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, reorganizing 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].

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, reorganizing 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].
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 GitHub1 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 based 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 tool2 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
Another interesting study by Bavota et al. used the measures delivered superior performance. Fokaefs et al. proposed an 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.

### 2.2 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 MaxFlow-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.

### 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 in 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 Snelling/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-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 METHODODOLOGY

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 \( V = \{CL_1, CL_2, \ldots, CL_N\} \), \( E = \{e_1, e_2, \ldots, e_{|E|}\} \), \( CL_i \in V \) 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 \( V = \{m_1, m_2, \ldots, m_M\} \), \( E = \{e_1, e_2, \ldots, e_{|E|}\} \), \( m_i \in V \) 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 \( E_1 \cup E_2 \cup \cdots \cup E_n = E \), and \( E_1 \cap E_2 \cap \cdots \cap E_n = \emptyset \). If \( \exists e_j \in E \), we have \( G_1 = (V, E_1) \), \( G_2 = (V, E_2) \), \ldots, \( G_n = (V, E_n) \), and thus \( G \) is referred to as the CMDN. The properties of a CMDN are described as follows.

Property 1. Suppose that \( E_i \) 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 \( e_{ab} \in E_i \) 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 \( G = G_1 \cup G_2 \), 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 \( M_i = \{m_1, m_2, \ldots, m_n\} \) and \( A_i = \{a_1, a_2, \ldