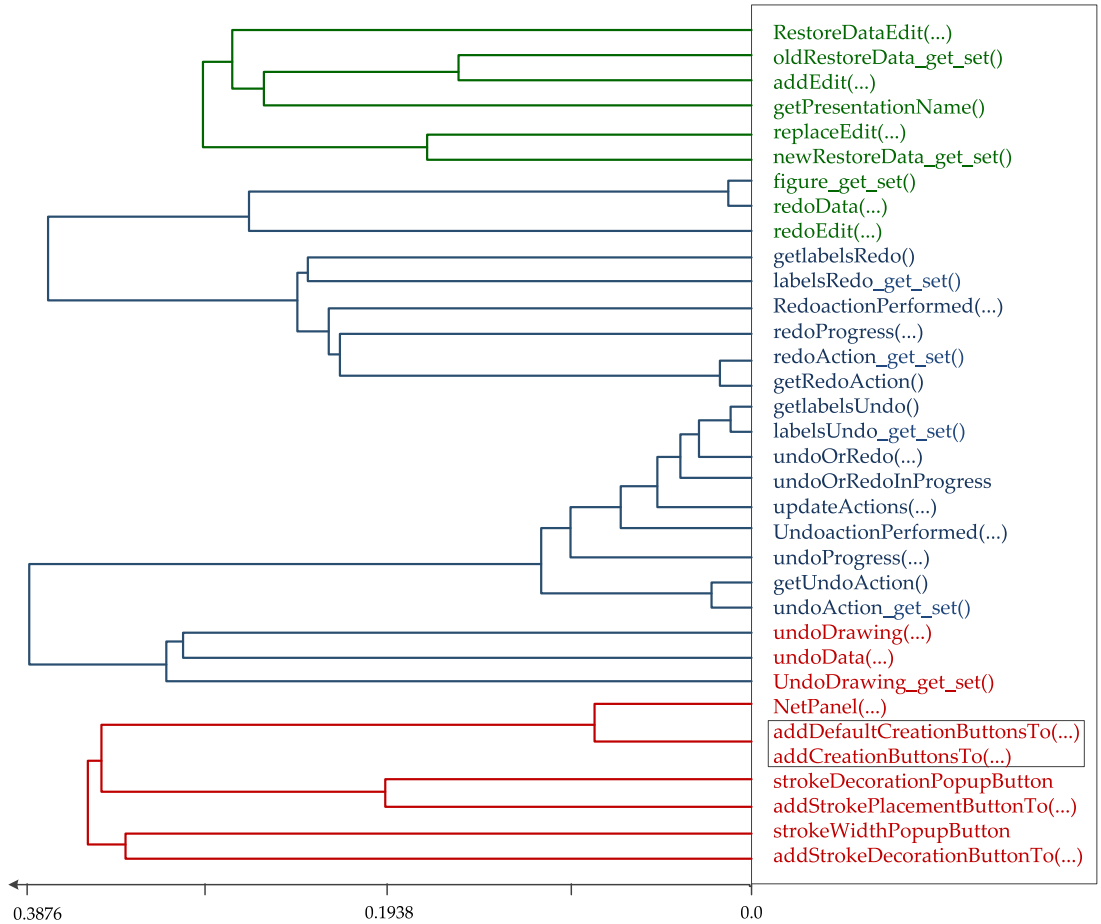


Fig. 9. Cluster dendrogram for the refactoring example.

which led to large increments in modularity, as shown in Fig. 9. As a typical example, when we ignored the FCW parameter and consider the parameter set: $\alpha = 0.6, \beta = 0.3$, $\gamma = 0$, and $\eta = 0.1$, then Fig. 11b clearly shows that it was suggested that $UndoRedoManager.undoOrRedo$(...), $UndoRedoManager.undoOrRedoInProgress$, and $UndoRedoManager.updateActions$(...) should be moved to three different new classes, which increased the coupling between classes.

In another extreme example, when we eliminated the semantic relationships, i.e., using $\alpha = 0.5, \beta = 0.2, \gamma = 0.3$, and $\eta = 0$, then Fig. 11c shows the community partition for the methods obtained with these parameter settings. Class $NetPanel$ was split into two classes because the structural cohesion of the entities was not sufficiently high. However, according to their semantics, the divided methods implemented the button-related functions so they did not need to be restructured. The semantic relevance is necessary to calculate the similarity between methods, but it has negative effects on the refactoring accuracy of the proposed algorithm if the coefficient of SSW is assigned an excessively

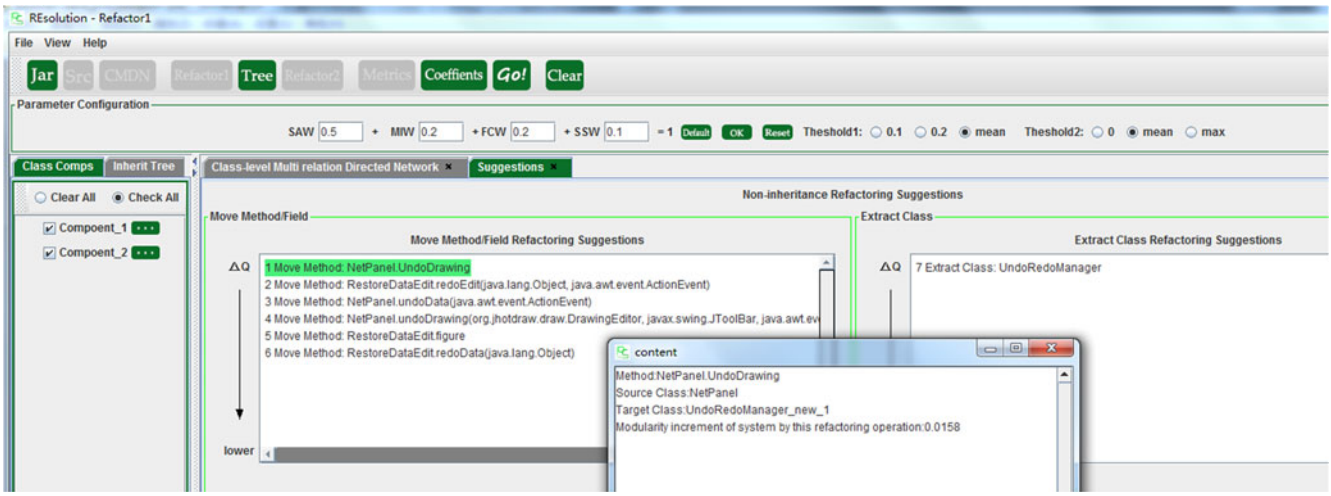
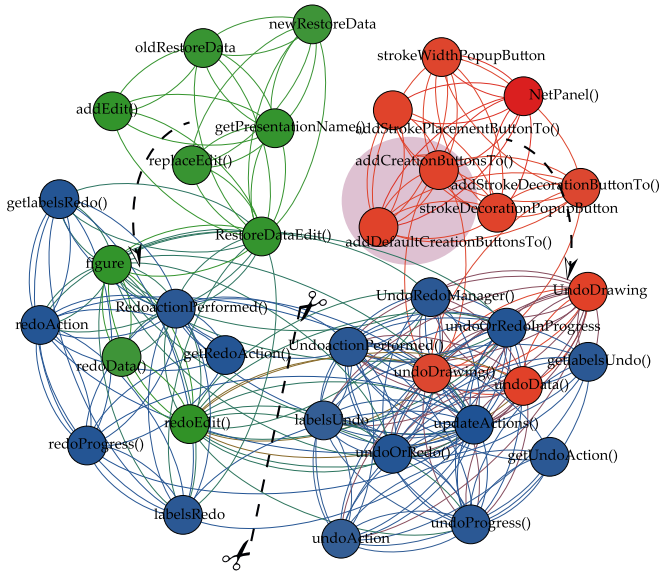
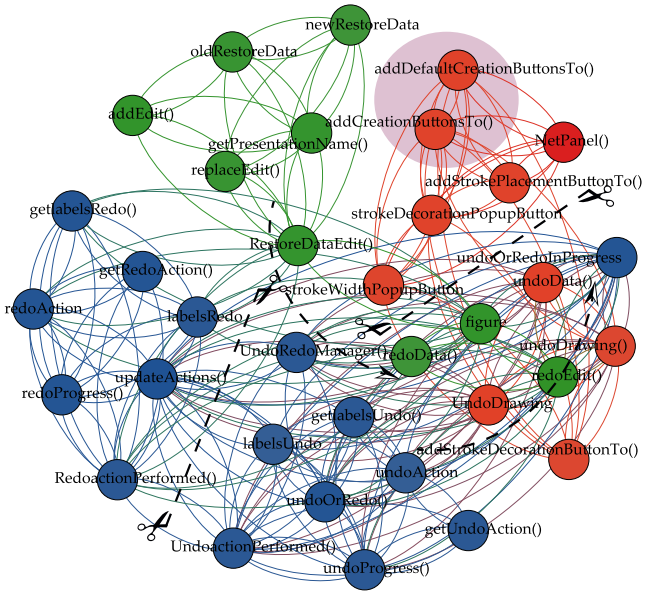


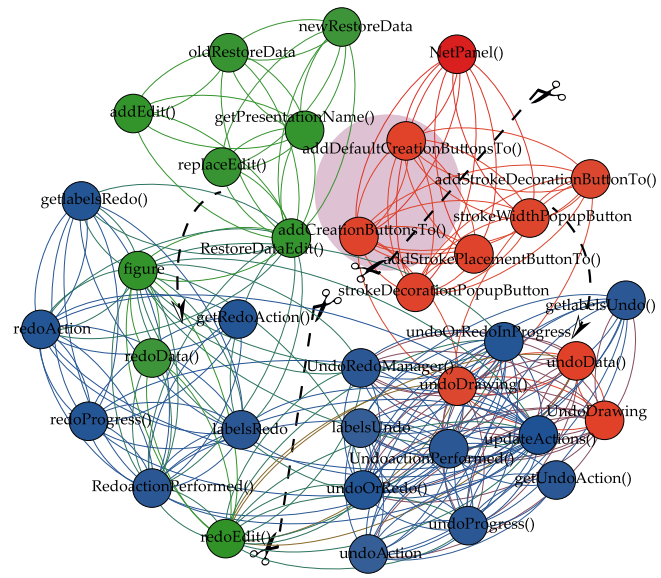
Fig. 10. Tooltip indicating the refactoring suggestions.



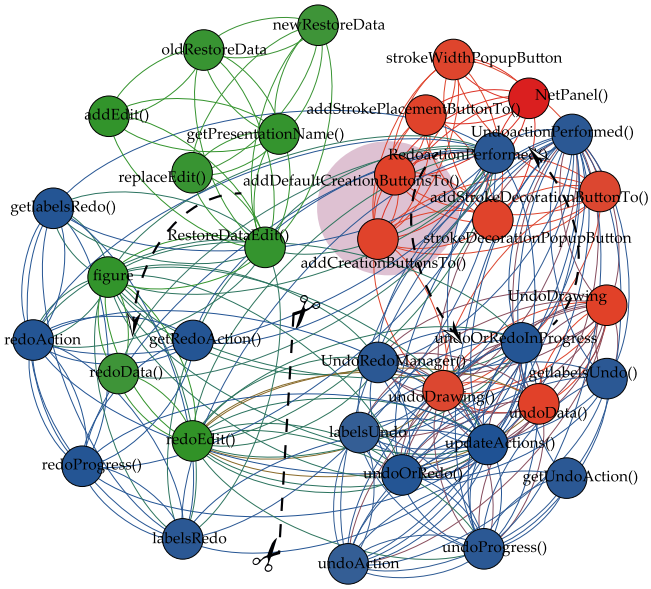
(a) Optimal parameter settings: $\alpha = 0.5, \beta = 0.2, \gamma = 0.2$, and $\eta = 0.1$



(b) Ignoring the FCW coefficient: $\alpha = 0.6, \beta = 0.3, \gamma = 0$, and $\eta = 0.1$



(c) Ignoring the SSW coefficient: $\alpha = 0.5, \beta = 0.2, \gamma = 0.3$, and $\eta = 0$



(d) Higher SSW coefficient: $0.3 \leq \alpha + \beta + \gamma \leq 0.8$ and $0.2 \leq \eta \leq 0.7$

Fig. 11. Partitions of the method-level network under different parameter settings.

high value. Fig. 11d shows the refactoring results obtained for $\alpha + \beta + \gamma \in [0.3, 0.8]$ and $\eta \in [0.2, 0.7]$. In this case, the methods $UndoRedoManager.UndoactionPerformed(...)$, $UndoRedoManager.RedoactionPerformed(...)$, and $NetPanel.addDefaultCreationButtonsTo(...)$ were clustered together because of their relatively high semantic similarity. These three methods used words such as "action" and "label" frequently, but they implemented three different functions. Thus, the refactoring suggestions were unreasonable. As discussed above, all four types of coupling relationships make sense for regrouping methods with appropriate settings for the coefficients.

During the clustering process, agglomerating the method $a_i\_get\_set()$ and its corresponding *Getter* or *Setter* methods always led to a greater increase in the weighted modularity Q because there were two types of relationships among these methods, i.e., sharing attribute and invocation, where the weights of the edges between them were relatively high. Thus, they were not divided in the community detection process. Similarly, all the methods that accessed the same attributes were agglomerated into a cluster at the extreme. Thus, we obtained a better compromise between improving the cohesion and hiding the information for a class in this manner.

During the restructuring of inheritance hierarchies, the root node *DrawingPanel* was split based on the principle of "high cohesion and low coupling," and we created an instance variable for class $DrawingPanel_{new2}$ in $DrawingPanel_{new1}$ in order to save the entities in $DrawingPanel_{new2}$. Thus, $DrawingPanel_{new2}$ was the root of the new inheritance tree, which could be invoked by the tree rooted at class $DrawingPanel_{new1}$. Similarly, in the second level, the subclass *PertPanel* was decomposed into two new classes. We

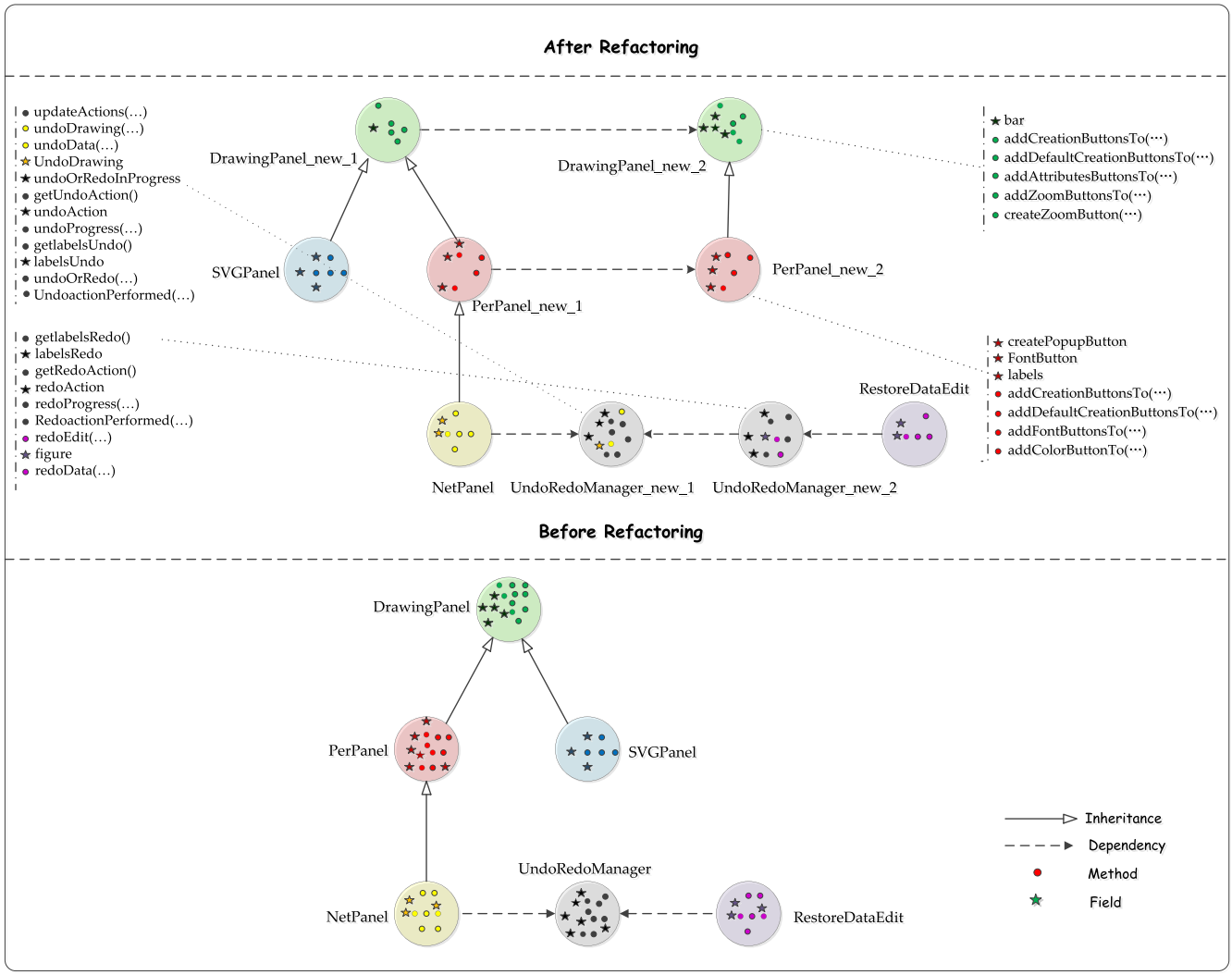


Fig. 12. The structure of the example system before and after refactoring.

let $PertPanel_{new1}$ and $PertPanel_{new2}$ inherit the classes $DrawingPanel_{new1}$ and $DrawingPanel_{new2}$, respectively; because they contained the methods invoking or overriding the methods defined in their corresponding superclasses. Finally, the quality of the leaf node $SVGPanel$ was good, so it did not need to be restructured by the clustering analysis. Fig. 12 shows that our approach could tease the original inheritance tree apart into two new trees, with less average inheritance depth and a smaller average number of children in each level without changing the code behaviors.

5 Adjusting the Parameters

For a given method-level network, different community partitions usually yield different weighted modularity values. Consequently, the refactoring accuracy is influenced by the settings of α, β, γ, and η, which represent the coefficients of SAW, MIW, FCW, and SSW in Eq. (6), respectively. We addressed the following research question in our case study.

- RQ_1: How can we assign the values of α, β, γ, and η to make them applicable to various software systems?

To address the research question (RQ_1), we used different parameter configurations to decompose the artificial god classes based on the weighted clustering algorithm. Next, we calculated the accuracy rates for refactoring by comparing the differences between the original classes and the new classes. The coefficients corresponding to the highest accuracy rate were considered as the optimal settings.

5.1 Experimental Design

The algorithm for adjusting coefficients was inspired by the method described by Bavota et al. [14], [15]. We randomly selected NU classes from the open source software systems and then merged the selected classes into an artificial god class. In the best case, the new classes obtained by decomposing the artificial god class were the same as the original classes. Consequently, the original classes were considered as the "gold standard" and the degree of similarity between the original classes and the new classes was treated as the refactoring accuracy. Similar to [15], we also used the MoJo EFfectiveness Metric (MoJoFM) [42] to evaluate the accuracy rate. If we suppose that PA_{new} is the partition obtained after the clustering analysis and PA_{ori} is the partition of the originally selected classes, then $MoJoFM$ is calculated using

$$MoJoFM(PA_{new}, PA_{ori}) = 1 - \frac{mno(PA_{new}, PA_{ori})}{\max(mno(\forall PA_{new}, PA_{ori}))}, \quad (10)$$

TABLE 1
Detailed Information About the Top 10 Selected Systems

Software	Version	Introduction	TNC	TNE	ANM	ANA	ALC
Elasticsearch	1.6.0	Elasticsearch is a distributed restful search engine built for the cloud	4,152	12,748	7.05	2.21	161.78
Android-async-http	1.4.7	An asynchronous, callback-based http client for Android built on top of Apache's "HttpClient" libraries	73	126	6.75	4.27	157.68
Iosched	2013	An Android app for the conference	495	1,057	5.78	3.93	127.59
Libgdx	1.6.2	LibGDX is a cross-platform Java game development framework based on OpenGL (ES)	1,988	4,572	7.22	2.67	105.60
Android-Annotations	3.3.1	Android-Annotations is an Open Source framework that speeds up Android development	696	1,948	4.79	3.94	79.20
Okhttp	2.4.0	An HTTP & SPDY client for Android and Java applications.	239	522	4.70	3.04	249.83
RxJava	1.0.12	RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs using observable sequences	436	951	7.65	3.52	220.88
Spring-framework	4.2.0	The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications – on any type of deployment platform	6,139	16,014	6.32	4.55	155.54
Zxing	4.7.3	ZXing is a multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages	4,703	9,241	5.71	4.01	138.26
MPAndroidChart	2.1.0	MPAndroidChart is an easy to use chart library for Android	149	297	10.54	3.62	202.70

where $mno(PA_{new}PA_{ori})$ represents the minimum distance between partition PA_{new} and the gold standard partition PA_{ori}, and $max(mno(\forall PA_{new}, PA_{ori}))$ denotes the maximum number of possible move or join operations when transforming any partition PA_{new} into the gold standard partition PA_{ori}. If the clustering analysis produces the longest distance between PA_{new} and PA_{ori}, then the value of $MoJoFM$ is 0. If the new extracted classes are the same as the original classes, then $MoJoFM$ is equal to 1.

To obtain the best configuration, the merged classes that we selected had to satisfy the following conditions.

(1) The NU classes to be merged should be well designed and the cohesion of each selected class should be higher than the average cohesion of all the classes in the system. The clustering algorithm used by our approach does not specify the number of clusters, so the god class may be split into more than NU new classes in the regrouping process if the selected original classes have cohesion or coupling problems. To ensure the diversity of sampling, we randomly selected NM sets of classes from each system, where each class set contained NU classes and $NM = 100$. Furthermore, the NU classes were merged into an artificial god class for decomposition. In general, a god class should be split into two or three new classes, where $NU \in [2, 3]$ for the extract class refactoring operations [15]. However, in our study, we regrouped the entity set obtained by merging the classes in each connected component of the residual network $\overline{G_1}$. Thus, these more general coefficients were effective even when the value of NU was higher. To ensure that the coefficients were not sensitive to the value of NU, we assumed that $NU \in \{2, 3, \ldots, |CS_{max}|\}$, where $|CS_{max}|$ represents the number of classes in the largest connected components of $\overline{G_1}$.

(2) From the perspective of a class-level network, the NU classes for merging should comprise a connected component. If the NU selected classes do not depend on each other, then the structural coupling weights between the methods defined in different classes are 0. In this case study, when $\eta = 0$, the artificial god class was split almost perfectly into NU new classes by clustering. Thus, classes without dependencies were not good experimental objects for tuning the coefficients.

To ensure the best configuration is applicable to various systems, we used 50 types of open source software from GitHub as the experimental data. All the selected systems[5] were ranked in the top 50 of a star-based rating list in July 2015. Detailed information about the top 10 open source systems is provided in Table 1, where TNC represents the total number of classes, TNE denotes the total number of edges in the class-level network, and ANM, ANA, and ALC are the average number of methods, attributes, and lines of code, respectively. Furthermore, we propose a random walk model based on the class-level dependency network G_1, which meets the criteria defined above for selecting the classes. The detailed algorithm is shown in Fig. 13. Similar to [15], all of the methods inherited from the superclasses and constructors of the selected classes were excluded when merging them into the artificial god classes. If the selected classes contained entities with the same name, then we renamed the entities before merging them. An illustration of the creation of an artificial god class is shown in Fig. 14.

5. https://github.com/wangying8052/Selected-System-information

Algorithm 3. The algorithm of selecting classes to be merged.

Input: Class-level dependency network G_2 of the open source system.

Output: The merged class set $VM = \{V_1, V_2, \cdots\cdots, V_{NM}\}$, and $NM = 100$;

Do{
1. Initialization: Suppose $NU \in [2, 3, ..., |CS_{\max}|]$ is the total number of classes in set $V_i \in VM$. Let $V_{\text{seed}} = \varnothing, V_i = \varnothing, V_H = \varnothing$, where V_{seed} represents the class set that can be considered as the seeds, V_i denotes the class set selected to be merged, and V_H is the class set that has been selected as seeds.
2. In network G_2, we randomly select a class whose cohesion is higher than the average cohesion of all the classes in the system as the seed node CL_{seed}.
Let $V_i = V_i \bigcup CL_{\text{seed}}, V_H = V_H \quad CL_{\text{seed}}$.
3. Traverse all the neighbours of class CL_{seed}, and denote the classes whose cohesion is higher than the average cohesion as the node set $V_{\text{seed}}^{\text{NB}}$. Let $V_{\text{seed}} = V_{\text{seed}} \bigcup (V_{\text{seed}}^{\text{NB}} - V_H)$.
4. If $|V_{\text{seed}}^{NB}| \geq NU - |V_i|$, then we randomly delete $|V_{\text{seed}}^{\text{NB}}| - NU + |V_i|$ classes from set $V_{\text{seed}}^{\text{NB}}$. Let $V_i = V_i \bigcup V_{\text{seed}}^{\text{NB}}, i++$.
If $|V_{\text{seed}}^{NB}| < NU - |V_i|$ && $V_{\text{seed}} \neq \varnothing$, then we randomly select a class from V_{seed} as the new seed CL_{seed}. Let $V_H = V_H \bigcup CL_{\text{seed}}$, and repeat steps (3)-(4).
If $|V_{\text{seed}}^{NB}| < NU - |V_i|$ && $V_{\text{seed}} = \varnothing$, continue;
} while ($i < NM$)

Fig. 13. Description of the algorithm for selecting the classes that need to be merged.

In our approach, we used the lack of cohesion of methods ($LCOM$) [45] and conceptual cohesion of classes ($C3$) [37] as metrics to measure the cohesion of classes, and the coupling between object classes (CBO) [45] and message passing coupling (MPC) [46] as metrics to evaluate the coupling between classes. For a class, $LCOM$ is defined as the total number of method pairs that do not share attributes subtracted from the total number of method pairs that share attributes with each other. $C3$ denotes the ratio of the summed conceptual similarities between each pair of methods relative to the total number of method pairs in the class. The CBO metric value for class CL_A is defined as the number of classes that have dependency relationships with class CL_A. MPC represents the total number of methods that do not belong to Class CL_A but that have dependencies with the methods in class CL_A. Consequently, when the value of $LCOM$ is lower, the cohesion of the classes is better, and higher values for $C3$, CBO, and MPC indicate the better quality of the classes.

We had no quality models, so the system JHotDraw, which was developed as a "design exercise," was treated as a quality standard in our study [15]. The algorithm for selecting the classes to be merged was carried out on the 50 systems and JHotdraw software. Note that to guarantee the quality of experimental data, the classes ranked in the bottom 20 persent of the system in cohesion measurement were filtered out beforehand. Table 2[6] shows a comparison between them in quality measurements. We can see that the average coupling and cohesion metric values of the classes selected from the 50 systems are all close to or better than those of JHotDraw which relies greatly on well-known design patterns. Therefore, the experimental data could be treated as suitable subjects for adjusting the coefficients.

5.2 Analysis and Comparison

Under the condition that $\alpha + \beta + \gamma + \eta = 1$, we iteratively adjusted the parameter configuration satisfying $\alpha \in [0, 0.1, \ldots, 1]$, $\beta \in [0, 0.1, \ldots, 1 - \alpha]$, $\gamma \in [0, 0.1, \ldots, 1 - \alpha - \beta]$, and $\eta \in [0, 0.1, \ldots, 1 - \alpha - \beta - \gamma]$. Nearly all the method pairs had the semantic relevance, so similar to approaches [14], [24], we set a threshold Th_1 to eliminate the lower semantic similarity values. For the three cases of $Th_1 = SSW_{med}$, $Th_1 = 0.1$, and $Th_1 = 0.2$, where SSW_{med} represents the median SSW between the methods, we averaged the accuracy rates of the refactoring operations performed on all the artificial god classes selected from the 50 systems under each coefficient setting. Fig. 15 shows a comparison of the refactoring accuracies corresponding to the different parameters.[7] After analyzing the results, we can make the following conclusions.

(1) Clearly, by setting the threshold Th_1 equal to the average semantic similarities, we could achieve higher refactoring accuracy. Fig. 15 shows that when α and β were kept constant, the accuracy tended to increase rapidly with the decrease in the semantic coefficient η. Furthermore, the average accuracy rate reached the peak values for all $\eta \in [0.1, 0.3]$. The troughs in the accuracy curve occurred with the setting of $\gamma = 0$, but the sum of γ and η remained unchanged. In all cases, the lowest accuracy occurred at $\eta = 1$, and $\alpha = \beta = \gamma = 0$. Based on these observations, we conclude that when the parameter $\eta \in [0.1, 0.3]$, the semantic coupling weights between methods were valuable for the proposed refactoring algorithm.

According to Bavota et al. [14], [15], combining semantic measures by clustering analysis can improve the accuracy rate for extract class refactoring. In their approaches, the coefficient for semantic similarity should be set higher than that of the structural similarity to obtain better refactoring results. The object of extract class refactoring is only one god class, so the artificial classes used for tuning the coefficients are created by merging two or three classes.

6. Cohesion and coupling metric values of the 50 selected systems are available at: https://github.com/wangying8052/Cohesion-and-coupling-metric-values

7. The statistics for the $MoJoFM$ values are available at: https://github.com/wangying8052/Statistics-of-Merging-God-classes

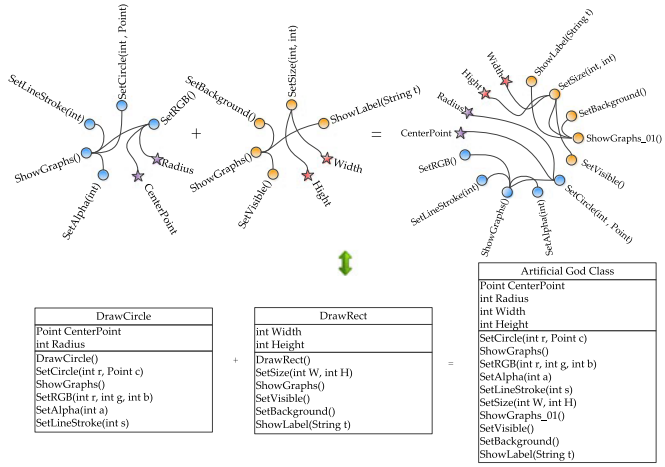


Fig. 14. Illustration of the creation of an artificial god class.

However, in our method, more classes are merged into an artificial god class when adjusting the coefficients that are suitable for system-level refactoring. Merging more classes allows more methods to implement different functionalities with a higher likelihood of semantically associating with each other. This is because the different coding and commenting styles used by different developers and the evolution of different versions may disrupt the matching between code elements and the functionalities that they actually reflect [23]. Thus, higher semantic coupling weights may decrease the system-level refactoring accuracy. Furthermore, the artificial god classes are split by the $MaxFlow - MinCut$ algorithm [14], whereas our approach uses hierarchical clustering analysis to restructure the classes. The difference between the two clustering algorithms employed can also affect the refactoring results.

(2) Based on the accuracy curves for the three cases, we can see that when $\gamma = 0$ and $\alpha + \beta + \eta = 1$, the average accuracies of the 50 systems were 0.6032, 0.5306, and 0.5663, which increased to 0.6186, 0.5423, and 0.5750 when $\gamma \neq 0$ and $\alpha + \beta + \gamma + \eta = 1$, respectively. Clearly, combining with FCW during refactoring operations can obtain higher accuracy compared with only considering the other three types of coupling relationships.

(3) Obviously, the optimal settings were $\alpha = 0.5, \beta = 0.2$, $\gamma = 0.2, \eta = 0.1$ and $Th_1 = SSW_{med}$. As shown in Fig. 15, the average accuracy had a significant positive correlation with the parameter α. Based on the three cases together, the highest average accuracy rate was obtained when $\alpha \in [0.5, 0.6], \beta \in [0.1, 0.2]$, $\gamma \in [0.2, 0.3]$ and $\eta \in [0.1, 0.2]$. Thus, to obtain superior refactoring results, the sharing attribute weight should play the dominant role when measuring the coupling between methods. We extracted as many methods as possible that shared the same attributes to form a new class, thereby balancing the tradeoff between improving the cohesion and hiding the information for the class.

We decomposed all the artificial classes based on the optimal coefficients $\alpha = 0.5, \beta = 0.2, \gamma = 0.2, \eta = 0.1$ and $Th_1 = SSW_{med}$. The $LCOM$ and $C3$ metrics were then used to measure the cohesion of the original classes, merged god classes, and extracted new classes, and the CBO and MPC metrics were employed to evaluate the coupling of the classes mentioned above. Fig. 16 shows that all the quality metric values for the artificial classes were far from those for the original and extracted classes, thereby indicating that they had cohesion and coupling problems, and thus they needed to be restructured. Compared with the merged and original classes, the average cohesion was improved for the new classes and the coupling decreased by different amounts.

To further address the research question (RQ_1), we performed F-tests and $Wilcoxon$ tests to statistically analyze the restructuring results. Table 3 shows the statistical results obtained for the top five systems selected from GitHub,

TABLE 2
Cohesion and Coupling Metric Values of the Top 5 Selected Systems

Cohesion /Coupling	Metrics	JHotDraw	Elastic search	Android-async-http	Iosched	Libgdx	Android annotations	Overall (50 systems)
CBO	Maximum	10	8	9	12	7	9	15
	Minimum	2	2	3	2	3	3	2
	mean	4	4	4	5	4	5	5
	Std.dev	1.3	1.2	1.5	1.7	1.1	1.6	1.9
MPC	Maximum	26	25	19	22	27	23	34
	Minimum	7	5	6	6	4	6	4
	mean	13	11	12	16	15	18	15
	Std.dev	4.4	5.5	4.9	4.8	6.1	5.2	5.7
$C3$	Maximum	0.48	0.50	0.51	0.55	0.57	0.52	0.61
	Minimum	0.23	0.37	0.36	0.35	0.38	0.41	0.32
	mean	0.37	0.44	0.43	0.44	0.45	0.43	0.41
	Std.dev	0.07	0.07	0.06	0.06	0.06	0.07	0.11
$LCOM$	Maximum	197	205	190	172	169	231	284
	Minimum	5	12	17	16	6	9	3
	mean	60	65	46	53	49	66	58
	Std.dev	42	46	36	42	37	45	53

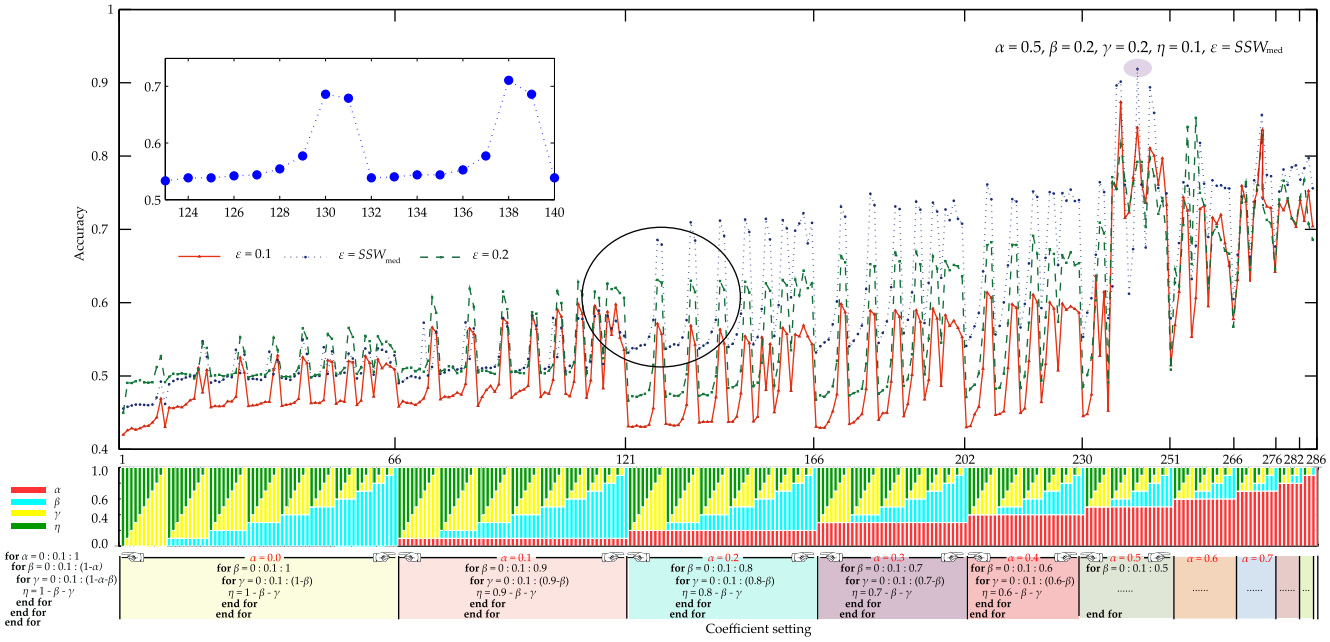
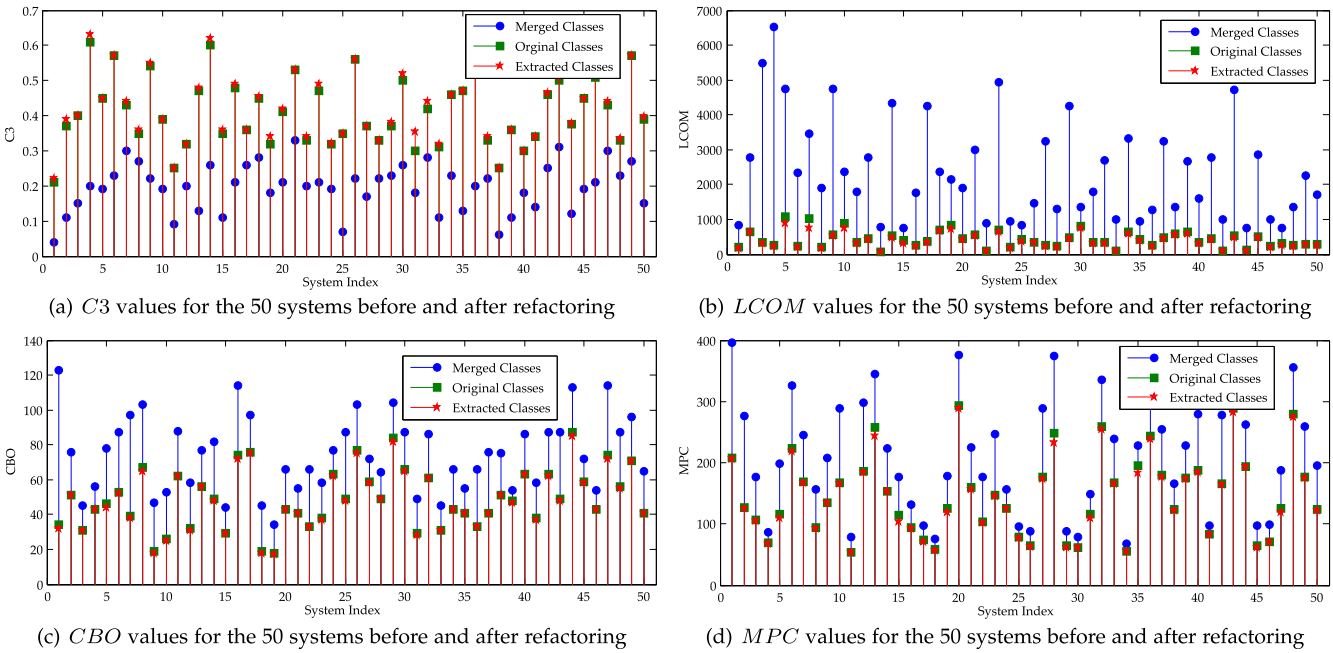


Fig. 15. Comparison of the accuracy values under different coefficients and thresholds.

where the p-values are shown in bold. The coefficient settings for the semantic method were $\alpha = \beta = \gamma = 0$, $\eta = 1.0$, and $Th_1 = SSW_{med}$. The optimal settings of $\alpha = 0.5, \beta = 0.2$, $\gamma = 0.2, \eta = 0.1$, and $Th_1 = SSW_{med}$ were considered as the coefficients for the combined method. The coefficients $\alpha = 0.7, \beta = 0.2, \gamma = 0.1, \eta = 0$, and $Th_1 = SSW_{med}$, which corresponded to the highest accuracy when $\eta = 0$ and $\alpha + \beta + \gamma = 1$, were used as the configurations for the structural method. Table 4 shows clearly that the refactoring results obtained by the combined method were better than those obtained using either the structural or semantic methods. Furthermore, the structural method outperformed the semantic method in terms of accuracy for all the 50 systems. The differences between the semantic, structural, and combined methods were demonstrated by the results of the $Wilcoxon$ tests. Obviously, all the effect-size values were high or medium for the methods compared, and most of the p-values for the 50 systems were less than 0.01. Thus, the statistical results obtained for the methods compared were significant with all the systems.

We compared the accuracy rates for restructuring the artificial god classes using four different configurations, i.e., the merged best configuration, principal component analysis (PCA)-based configuration, variance coefficient-based configuration, and optimal $MoJoFM$-based configuration. These methods are described briefly in the following.



(a) $C3$ values for the 50 systems before and after refactoring

(b) $LCOM$ values for the 50 systems before and after refactoring

(c) CBO values for the 50 systems before and after refactoring

(d) MPC values for the 50 systems before and after refactoring

Fig. 16. Contrast analysis of the quality for the 50 systems.

TABLE 3
Statistical Results Obtained from the F-Test and $Wilcoxon$ Test

System	method	mean	median	Std. dev	Compared method	p-Value	Effect-size
Elasticsearch	Semantic	0.4135	0.4219	0.1003	Combined versus Structural	<**0.01**	0.7(Medium)
	Structural	0.8499	0.8510	0.0146	Combined versus Semantic	<**0.01**	0.9(High)
	Combined	0.9060	0.8709	0.0718	Semantic versus Structural	<**0.01**	0.9(High)
Android-async-http	Semantic	0.4530	0.4406	0.1216	Combined versus Structural	<**0.01**	0.7(Medium)
	Structural	0.8710	0.8938	0.1150	Combined versus Semantic	<**0.01**	0.9(High)
	Combined	0.8976	0.8911	0.1544	Semantic versus Structural	<**0.01**	0.8(High)
Iosched	Semantic	0.4391	0.4620	0.1107	Combined versus Structural	0.0129	0.7(Medium)
	Structural	0.8312	0.8533	0.1322	Combined versus Semantic	<**0.01**	0.9(High)
	Combined	0.9156	0.9218	0.1200	Semantic versus Structural	<**0.01**	0.8(High)
Libgdx	Semantic	0.4518	0.4262	0.0097	Combined versus Structural	<**0.01**	0.6(Medium)
	Structural	0.8921	0.8654	0.1249	Combined versus Semantic	<**0.01**	0.9(High)
	Combined	0.9077	0.9178	0.1093	Semantic versus Structural	<**0.01**	0.9(High)
Android-annotations	Semantic	0.4382	0.4456	0.1410	Combined versus Structural	0.0152	0.7(Medium)
	Structural	0.8508	0.8621	0.1574	Combined versus Semantic	<**0.01**	0.9(High)
	Combined	0.9123	0.9315	0.1308	Semantic versus Structural	<**0.01**	0.9(High)

- Merged best configuration: the personalized parameter settings corresponding to the highest accuracy are considered when adjusting the coefficients of each system.
- PCA-based configuration: Bavota et al. used PCA for extract class refactoring because it allows the identification of different dimensions and indicates the importance of each dimension of the data [15]. In our approach, the dimensions comprised the SAW, MIW, FCW, and SSW coupling relationships between method pairs.
- Variance coefficient-based configuration: a type of objective weight assignment method based on the ratio of the standard deviation relative to the average value.
- Optimal $MoJoFM$-based configuration: this method represents the optimal coefficients obtained for $\alpha = 0.5, \beta = 0.2, \gamma = 0.2, \eta = 0.1$ and $Th_1 = SSW_{med}$.

Table 4 shows that the accuracy rates obtained by the optimal $MoJoFM$-based configuration were very similar to those produced by the merged best configuration for all the selected software system. The average accuracy obtained by the optimal $MoJoFM$-based configuration, the PCA-based configuration, and the variance coefficient-based configuration differed by less than 0.0092, 0.3657, and 0.3704 compared with that produced using the merged best configuration, respectively. Thus, the optimal $MoJoFM$-based configuration was stable across the software systems.

5.3 Threats to Validity

The optimal coefficient settings were obtained using a large amount of experimental data and accepted evaluation metrics, but threats to the external validity affect the results of our study.

First, the premise for evaluating the refactoring accuracy was that all the classes could be merged into artificial classes with good quality. We selected several sets of classes with higher cohesion from the open source systems, which were recognized as well designed, but code smells were inevitably present in them. As shown in Fig. 16, some of the quality metric values were better than those for the original classes because there were still move method/field refactoring opportunities in the selected classes.

Another threat to the validity is that the optimal coefficient settings obtained were not specifically applicable to all of the software systems. Software is a type of artificial system, so finding general parameter settings that are suitable for all systems is a complex problem. To mitigate this threat, we selected classes from the 50 most popular systems on GitHub as the experimental data for tuning the coefficients. We compared four weight assignment schemes and according to the simulation results, the refactoring accuracy of the optimal $MoJoFM$-based configuration was very similar to that of the merged best configuration, and better than the refactoring accuracy obtained using the other algorithms for all the systems selected. Thus, the optimal coefficient settings were considered as a general configuration that was applicable to various systems.

6 Comparisons with Previous Research

Using clustering analysis to improve the quality of software is no longer a new research topic. Specially, similar

TABLE 4
Comparison of the Accuracy Rates Using Four Types of Configurations ($Th_1 = SSW_{med}$)

System	Merged Best configuration $(\alpha, \beta, \gamma, \eta)$	PCA-based configuration $(\alpha, \beta, \gamma, \eta)$	Variance coefficient-based configuration $(\alpha, \beta, \gamma, \eta)$	$MoJoFM$
Elasticsearch	(0.5, 0.2, 0.2, 0.1) **0.9060**	(0.2, 0.2, 0.1, 0.5) 0.5314	(0.2, 0.2, 0.1, 0.5) 0.5314	**0.9060**
Android-async-http	(0.5, 0.2, 0.2, 0.1) **0.8976**	(0.2, 0.2, 0.1, 0.5) 0.5611	(0.2, 0.2, 0.1, 0.5) 0.5611	**0.8976**
Iosched	(0.5, 0.1, 0.3, 0.1) **0.9180**	(0.3, 0.2, 0.1, 0.4) 0.5827	(0.3, 0.1, 0.1, 0.5) 0.5470	0.9156
Libgdx	(0.6, 0.1, 0.2, 0.1) **0.9112**	(0.3, 0.1, 0.0, 0.6) 0.5405	(0.2, 0.1, 0.1, 0.6) 0.5532	0.9077
Android-annotations	(0.5, 0.2, 0.2, 0.1) **0.9123**	(0.4, 0.1, 0.0, 0.5) 0.5590	(0.2, 0.2, 0.1, 0.5) 0.5384	**0.9123**

approaches [14], [15], [23] aimed to identify extract class refactoring opportunities by clustering. Our proposed approach can be considered as a system-level refactoring algorithm, but the main aim is to merge interdependent classes into a god class and then regroup them by clustering. In all the approaches described above, the ability of clustering to decompose the god class determines the refactoring results. Bavota et al. had already showed in Ref. [15] that the $MaxFlow - MinCut$ algorithm [14] had the clear limitation that only can split the god class into two new classes, and Bavota's extract class refactoring algorithm [15] performed better than it. For this reason, we only compare our algorithm with the approaches proposed by Bavota et al. [15] and Fokaefs et al. [23] in terms of the effectiveness of clustering in this section.

6.1 Research Questions and Experimental Design

Based on the context of our study, we addressed the following research questions.

- RQ_2: What are the advantages of our approach compared with clustering analysis using other refactoring algorithms?
- RQ_3: Do the refactoring suggestions obtained using our approach actually make sense from the perspective of the developers?

To address research question RQ_2, we applied all the algorithms mentioned above to the artificial god classes obtained by merging two, three, or four high quality classes, which we selected from each system listed in Table 1 and followed the steps of the algorithm describes in Fig. 13. We repeated the process 50 times and then compared the average refactoring accuracy rates with the three approaches. Furthermore, the MPC and CBO metrics were used to measure the coupling of refactored classes obtained by all the comparable algorithms. It is important to note that MPC is more suitable than CBO for evaluating the effects of move method refactoring. If k pairs of methods are called by each other between classes CL_X and CL_Y, and $k'(k' < k)$ methods are moved from CL_X to CL_Y, then the value of CBO will not be changed, but the value of MPC will be reduced by k'. The $Connectivity$ metric is the number of method pairs with invocation or sharing attribute relationships over the total number of method pairs for the class [48]. The $LCOM$ metric only considers the sharing attribute relationships, so we used both the $Connectivity$ and $LCOM$ metrics to evaluate the structural cohesion of the restructured classes.

To address research question RQ_3, we used the experimental data provided by Bavota et al. [15], which contains 11 meaningful extract class operations performed by the original developers. All of the extract class operations were identified by analyzing the sequential software releases using ReFinder [15]. ReFinder is a powerful tool developed by Prete et al., which can identify 63 types of refactoring operations, but unfortunately not the extract class operation. To solve this problem, the authors manually validated sets of the move method and move field refactorings found by ReFinder to identify the extract class refactoring operations performed by the original developers. More information about the experimental data can be found in Ref. [15]. We considered the partition produced by the original developers as the "gold standard" and we then calculated the $MoJoFM$ values after applying the refactoring algorithms. The similarity between the refactoring operations suggested by these algorithms and those made by the original developers were evaluated based on the $MoJoFM$ metric. From the viewpoint of developers, the refactoring suggestions are more meaningful if the $MoJoFM$ metric has a higher value.

6.2 Results and Analysis

All the simulations were performed on a personal computer with the following hardware environment: 3.7 GHz CPU, 12 GB memory, and a 1 TB HDD. The software operating environment was Windows 8.1 and the compiler platform was Eclipse 4.5.0. The optimal parameter configuration obtained in Section 4 was used by the proposed algorithm. We re-implemented the two refactoring approaches used in the comparison, and their main parameter settings and implementations are described briefly as follows.

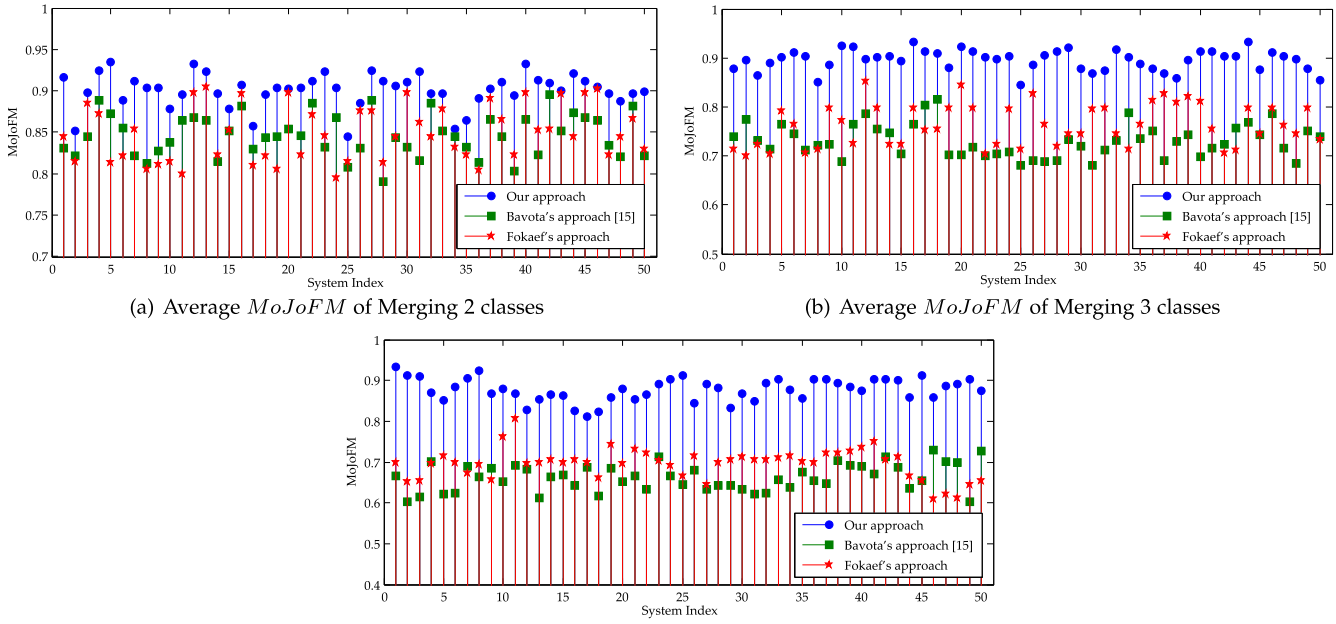
- Bavota's extract class refactoring algorithm [15]: According to [15], a given god class can be decomposed into more than two new classes. Bavota et al. used PCA to assign customized coefficients for the structural and semantic weights for each software system. The lightweight relationships between methods are removed according to the threshold, which is set as the median of the semantic similarities between methods.
- Fokaefs' extract class refactoring algorithm [23]: A hierarchical agglomerative algorithm was proposed by Fokaefs et al. for restructuring god classes, where the similarity between entities is defined as the $Jaccard$ distance according to the structural coupling relationships. In this approach, the stop condition for the hierarchical clustering algorithm requires that all the entities are merged into one cluster. Finally, the new concepts are identified as new classes for extraction based on the EP metric, which combines both coupling and cohesion measures [21].

6.2.1 Case Studies for RQ_2

Fig. 17 compares the $MoJoFM$ metric values obtained using the three approaches. Figs. 17a, 17b, and 17c show the average $MoJoFM$ metric values obtained by the refactoring operations when applied to the artificial god classes created by merging two, three, and four classes, respectively. Based on the refactoring results, we can make the following conclusions.

(1) In the three cases, the average $MoJoFM$ metric values obtained using our approach were always higher than those produced by the other algorithms. Obviously, our approach and Fokaefs' algorithm using hierarchical clustering performed better than Bavota's approach in terms of accuracy. Thus, the hierarchical clustering algorithm is more applicable to redistribute the functionalities of classes with code smells.

(2) The $MoJoFM$ values decreased as the number of classes for merging increased. Significantly, the



(a) Average $MoJoFM$ of Merging 2 classes

(b) Average $MoJoFM$ of Merging 3 classes

(c) Average $MoJoFM$ of Merging 4 classes

Fig. 17. Comparison of the $MoJoFM$ metric values obtained using the four approaches.

decrease in the accuracy obtained using our approach was less than that with the other two algorithms. The results also confirmed that our algorithm was not sensitive to the size of the god class and it delivered stable performance with different software systems. The differences in accuracy are explained as follows.

① Fokaefs et al. also used the hierarchical clustering algorithm to generate refactoring suggestions, but they did not weight the different types of structural coupling relationships when calculating the similarity between methods. In addition, the stop conditions for hierarchical clustering were different in our approach and Fokaefs's algorithm. We stopped clustering when the maximum modularity increment obtained by merging method pairs was less than 0. However, in Fokaefs et al.'s approach, the stop condition required that the entities were aggregated into one cluster. Thus, our proposed algorithm had better stability and higher precision.

② The coefficient of semantic coupling assigned by our approach was lower than that using Bavota's algorithm. In general, analyzing the semantic similarities between methods can improve the refactoring accuracy. However, when merging more classes, methods that implement different functionalities have a higher probability of being conceptually related to each other. In these cases, an excessively high semantic coefficient may have negative effects on the clustering analysis.

Tables 5 and 6 show the cohesion and coupling metrics obtained for the 11 god classes after performing the all the refactoring operations suggested by the three algorithms[8] and their changes compared with the original structure. We found that the value of the $LCOM$ metric decreased significantly after performing the refactoring operations suggested by the proposed approach. On average, the values of the $C3$ and $Connectivity$ metrics increased by 95 and 88 percent, respectively, compared with those for the original classes. Thus, the refactoring suggestions obtained by the proposed approach had a positive effect on the system, which improved the cohesion of the classes under the condition of controlling the overall coupling. The values of the conceptual cohesion metric $C3$ obtained by our approach were very close to those produced using Bavota's approach, which considers semantic measures as the major factors for identifying extract class refactoring opportunities. In addition, the other metrics obtained by the proposed approach were ranked in first place by a narrow margin, compared with the results of Bavota's algorithm.

In Tables 5 and 6, the execution time refers to all the steps in each algorithm. According to Day and Edelsbrunner [50], the time complexities for agglomerative hierarchical clustering algorithms is $O(|V|^3)$, however, as noted by Clauset et al. [38], the community detection algorithm adopted in the proposed approach runs far more quickly by exploiting some shortcuts in the optimization problem and using more sophisticated data structures. The time complexities of the clustering algorithms used in our approach, Bavota's approach and Fokaefs's approach were $O(|V|\log^2|V|)$ [38], $O(f \times g)$ and $O(|V|^3)$ [50], respectively, where $|V|$ denotes the number of entities to be clustered, $|E|$ denotes the number of edges between the entities, f is the number of methods in the trivial chains [15], and g is the number of methods in the non-trivial chains. The ascending order of time complexity was: $O(f \times g) < O(|V|\log^2|V|) < O(|V|^3)$. Calculating the semantic weight between methods requires the extraction of the vocabulary from Java files and the LSI algorithm is used to determine the conceptual similarity between methods, which is a time-consuming process. Thus, although the time complexity of the clustering algorithm used in Bavota's approach was lower than that of the

8. https://github.com/wangying8052/Raw-Data-in-Section-6

TABLE 5
Cohesion and Coupling Metric Values: Our Approach versus the Three Approaches Used for Comparison, Where L = $LCOM$ (Average), Z = $C3$ (Average), X = $Connectivity$ (Average), S = MPC (Total), H = CBO (Total), K = Time (s)

Class	Pre-refactoring					Our approach					
	L	Z	X	S	H	L	Z	X	S	H	K
Database	638	0.15	0.11	69	52	139 ↓78%	0.29 ↑93%	0.38 ↑245%	70 ↑1%	55 ↑6%	6.07
Select	43	0.27	0.35	36	12	12 ↓72%	0.33 ↑22%	0.44 ↑26%	38 ↑6%	16 ↑33%	3.09
UserManager	20	0.21	0.42	40	20	4 ↓80%	0.32 ↑52%	0.49 ↑17%	41 ↑3%	21 ↑5%	3.65
FileGeneratorAdapter	8	0.31	0.75	26	9	5 ↓38%	0.42 ↑35%	0.87 ↑16%	29 ↑12%	13 ↑44%	6.89
Import	29	0.10	0.20	24	17	4.5 ↓84%	0.25 ↑150%	0.28 ↑40%	25 ↑4%	19 ↑12%	6.69
JEditTextArea	12,876	0.11	0.23	141	36	523.5 ↓96%	0.26 ↑136%	0.26 ↑13%	162 ↑15%	53 ↑47%	357
JFreeChart	236	0.09	0.11	31	36	54 ↓77%	0.26 ↑189%	0.21 ↑91%	32 ↑3%	38 ↑6%	2.68
NumberAxis	102	0.12	0.15	16	13	19 ↓81%	0.28 ↑133%	0.29 ↑93%	17 ↑6%	18 ↑38%	2.96
DefaultAppLicationModel	81	0.14	0.09	4	13	24 ↓70%	0.26 ↑86%	0.22 ↑144%	5 ↑25%	14 ↑8%	5.77
XMLDTDValidator	6,695	0.11	0.13	192	60	1,620 ↓76%	0.24 ↑118%	0.30 ↑131%	200 ↑4%	68 ↑13%	325
XMLSerializer	95	0.20	0.11	45	15	34 ↓64%	0.27 ↑35%	0.28 ↑155%	50 ↑11%	25 ↑67%	7.80
Average	-	-	-	-	-	↓74%	↑95%	↑88%	↑8%	↑25%	-

TABLE 6
(Table 5 Continued) Cohesion and Coupling Metric Values: Our Approach versus the Three Used for Comparison, Where L = $LCOM$ (Average), Z = $C3$ (Average), X = $Connectivity$ (Average), S = MPC (Total), H = CBO (Total), K = Time (s)

Class	Bavota's approach [15]						Fokaefs's approach					
	L	Z	X	S	H	K	L	Z	X	S	H	K
Database	155 ↓76%	0.29 ↑93%	0.35 ↑218%	70 ↑1%	56 ↑8%	6.02	155.5 ↓76%	0.24 ↑60%	0.13 ↑18%	74 ↑7%	65 ↑3%	4.23
Select	10 ↓78%	0.37 ↑37%	0.43 ↑23%	40 ↑11%	17 ↑42%	2.91	6 ↓86%	0.31 ↑15%	0.36 ↑3%	40 ↑11%	15 ↑25%	0.53
UserManager	6.5 ↓68%	0.31 ↑48%	0.46 ↑10%	44 ↑10%	23 ↑15%	2.37	4.5 ↓78%	0.29 ↑38%	0.44 ↑5%	43 ↑8%	24 ↑20%	0.64
FileGeneratorAdapter	5 ↓38%	0.43 ↑39%	0.87 ↑16%	29 ↑12%	14 ↑56%	6.21	4 ↓50%	0.41 ↑32%	0.8 ↑7%	28 ↑8%	14 ↑56%	0.63
Import	5.5 ↓81%	0.26 ↑160%	0.29 ↑45%	25 ↑4%	19 ↑12%	6.27	6.5 ↓78%	0.22 ↑120%	0.26 ↑30%	28 ↑17%	21 ↑24%	3.12
JEditTextArea	721 ↓94%	0.26 ↑136%	0.25 ↑9%	169 ↑20%	56 ↑56%	322	3,471 ↓73%	0.21 ↑90%	0.3 ↑30%	172 ↑22%	61 ↑69%	85.8
JFreeChart	61 ↓74%	0.27 ↑200%	0.19 ↑73%	33 ↑6%	41 ↑14%	2.7	59.5 ↓75%	0.23 ↑156%	0.12 ↑9%	35 ↑13%	43 ↑19%	0.54
NumberAxis	35 ↓66%	0.28 ↑133%	0.26 ↑73%	18 ↑13%	18 ↑38%	2.74	23 ↓77%	0.19 ↑58%	0.16 ↑7%	19 ↑19%	20 ↑54%	0.47
DefaultAppLicationModel	20.5 ↓75%	0.26 ↑86%	0.27 ↑200%	5 ↑25%	14 ↑8%	4.91	17 ↓79%	0.19 ↑36%	0.11 ↑22%	6 ↑50%	17 ↑31%	0.53
XMLDTDValidator	2,313 ↓65%	0.22 ↑100%	0.22 ↑69%	202 ↑5%	68 ↑13%	261	1,679 ↓75%	0.26 ↑136%	0.28 ↑115%	203 ↑6%	70 ↑17%	44
XMLSerializer	34 ↓64%	0.26 ↑30%	0.24 ↑118%	48 ↑7%	25 ↑67%	7.52	48 ↓49%	0.23 ↑15%	0.17 ↑55%	48 ↑7%	25 ↑67%	1.69
Average	↓71%	↑97%	↑78%	↑10%	↑30%	-	↓72%	↑69%	↑27%	↑15%	↑35%	-

TABLE 7
$MoJoFM$ Between the Refactoring Suggestions Obtained by Four Approaches and Those Performed by the Original Developers

Class(TOF)	Ours versus dev.		Bavota et al. [15] versus dev.		Fokaefs et al. [23] versus dev.	
	$MoJoFM$	MJ	$MoJoFM$	MJ	$MoJoFM$	MJ
Database(41)	1	0	0.97	1	0.94	4
Select(14)	0.91	2	0.83	2	0.79	5
UserManager(13)	0.86	2	0.93	1	0.87	3
FileGeneratorAdapter(9)	0.86	1	0.86	1	0.76	2
Import(10)	1	0	1	0	0.65	9
JEditTextArea(214)	0.94	15	0.84	34	0.94	15
JFreeChart(24)	1	0	0.95	1	0.63	12
NumberAxis(20)	1	0	0.94	1	0.75	8
DefaultApplicationModel(14)	0.92	1	0.92	1	0.82	4
XMLDTDValiDator(69)	0.98	2	0.88	8	0.79	23
XMLSerializer(69)	0.97	1	0.91	2	0.89	4
Average	0.95	2.2	0.91	4.7	0.80	8.09

other algorithms, its total execution time was still longer than that of Fokaefs' algorithm. Time is required to calculate the four types of structural coupling weights between each method pair in a god class, so the total execution time of the proposed approach was ranked third.

6.2.2 Case Studies for RQ_3

Table 7 shows the refactoring results obtained by the three approaches, where MJ represents the number of move or join operations performed to transform the partitions obtained by the algorithms into the partitions made by the original developers. Clearly, the refactoring opportunities identified by our and Bavota's approaches were extremely similar to the operations performed by the original developers. The $MoJoFM$ metric obtained by our approach was slightly above that achieved by Bavota's approach and significantly higher than the results of Fokaefs' algorithm. Especially, the $MoJoFM$ metric values were 1 for the refactoring operations performed on classes $JFreeChart$, $NumberAxis$, and $Import$. This means that all of the refactoring operations suggested by our approach for these three classes made sense from the developer's viewpoint.

The MJ value obtained for the $JEditTextArea$ class using our approach was higher than that for the other classes. The $JEditTextArea$ class included 214 methods for implementing functions related to text editing and it was actually decomposed into three new classes by the comparable algorithms. We used the topic maps presented by Kuhn et al. [47] to visualize the function distributions of the restructured classes. As shown in Fig. 18, the $JEditTextArea$ class contained five main topics, i.e., $Line$ (line editing operations), $Caret$ (caret editing operations), $Scroll$ (scrolling text operations), $Selection$ (text selection operations), and $Drag$ (dragging text operations). The five axes in each topic map described the percentage of methods defined in the class for implementing the corresponding topic. Obviously, the distributions of the functions obtained by our approach were better than those of the original class and the other two algorithms mentioned above.

6.3 Threats to Validity

To perform a comprehensive comparison with previous methods, we re-implemented the baseline approaches and employed their corresponding parameter configurations from Section 5.2, which comprises a threat to the validity of our study. The raw data and experimental material are available online for replication is necessary.

Furthermore, we used the experimental data provided by [15], which are recognized as the meaningful refactoring operations suggested by the original developers. In their approach, Bavota et al. used ReFinder to identify the sets of move method or move field refactorings among successive releases, and then recognized them as extract class operations by manual validation. These mining processes may reduce the reliability of the extract class operations.

7 Evaluations of the Refactoring Solutions

In this section, we provide simulation results and evaluations for the proposed algorithm. System-level automatic refactoring was performed using the optimal coefficient settings of: $\alpha = 0.5, \beta = 0.2, \gamma = 0.2, \eta = 0.1$ and $Th_1 = SSW_{med}$, using five open source software systems, i.e., JHotDraw 7.0.6, JFreeChart 0.9.7, JEdit 2.7, HSQLDB 1.8.1.4 and Jmol 9.0, which are well known and they were also used by similar approaches [21], [22], [23]. We let $Th_2 = avg_o$. To ensure that the experimental data did not bias the results, these five systems were not included in the 50 training systems used for adjusting the parameter configuration, as described in Section 4. Furthermore, we evaluated the effectiveness of the refactoring solutions obtained from the perspectives of developers and based on metrics.

7.1 Evaluations by Experts

We asked 61 software quality evaluation experts and 39 masters/PhD students with an academic software engineering background to complete questionnaires,[9] with the aim of further addressing research question RQ_3.

7.1.1 Planning

All of the software quality evaluation experts had more than three years development experience and they worked for globally recognized companies, i.e., $Baidu$ Inc. (Nasdaq: BIDU), $Netease$ Inc. (Nasdaq: NTES), $Kingsoft$ Corporation

9. https://github.com/wangying8052/Questionnaire

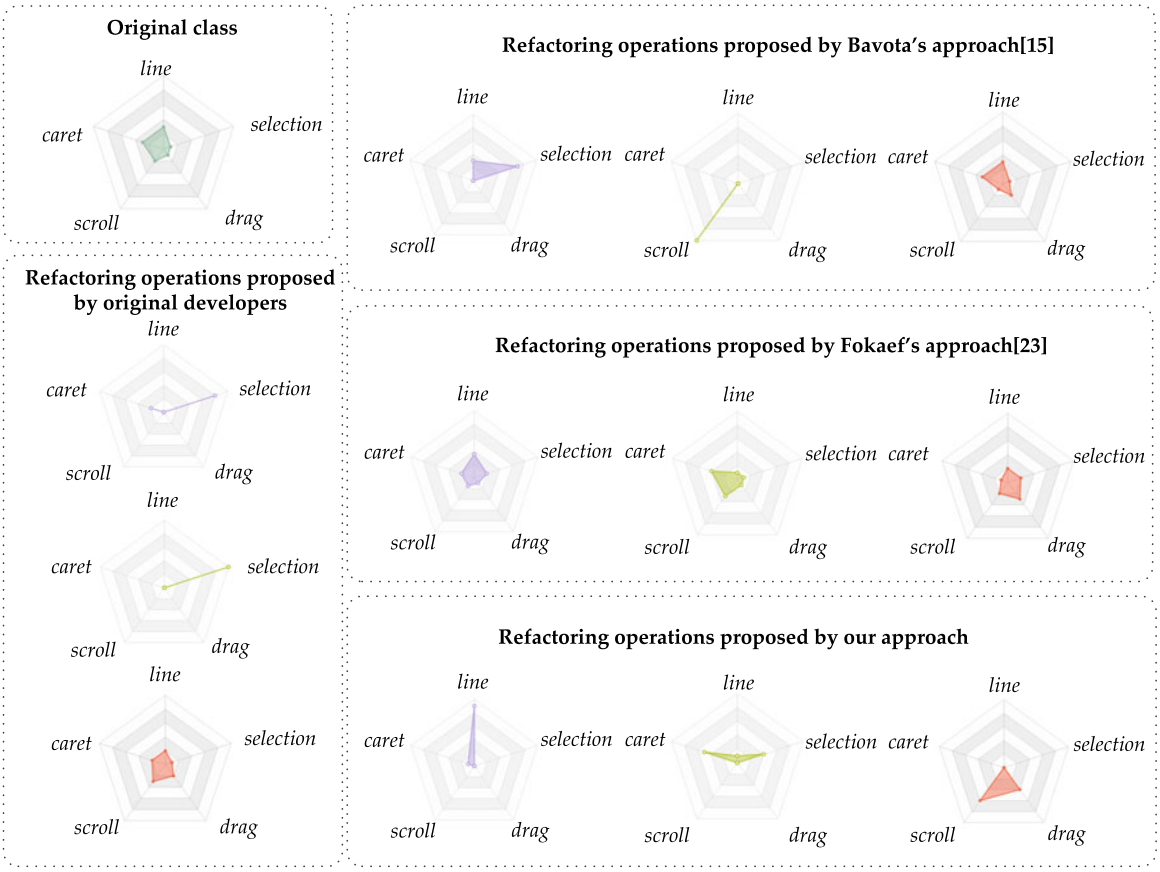


Fig. 18. Topic maps of the $JEditTextArea$ class before and after performing refactoring operations.

TABLE 8
Statistics Analysis of the Refactoring Results Obtained Using Our Proposed Approach

System	Bf/Af	$\|V\|$	N_{inh}	N_{noih}	N_{leaf}	$\|E_1\|$	$\|E_2\|$	$\|\overline{V_1}\|$	$\|\overline{E_1}\|$	$\|TR\|$	N_{extr}	N_{mvm}	N_{mvf}
JHotDraw 7.0.6	Bf	309	205	104	114	1,435	182	175	479	33	11	13	1
	Af	343	227	118	123	1,480	224	198	498	36			
JFreeChart 0.9.7	Bf	551	309	242	140	1,344	311	159	293	50	16	14	8
	Af	599	335	259	167	1,381	337	203	311	60			
jEdit 2.7	Bf	251	156	95	112	1,154	128	183	739	36	20	19	8
	Af	278	158	120	116	1,185	132	212	773	37			
HSQLDB 1.8.1.4	Bf	301	132	169	57	1,419	84	153	745	37	20	17	1
	Af	354	161	193	72	1,454	113	192	796	48			
Jmol 9.0	Bf	169	78	91	44	558	47	133	363	14	11	8	0
	Af	187	82	105	47	579	51	150	386	15			

(HKEX: 3888), *YonyouNetwork* Co. Ltd (SSE: 600588), and *Senyint* Co. Ltd. The masters and PhD students had participated in actual software development projects. Each of the participants was asked to evaluate the refactoring suggestions at different system granularities and to submit their assessment results by E-mail after analysis for two or three days. We used a three-point Likert scale for the assessment, where they had to give Yes/No/Maybe answers to the following questions for each refactoring suggestion.

Question 1 (Q_1): Does each of the refactored classes encapsulate a single function?

Question 2 (Q_2): If a tool can generate the suggestions automatically, would you apply the proposed refactoring?

Question 3 (Q_3): Does it improve the maintainability of the code?

To ensure the quality of the evaluation, we only used two systems for each subject; on average, each system was evaluated by 20 experts and 20 masters/PhD students.

7.1.2 Analysis of the Results

Table 8 shows the statistical analysis of the refactoring results,[10] where Bf/Af represents the status before or after refactoring; N_{inh} and N_{noih} denote the number of classes in the inheritance and noninheritance hierarchies, respectively; N_{leaf} is the number of leaf nodes in the inheritance

10. https://github.com/wangying8052/Raw-Data-in-Section-7

TABLE 9
Expert Evaluation Results

System	Refactoring	Q_1			Q_2			Q_3		
		Yes	No	Maybe	Yes	No	Maybe	Yes	No	Maybe
JHotDraw 7.0.6	Non-inheritance Move Method/Field	0.72	0.16	0.12	0.66	0.12	0.23	0.67	0.13	0.20
	Non-inheritance Extract Class	0.77	0.03	0.20	0.70	0.08	0.22	0.75	0.04	0.21
	Inheritance Extract Class	0.76	0.03	0.20	0.71	0.10	0.19	0.76	0.20	0.04
JFreeChart 0.9.7	Non-inheritance Move Method/Field	0.69	0.08	0.24	0.65	0.24	0.10	0.69	0.10	0.22
	Non-inheritance Extract Class	0.95	0	0.05	0.84	0.04	0.11	0.89	0.11	0
	Inheritance Extract Class	0.92	0.06	0.03	0.72	0.18	0.10	0.82	0.16	0.02
jEdit 2.7	Non-inheritance Move Method/Field	0.66	0.11	0.23	0.50	0.21	0.29	0.66	0.11	0.23
	Non-inheritance Extract Class	0.89	0.03	0.08	0.66	0.07	0.27	0.89	0.03	0.08
	Inheritance Extract Class	1.00	0	0	0.63	0	0.37	1.00	0	0
HSQLDB 1.8.1.4	Non-inheritance Move Method/Field	0.76	0.09	0.15	0.64	0.16	0.20	0.76	0.16	0.08
	Non-inheritance Extract Class	0.80	0.04	0.16	0.63	0.13	0.24	0.8	0.04	0.16
	Inheritance Extract Class	0.84	0.03	0.13	0.69	0.17	0.14	0.84	0.13	0.13
Jmol 9.0	Non-inheritance Move Method/Field	0.93	0	0.07	0.68	0.01	0.31	0.93	0	0.07
	Non-inheritance Extract Class	0.85	0.05	0.10	0.70	0.11	0.19	0.85	0.10	0.05
	Inheritance Extract Class	0.99	0.01	0	0.69	0.01	0.30	0.99	0.01	0

hierarchies; $|\overline{V}|$ and $|\overline{E_1}|$ are the number of nodes and edges in the residual network $|\overline{G_1}|$, respectively; and N_{extr}, N_{mvm}, and N_{mvf} denote the numbers of extract class, move method, and move field refactoring operations suggested by our approach, respectively.

As shown in Table 9, the evaluation results obtained for the five systems were quite similar. In all the cases, the number who selected "Yes" was far higher than that of those who chose "No" and "Maybe". All of the participants stated that 100 percent of the extract class suggestions in the non-inheritance related classes of JFreeChart 0.9.7 and the refactoring operations in the inheritance of jEdit 2.7 made sense, and 84 and 63 percent of the evaluators were willing to perform these two types of operations, respectively. Based on the feedback in the questionnaires, some of the move method/Field refactoring operations received high marks. In particular, most of the participants agreed that the attribute *View.recorder* as well as its *Getter* and *Setter* methods should be moved to the class *Macros*. As shown in Fig. 19, by analyzing the coupling relationships between the methods, we found that these moved entities were used mainly to deal with operations related to macros, as a result, they were frequently invoked by the methods defined in the class *Macros*. The non-inheritance refactoring suggestions for the JFreeChart 0.9.7 system were mostly approved by the subjects. Significantly, one evaluator commented that the restructuring operation for the class *chartUtilites* could be considered as the perfect division. As shown in Fig. 20, after the clustering analysis, the classes obtained, *chartUtilites_new*_1 and *chartUtilites_new*_2, encapsulated the functionalities of the PNG and JPEG image format settings, respectively. Moreover, for the jEdit 2.7 system, the god classes *XmlParser* and *Interpreter* contained 210 and 384 entities, respectively, and thus the evaluators suggested that they should be split to improve the readability of the code.

The evaluators agreed that the automatic tool could reduce the time required for refactoring, especially for inheritance because it is rather difficult to handle the relationships between classes based on observations. For the HSQLDB system, all of the participants considered that the restructuring suggestions for the class *HsqlProperties* and its subclass were reasonable. Fig. 21 shows the refactoring results. First, we extracted all the entities based on the functions related to "properties" into a new class *HsqlProperties_new*_2. Furthermore, all the methods in the class *HsqlDatabaseProperties* that could overwrite or invoke the super-methods defined in *HsqlProperties_new*_1 and *HsqlProperties_new*_2 were extracted as their corresponding subclasses *HsqlDatabaseProperties_new*_1 and *HsqlDatabaseProperties_new*_2, respectively. Finally, the class *HsqlDatabaseProperties_new*_3 was considered as a data class that encapsulated the attributes for recording the values in the fields.

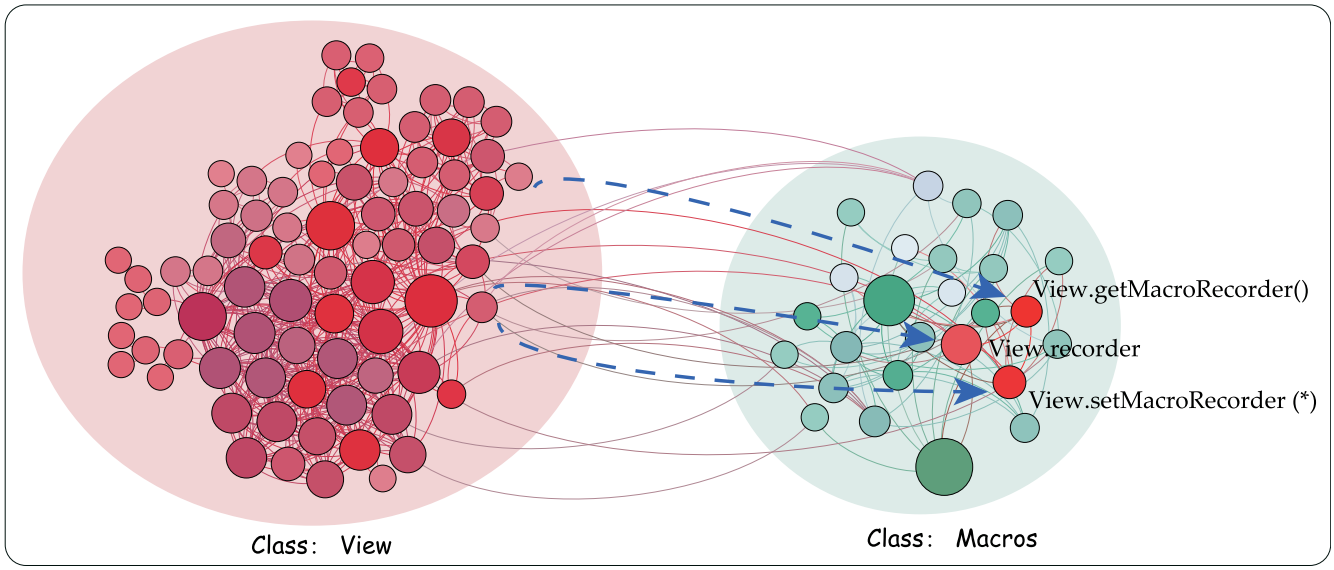


Fig. 19. Coupling relationships between the entities in the *View* and *Macros* classes.

The suggestions obtained were not perfect, but nearly 30 percent of unused suggestions were considered valuable for improving the maintainability of the system. In addition, 34 percent of the evaluators would not like to execute the suggested refactoring operations, although they considered that the extracted classes encapsulated single functions. For example, one participate explained that although the suggestions for class *DINameSpace* in HSQLDB system made sense, they preferred to keep the original code structure as its complexity was acceptable. In other words, to make a trade-off between the costs of refactoring operations and their benefits to the system, they did not adopt our suggestions.

Indeed, removing code smells caused by cohesion and coupling problems is not the only way to improve the system quality. In some cases, combining with the other types of refactoring operations can maximize the effectiveness. Four subjects commented that the duplicated codes between classes *BandPlotG2DRenderer* and *BandPlotEPSRenderer* in Jmol system should be pulled up to class *BandPlotRenderer* except splitting them into new classes. In addition, for class *HorizontalBarRenderer3D* in JFreeChart system, one expert agreed that before performing extract class restructuring operation, we should redistribute the functionalities at a finer granularity by Extract Method and Replace Parameter with Method refactoring [2], as the "Long Method" and "Long Parameter List" code smells existed in method *drawItem* (*Graphics2D*, *Rectangle2D*, *CategoryPlot*, *CategoryAxis*, *ValueAxis*, *KeyedValues2DDataset*, *int*, *int*, *int*).

Concerning the rejected suggestions, all the PhD students participated to our study approved that some refactoring operations could improve the system quality from the perspective of metrics; however, they were not meaningful from developer's view. That can be treated as the limitation existing in all the automatic software refactoring algorithms.

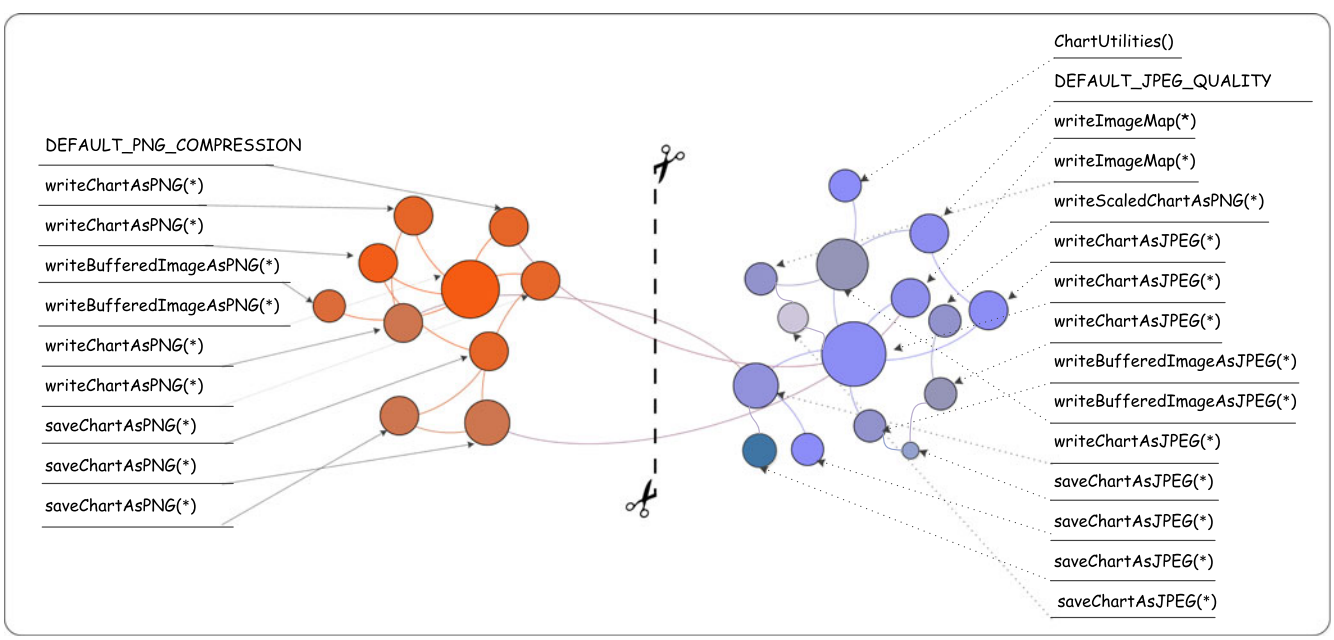


Fig. 20. Division of the *chartUtilites* class suggested by the proposed approach.

TABLE 10
Design Metrics for Design Properties in QMOOD and Maintainability Models [25], [26], [27], [36], [43]

Metric	Abbreviation	Definition
Design size	(DSC, design size in classes)	Total number of classes in the object-oriented software system
Abstraction	(ANA, average number of ancestors)(DIT, depth of inheritance for a tree in a class)	ANA and DIT metrics are considered as the quotients in QMOOD and maintainability models, respectively. Both of the metrics are equal to the maximum length from the node to the root in the inheritance tree
Encapsulation	(DAM, data access metric)	The ratio of the number of non-public methods relative to the total number of methods declared in the class
Coupling	(DCC, direct class coupling) (CBO, coupling between object classes)	The DCC metric is applied in the QMOOD model and the CBO metric is used in the maintainability model. Both of the metrics represent the number of classes that have direct coupling relationships with the class
Cohesion	(CAM, cohesion among methods in class)($LCOM$, lack of cohesion of methods)	The CAM metric applied in the QMOOD model is computed by summing the intersection of the parameters in a method and the maximum independent set of all parameter types in the class. The $LCOM$ metric used in the maintainability model is equal to the total number of method pairs not sharing attributes subtracted from the total number of method pairs sharing attributes with each other
Composition	(MOA, measure of aggregation)	The total number of data declarations with types that are user-defined classes
Inheritance	(NOC, number of children for a class)	It measures the total number of immediate subclasses for a class in the inheritance hierarchy
Polymorphism	(NOP, number of polymorphic methods)	The total number of methods that can be overridden by the methods declared in its subclasses
Messaging	(CIS, class interface size) (RFC, response sets for a class)	The CIS metric is considered to be the quotient in the QMOOD model and it is defined as the total number of public methods in a class. The RFC metric used in the maintainability model represents the union of the methods invoked by the class and all the methods declared in the class
Complexity	(NOM, number of methods) (WMC, weighted methods per class)	The NOM and WMC metrics are applied in the QMOOD and maintainability models, respectively. Both of the metrics are equal to the total number of methods declared in a class

A clear example is that the entities including $DATE\_FORMAT\_LONG$, $DATE\_FORMAT\_MEDIUM$, $DATE\_FORMAT$, $DATE\_FORMAT\_SHORT$ and $parseDay$ $(String)$ should not be moved from to class Day to class $Hour$, as they implemented the date-related operations. In fact, the imperfect suggestions obtained by the proposed approach were also considered as the key points to remove the code smells, as well as could simplify the remaining refactoring work.

7.2 Evaluations Based on Metrics

7.2.1 Research Questions and Experimental Design

Based on the experiment results discussed above, we formulated a new research question, as follows.

- RQ_4: In addition to solving the cohesion and coupling problems, do the refactoring operations improve the design quality of the code from other perspectives?

Improving the cohesion and coupling metrics is a prerequisite for refactoring, rather than the final goal. To respond to research question RQ_4, we used the metrics defined in QMOOD [36] to evaluate the improvement in the $Reusability$, $Flexibility$, and $Understandability$ of the system after restructuring [36]. Furthermore, the maintainability model [43] and EP metric [21] were also applied in our evaluation. As defined in Eqs. (11), (12), and (13), the $Reusability$ reflects the ability of a design to be reused in multiple contexts, which has a negative correlation with $Coupling$ and a positive correlation with $Cohesion$, $Messaging$, and $DesignSize$. $Flexibility$ is calculated as the weighted summation of $Encapsulation(+0.25)$, $Coupling(-0.25)$, $Composition(+0.5)$, and $Polymorphism$ $(+0.5)$. $Understandability$ represents the readability and intelligibility of the code. Obviously, $Abstraction$, $Coupling$, $Polymorphism$, $Complexity$, and $DesignSize$ have a negative influence on $Understandability$, whereas the properties containing $Encapsulation$ and $Cohesion$ positively influence the $Understandability$. The $maintainability$ model proposed by Dubey et al. [43] has a negative correlation with the design metrics including $Coupling$, $Complexity$, $Abstraction$, $Inheritance$, and $Messaging$. It is defined by Eq. (14), where the constant k_0 depends on the characteristics related to the software development process. In this study, we assumed that k_0 was equal to 1. Table 10 shows the detailed correspondences between the metrics and design properties

$$Reusability = -0.25 \times Coupling + 0.25 \times Cohesion + 0.5 \times Messaging + 0.5 \times DesignSize \quad (11)$$

$$Flexibility = 0.25 \times Encapsulation - 0.25 \times Coupling + 0.5 \times Composition + 0.5 \times Polymorphism \quad (12)$$

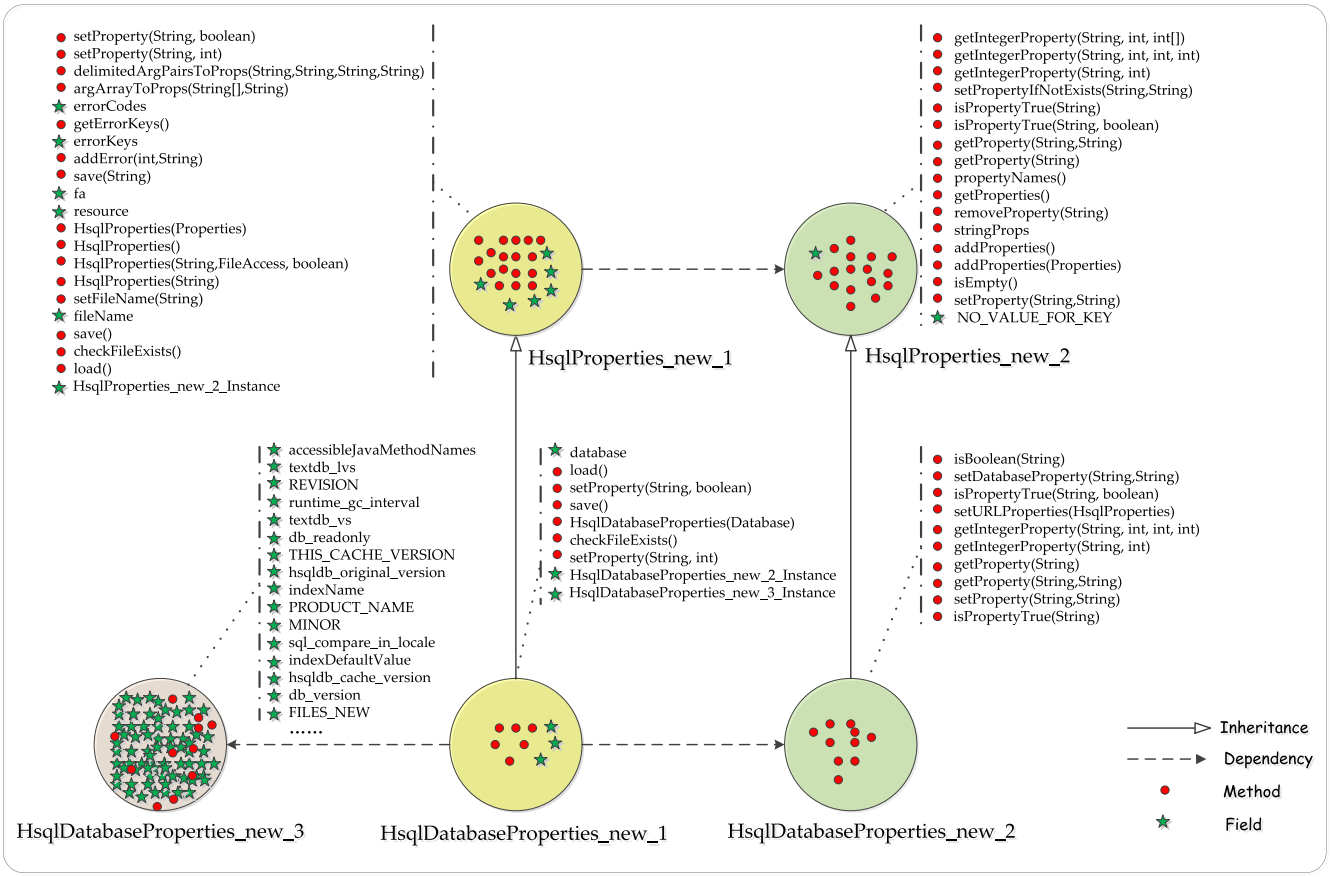


Fig. 21. Refactoring suggestions for the inheritance tree in the HSQLDB system.

$$\begin{aligned} Understandability = &-0.33 \times Abstraction - 0.33 \\ &\times Coupling + 0.33 \times Encapsulation + 0.33 \times Cohesion \\ &- 0.33 \times Polymorphism - 0.33 \times Complexity \\ &- 0.33 \times DesignSize \end{aligned} \tag{13}$$

$$\begin{aligned} Maintainability = k_0 \times 1/(&LCOM \times CBO \times WMC \\ &\times DIT \times NOC \times RFC). \end{aligned} \tag{14}$$

The EP metric [18], denoted as EP_{System}, can measure the cohesion and coupling of a system simultaneously. Eq. (15) shows the definition of EP_{System}, where $|CL_i|$ is the total number of the entities belonging to class $|CL_i|$ and EP_{CL_i} denotes the EP metric value of class $|CL_i|$, which is calculated by

$$EP_{\text{System}} = \sum_{|V|} \frac{|CL_i|}{\sum_{|V|} |CL_i|} \times EP_{CL_i} \tag{15}$$

$$EP_{CL_i} = \frac{\sum_{et_i \in CL_i} distance(et_i, CL_i)/|CL_i|}{\sum_{et_j \notin CL_i} distance(et_j, CL_i)/|entities \notin CL_i|}, \tag{16}$$

where $distance(et_i, CL_i)$ is the $Jaccard$ distance between entity et_i and class CL_i, and $|entities \notin CL_i|$ is the total number of the entities that do not belong to class CL_i. Thus, the definition of the EP metric can be understood as the ratio of the average inner entity distance relative to the average outer entity distance. Thus, the refactoring operations are more effective if the value of EP_{System} is lower.

7.2.2 Analysis of the Results

We compared the two approaches [29] and [30], which can also identity multiple refactoring opportunities, including move method, move field, and extract class. We re-implemented the algorithms and applied them to the five systems mentioned above. Moore's algorithm [29] was designed for the Self language, so the suggested refactoring operations may lead to multiple inheritances that are not supported in Java. Thus, only parts of the duplicated methods could be removed by move method refactoring. However, in the Snelting/Tip approach [30], all of the refactoring suggestions are generated based on the usage of the hierarchies. Therefore, the classes that are not directly or indirectly invoked by the given client programs cannot be restructured.

Tables 11 and 12 show the results obtained by the three system-level refactoring algorithms, where R_u, F_e, U_n, and M_a denote the $Reusability$, $Flexibility$, $Understandability$, and $Maintainability$, respectively. We normalized these metrics by calculating the ratio of the metric values obtained relative to the original values. Figs. 22a, 22b, 22c, 22d, and 22e show the changes in the metric quotient for the QMOOD and Maintainability models, and Fig. 23 shows the changes in quality for the five systems after performing the refactoring operations suggested by the three comparable algorithms. After comparing the evaluation metrics, we can make the following conclusions.

(1) The DIT and NOC metric values obtained by the proposed approach were decreased after refactoring. For the example described in Section 3.6, the

TABLE 11
Design Properties of the Three System-Level Refactoring Algorithms Used for Comparison

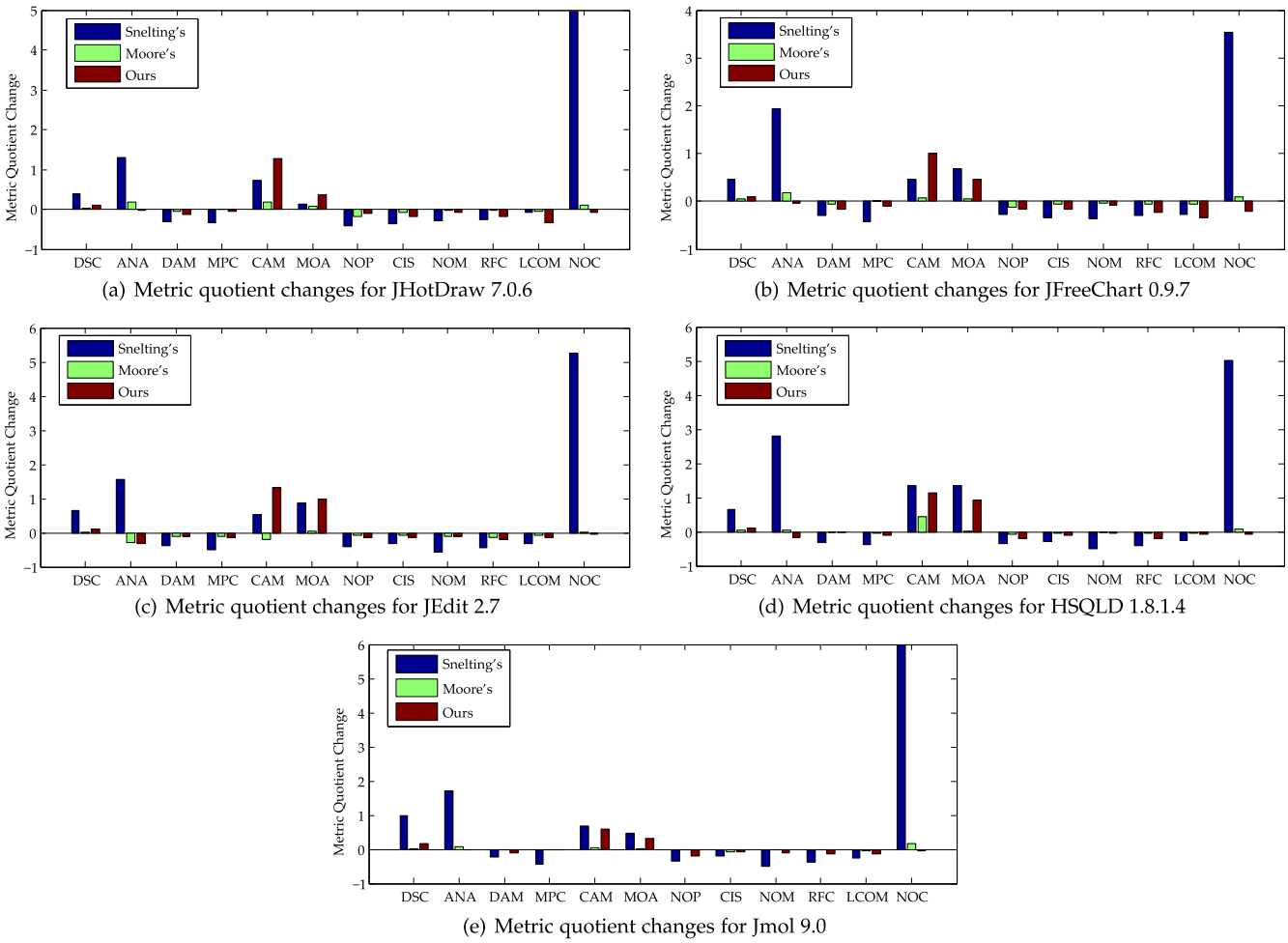
Software	Approach	*DSC*	*ANA* (*DIT*)	*DAM*	*MPC*	*CAM*	*MOA*	*NOP*	*CIS*	*NOM* (*WMC*)	*RFC*	*LCOM*	*NOC*
JHotDraw 7.0.6	Before	309	0.62	0.73	18.78	0.11	0.53	1.83	11.96	16.31	23.15	218.73	1.17
	Snelting's	428	1.42	0.50	12.38	0.19	0.60	1.07	7.54	11.80	17.01	201.32	6.98
	Moore's	317	0.73	0.69	18.83	0.13	0.57	1.49	11.10	16.02	22.42	207.55	1.30
	Ours	343	0.61	0.63	17.61	0.25	0.72	1.63	9.92	14.95	18.95	142.85	1.08
JFreeChart 0.9.7	Before	551	0.60	0.63	14.02	0.13	0.34	1.34	12.20	16.72	19.66	413.50	1.45
	Snelting's	801	1.76	0.44	7.99	0.19	0.57	0.96	8.04	10.75	13.71	298.68	6.56
	Moore's	573	0.71	0.60	14.07	0.14	0.36	1.17	11.42	16.01	18.32	390.43	1.59
	Ours	599	0.58	0.53	12.74	0.26	0.50	1.12	10.15	15.25	15.08	270.64	1.15
jEdit 2.7	Before	251	0.40	0.72	21.02	0.15	0.56	1.52	14.39	20.19	23.55	235.82	1.09
	Snelting's	414	1.03	0.44	10.43	0.23	1.05	0.90	9.67	9.02	12.97	157.90	6.82
	Moore's	256	0.29	0.64	19.09	0.12	0.59	1.40	13.46	17.83	20.34	219.04	1.11
	Ours	278	0.27	0.65	18.35	0.35	1.12	1.30	12.36	17.98	18.7	203.62	1.05
HSQLDB 1.8.1.4	Before	301	0.85	0.75	35.62	0.19	1.16	1.25	25.31	35.32	36.23	559.66	1.12
	Snelting's	602	2.31	0.59	20.12	0.32	1.72	0.83	20.11	17.23	21.99	423.40	7.81
	Moore's	309	0.91	0.73	34.87	0.20	1.17	1.25	23.2	34.96	35.43	530.77	1.29
	Ours	354	0.84	0.67	35.17	0.30	1.53	1.01	23.29	31.63	31.81	491.05	1.08
Jmol 9.0	Before	169	0.41	0.83	17.88	0.14	0.89	1.28	18.46	24.39	22.65	450.71	1.00
	Snelting's	281	1.56	0.56	11.06	0.33	2.09	0.85	13.21	12.03	13.09	328.14	6.01
	Moore's	176	0.43	0.81	17.20	0.20	0.90	1.20	17.50	23.96	21.86	437.00	1.08
	Ours	187	0.34	0.81	15.75	0.30	1.71	1.02	16.39	23.08	18.35	417.18	0.93

inheritance tree was decomposed from top to bottom. The class *DrawingPanel* was split into two new classes, *PerPanel_new_1* and *PerPanel_new_2*, with greater cohesion because the methods defined in its corresponding subclass *NetPanel* could only invoke or override the super-methods of *PerPanel_new_1*, so the class *PerPanel_new_2* had no subclasses after refactoring. Thus, the depth of the extracted inheritance tree and the average number of immediate descendants of the classes were reduced. There are no pull up/down refactoring operations in our approach, so the *DIT* and *NOC* metric values should not increase. However, the Snelting/Tip algorithm will introduce new subclasses if their methods are executed together, which improves their functional cohesion. In Moore's algorithm, to remove the duplicated method, we need to pull up the duplicated method to its corresponding superclass. Both of these operations made the *DIT* and *NOC* metric values higher. According to empirical studies, we conclude that the *DIT* and *NOC* metrics are good indicators of fault proneness [31], [32], [33]. A system becomes easier to understand and more maintainable during the software life cycle as the *DIT* and *NOC* metric values decrease. Thus, the values of U_n and M_a obtained by our approach were higher.

(2) Our approach identifies the refactoring opportunities based on the principle of "high cohesion and low coupling," so it is not surprising that the coupling and cohesion metrics for *MPC*, *RFC*, *CAM*, *LCOM*, and *EP* improved after performing the refactoring operations. There were a limited number of duplicated methods in the system and we removed some operations to avoid introducing multiple inheritances, so the quality changes caused by Moore's algorithm were less obvious. Compared with the Snelting/Tip algorithm, far fewer refactoring operations were suggested by our approach because the threshold Th_2 was used to control the restructuring efforts. However, the values of the metrics obtained by the proposed approach were better or similar to those obtained by the Snelting/Tip algorithm. Lower cohesion leads to more duplicate work and greater effort when reusing the system design [44]. The fault-proneness of a class is higher when the coupling between the software components is stronger [32]. Thus, lower cohesion and greater coupling can decrease the *Reusability*,

TABLE 12
Metrics for the QMOOD and Maintainability Models Obtained by the Three System-Level Refactoring Algorithms Used for Comparison

Software	Approach	R_u	F_e	U_n	M_a	*EP*	Time (min)
JHotDraw 7.0.6	Before	1.00	1.00	−0.99	1.00	0.92	-
	Snelting's	1.27	0.86	−1.07	0.23	0.80	64.60
	Moore's	1.03	1.02	−0.95	0.84	0.90	1.89
	Ours	1.29	1.07	−0.59	2.41	0.84	0.98
JFreeChart 0.9.7	Before	1.00	1.00	−0.99	1.00	0.91	-
	Snelting's	1.28	1.23	−1.37	0.41	0.83	50.89
	Moore's	1.01	0.95	−1.00	0.91	0.88	2.13
	Ours	1.17	1.09	−0.70	3.15	0.84	1.12
jEdit 2.7	Before	1.00	1.00	−0.99	1.00	0.93	-
	Snelting's	1.42	1.26	−1.19	0.76	0.85	27.98
	Moore's	0.95	0.98	−0.91	2.11	0.91	2.54
	Ours	1.27	1.36	−0.75	2.79	0.86	1.97
HSQLDB 1.8.1.4	Before	1.00	1.00	−0.99	1.00	0.95	-
	Snelting's	1.68	1.13	−1.31	0.42	0.84	25.35
	Moore's	0.99	1.00	−1.00	0.9	0.93	1.96
	Ours	1.20	1.03	−0.78	1.54	0.82	2.55
Jmol 9.0	Before	1.00	1.00	−0.99	1.00	0.86	-
	Snelting's	1.62	1.52	−1.39	0.34	0.8	19.33
	Moore's	1.03	0.98	−0.85	1.05	0.83	2.28
	Ours	1.23	1.33	−0.71	2.07	0.82	1.32



(a) Metric quotient changes for JHotDraw 7.0.6

(b) Metric quotient changes for JFreeChart 0.9.7

(c) Metric quotient changes for JEdit 2.7

(d) Metric quotient changes for HSQLD 1.8.1.4

(e) Metric quotient changes for Jmol 9.0

Fig. 22. Metric quotient changes after performing refactoring operations.

Flexibility, *Understandability*, and *Maintainability* of software. Consequently, all the system-level metrics were improved by our approach.

(3) If method m_i is identified as a move method refactoring opportunity, then all the classes that contain the invocations of m_i should add an instance variable of the target class for m_i. Thus, the MOA metric value will increase due to the move method/field refactorings. In our approach, to ensure that the refactored inheritance tree can invoke the extracted classes, we create instance variables for the extracted classes in the refactored classes. The *Flexibility* function defined in the QMOOD model demonstrates that the system design is more flexible when the MOA metric value is higher.

(4) Extract class refactoring operations were suggested by all three algorithms, and thus the DSC metric values increased in all cases. Therefore, the NOM metric, which is considered an indicator of complexity, clearly decreased. Obviously, the total number of

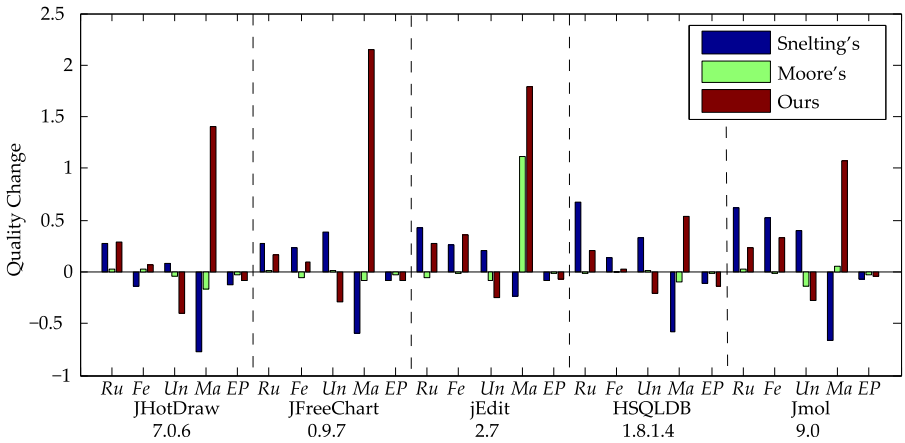


Fig. 23. Quality changes after performing refactoring operations.

new classes introduced by the Snelting/Tip algorithm was larger than that obtained by the other approaches because of its larger refactoring effects.

(5) Some classes are decomposed by refactoring, so the class size becomes smaller and the number of private (protected) attributes is reduced; thus, the DAM metric values decrease. However, the changes in the DAM metrics with our approach were smaller than those using the Snelting/Tip algorithm. This is because a higher weight is assigned to cluster the attributes and the methods that access them, and thus the attributes largely avoid being accessed by methods declared in other classes, and their visibility is not changed. In this manner, side effects on the $Flexibility$ and $Understandability$ of the software system are minimized.

(6) As the class size decreases, the average number of methods that can exhibit polymorphic behavior naturally becomes smaller. The decrease in the NOP metric improves the $Understandability$ and reduces the $Flexibility$ value. Moreover, the private methods should become visible to all the classes that depend on them if they are moved from the original class. Thus, the CIS metric values become lower, thereby decreasing the $Reusability$ according to the function defined in the QMOOD model. The least refactoring operations are performed using Moore's approach, which means that it has the fewest side effects on the $Flexibility$ and $Reusability$ of the code.

(7) Obviously, Moore's algorithm consumes the least time because it performs the fewest refactoring operations. In the Snelting/Tip algorithm, the time complexity for generating the concept lattice is $O(2^{|Ent|-1}|Obj||Ent|^2)$ [51] in the worst case, where $|Ent|$ is the total number of the entities in the system and $|Obj|$ is the total number of objects declared in the clients. The Snelting/Tip algorithm also takes the longest time because the process required to analyze the relationships between objects and entities is also time-consuming.

The simulation results indicated that the performance of our algorithm was highly stable with the different systems and its suggested refactoring operations were meaningful.

7.3 Threats to Validity

Three types of subjects completed our questionnaires, i.e., Master students, PhD students and professionals, which considered as the junior, intermediate and senior software quality evaluators, respectively. As the subjects were not the original developers of the object systems, they might not fully understand the source codes. To mitigate this threat, we reserved 2 or 3 days for participants to perform rich analysis of refactoring results, and all the students have received training about the software refactoring techniques. Fortunately, the survey results provided by these three types of subjects were similar, so that they had important reference values in evaluating the usability of the proposed refactoring tool.

Nevertheless, evaluation results by experts may be subjective to some extent. To avoid bias, we design the questionnaire following the principles proposed by Bavota et al. [22] and Stone [52]:

1) Providing the objective answer options for participants, including "Yes", "No" and "Maybe". Also, they could add an optional comment explaining the rationality behind each score.
2) We did not show the goal of our experimentation during the investigation to avoid suggestive behaviors.
3) No conformity and authority effects on the evaluation results, as the evaluators submitted answers via E-mails without discussion. Thus, neither participant knows the results of others.

Based on the above considerations, we can say that the subjects not tried to please the experimenters even though they provided the positive results.

We had to re-implement the algorithms compared in this study because they are no longer active projects. It should be noted that Moore's algorithm was designed for the Self language, but we applied it to Java projects by removing the refactoring operations that introduce multiple inheritances. These changes may have affected the refactoring results obtained. To ensure a fair comparison, we removed as many duplicated methods as possible but without changing that the code's behavior.

8 Conclusions

In this study, we proposed a refactoring algorithm based on complex network theory, which obtains the optimal functionality distribution from a system viewpoint. This approach combines three types of refactoring, i.e., move method, move field, and extract class, to remove the "bad smells" caused by cohesion and coupling problems associated with both inheritance and non-inheritance hierarchies. The software system is described by a class-level multi-relation directed network and method-level weighted undirected networks. We complete the refactoring preprocessing using the former, whereas the latter is combined with a weighted clustering algorithm to perform refactoring operations according to the principle of "high cohesion and low coupling." The similarity between the methods is equal to the weighted summation of the four types of coupling relationships, i.e., sharing attribute, method invocation, semantic relevance, and functional coupling. To obtain a more general parameter configuration, we used 50 systems with good designs from GitHub to tune the four types of coupling coefficients. We proposed a flexible mechanism to allow developers to balance the system benefits against the refactoring costs. Finally, the functions mentioned above were encapsulated in an executable tool, which allows users to perform refactoring operations automatically.

To verify the validity of the proposed approach, we performed comparisons with similar approaches. Furthermore, we considered the refactoring operations performed by the original developers as the "gold standard" and we evaluated whether the proposed refactoring suggestions made sense from a developer's viewpoint. System-level metrics for the $Reusability$, $Flexibility$, and $Understandability$ functions defined in the QMOOD and maintainability models were also used to evaluate the refactoring effects. Automatic refactoring experiments were conducted using five open source software systems, i.e., JHotDraw, JFreeChart, JEdit, HSQLDB, and Jmol. Lists of refactoring suggestions were obtained by comparing the structure of the system before and after performing the refactoring operations. In total,

40 PhD and masters students as well as 60 professional software quality evaluators from five globally recognized companies, i.e., *Baidu*, *Netease*, *Kingsoft*, *Yonyou*, and *Senyint*, completed questionnaires to evaluate the effectiveness of the refactoring algorithm. The assessment results demonstrated that the proposed approach can resolve cohesion and coupling problems without changing the external behavior of the code, as well as helping to improve the understandability, flexibility, reusability, and maintainability of code.

Acknowledgments

The authors gratefully acknowledge all the students and experts who participated to our study and all the reviewers for their positive and valuable comments and suggestions regarding our manuscript. We improved the original version of this paper according to their high-quality feedback. This research was supported by the National Natural Science Foundation of China (Grant Nos. 61374178, 61402092), The online education research fund of MOE research center for online education, China (Qtone education, Grant No. 2016ZD306) and the Ph.D. Start-up Foundation of Liaoning Province, China (Grant No. 201501141). Hai Yu is the corresponding author.

References

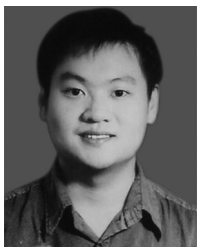
[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
[3] M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodology*, vol. 23, no. 4, pp. 33:1–33:39, Aug. 2014.
[4] L. M. Hakik and R. E. Harti, "Measuring coupling to evaluate the quality of a remodularized software architecture result of an approach based on formal concept analysis," *Int. J. Comput. Sci. Netw. Secur.*, vol. 14, no. 1, pp. 11–16, Jan. 2014.
[5] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, Jun. 2011.
[6] S. Mancoridis, B. S. Mitchell, and C. Rorres, "Using automatic clustering to produce high-level system organizations of source code," in *Proc. 6th Workshop Program Comprehension*, Jun. 1998, pp. 45–52.
[7] A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock, "Spectral and meta-heuristic algorithms for software clustering," *J. Syst. Softw.*, vol. 77, no. 3, pp. 213–223, Sep. 2005.
[8] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering*, Berlin, Germany: Springer, 2014, pp. 387–419.
[9] D. I. K. Sjoberg, A. Yamashita, B. C. D. Anda, and A. Mockus, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2012.
[10] S. G. Ganesh, T. Sharma, and G. Suryanarayana, "Towards a principle-based classification of structural design smells," *J. Object Technol.*, vol. 12, no. 2, pp. 1:10–1:29, Jun. 2013.
[11] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 220–235, Jan./Feb. 2012.
[12] S. Kimura, Y. Higo, and H. lgaki, "Move code refactoring with dynamic analysis," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sep. 2012, pp. 575–578.
[13] B. D. Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *Proc. 11th Working Conf. Reverse Eng.*, Nov. 2004, pp. 144–151.
[14] G. Bavota, A. D. Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Syst. Softw.*, vol. 84, no. 3, pp. 397–414, Mar. 2011.
[15] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation," *Empirical Softw. Eng.*, vol. 19, no. 6, pp. 1617–1664, May 2014.
[16] G. Czibula and I. G. Czibula, "Clustering based adaptive refactoring," *J. WSEAS Trans. Inf. Sci. Appl.*, vol. 7, no. 3, pp. 391–400, Mar. 2010.
[17] M. Bowman, L. C. Briand, and Y. Labiche, "Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 817–837, Nov./Dec. 2010.
[18] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent GA," *Softw.: Practice Experience*, vol. 41, no. 5, pp. 521–550, Apr. 2011.
[19] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. 9th Annu. Conf. Genentic Evol. Comput.*, Jul. 2006, pp. 1909–1916.
[20] A. Han, D. Bae, and S. Cha, "An efficient approach to identify multiple and independent move method refactoring candidates," *Inf. Softw. Technol.*, vol. 59, pp. 53–66, Mar. 2015.
[21] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–366, May/Jun. 2009.
[22] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, Jul. 2014.
[23] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorigou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, pp. 2241–2260, Oct. 2012.
[24] G. Bavota, A. D. Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in Eclipse: The ARIES project," in *Proc. 34th IEEE Int. Conf. Softw. Eng.*, Jun. 2012, pp. 1419–1422.
[25] M. O'Keeffe, and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *J. Syst. Softw.*, vol. 81, no. 4, pp. 502–516, Apr. 2002.
[26] M. O'Keeffe, and M. Ó Cinnéide, "Search-based software maintenance," in *Proc. 10th Eur. Conf. Softw. Maintenance Reengineering*, Mar. 2006, pp. 10–19.
[27] M. O'Keeffe and M. Ó Cinnéide, "Getting the most from search-based refactoring," in *Proc. 9th Annu. Conf. Genentic Evol. Comput.*, Jul. 2007, pp. 1114–1120.
[28] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.
[29] I. Moore, "Automatic inheritance hierarchy restructuring and method refactoring," in *Proc. 11th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, Oct. 1996, vol. 31, no. 10, pp. 235–250.
[30] M. Strekenbach and G. Snelting, "Refactoring class hierarchies with KABA," in *Proc. 18th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, Oct. 2004, vol. 39, no. 10, pp. 315–330.
[31] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.
[32] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, Jan. 1999.
[33] R. Harrison, S. Counsell, and R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *J. Syst. Softw.*, vol. 52, no. 3, pp. 173–179, Jun. 2000.
[34] C. R. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E*, vol. 68, no. 4, pp. 1–16, Oct. 2003.
[35] A. K. Sharma, A. Kalia, and H. Singh, "Metrics identification for measuring object oriented software quality," *Int. J. Soft Comput. Eng.*, vol. 2, no. 5, pp. 255–258, Nov. 2012.
[36] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.
[37] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *Proc. 21th IEEE Int. Conf. Softw. Maintenance*, Sep. 2005, pp. 133–142.

[38] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 69, no. 2, pp. 066111:1–066111:6. Dec. 2004.
[39] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, pp. 026113:1–026113:15, Feb. 2004.
[40] E. Thébault and C. Fontaine, "Stability of ecological communities and the architecture of mutualistic and trophic networks," *Science*, vol. 329, no. 5993, pp. 853–856, Aug. 2010.
[41] W. F. Opdyke, "Refactoring object-oriented frameworks," PhD dissertation, Dept. Comput. Sci., Univ. of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.
[42] Z. H. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proc. 12th IEEE Int. Workshop Program Comprehension*, Jun. 2004, pp. 24–26.
[43] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 5, pp. 1–7, Sep. 2011.
[44] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 629–639, Aug. 1998.
[45] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[46] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993.
[47] A. Kuhn, S. Ducasse, and T. Girba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007.
[48] J. M. Bieman and B. K. Kang, "Cohesion and reuse in an object-oriented system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, pp. 259–262, Aug. 1995.
[49] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 9, pp. 1124–1137, Sep. 2004.
[50] W. H. E. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *J. Classification*, vol. 1, no. 1, pp. 7–24, Dec. 1984.
[51] S. O. Kuznetsov and S. A. Obiedkov, "Comparing performance of algorithms for generating concept lattices," *J. Exp. Theoretical Artif. Intell.*, vol. 14, no. 2, pp. 189–216, Nov. 2002.
[52] D. H. Stone, "Design a questionnaire," *British Med. J.*, vol. 307, no. 6914, pp. 1264–1266, 1993.

Ying Wang is working toward the PhD degree in the Software College, Northeastern University, China. Her main research interests include software refactoring, software architecture, software testing, and complex networks.

Hai Yu received the BE degree in electronic engineering from Jilin University, China, in 1993 and the PhD degree in computer software and theory from Northeastern University, China, in 2006. He is currently an associate professor of software engineering with Northeastern University, China. His research interests include complex networks, chaotic encryption, software testing, software refactoring, and software architecture. At present, he serves as an associate editor of the *International Journal of Bifurcation and Chaos*, guest editor of the *Entropy*, and guest editor of the *Journal of Applied Analysis and Computation*. In addition, he was a lead guest editor for *mathematical problems in engineering* during 2013. Moreover, he has served different roles at several international conferences, such as associate chair for the 7th IWCFTA in 2014, program committee chair for the 4th IWCFTA in 2010, Chair of the Best Paper Award Committee at the 9th International Conference for Young Computer Scientists in 2008, and Program committee member for the $3\mathrm{rd}-9$th IWCFTA and the 5th Asia Pacific Workshop on Chaos Control and Synchronization.

Zhiliang Zhu received the MS degree in computer applications and the PhD degree in computer science from Northeastern University, China. His main research interests include information integration, complexity software systems, network coding and communication security, chaos-based digital communications, applications of complex network theories, and cryptography. He has authored and co-authored more than 130 international journal papers and 100 conference papers. In addition, he has published five books, including *Introduction to Communication* and *Program Designing of Visual Basic .NET*. He is also the recipient of nine academic awards at national, ministerial, and provincial levels. He has served in different capacities at many international journals and conferences. Currently, he serves as co-chair of the 1st-9th International Workshop on Chaos-Fractals Theories and Applications. He is a senior member of the Chinese Institute of Electronics and the Teaching Guiding Committee for Software Engineering under the Ministry of Education. He is a fellow of the China Institute of Communications and a member of the IEEE.



Wei Zhang received the PhD degree in computer science and technology from Northeastern University, China, in 2013. He currently works as an associate professor in the Software College, Northeastern University. His research interests include signal processing, multimedia coding, software refactoring, and software architecture.

Yuli Zhao received the PhD degree in communication and information systems from Northeastern University, China, in 2013. She currently works as a lecture with Northeastern University. Her research interests include applications of complex-network theories to communications, software refactoring, software architecture, and software testing.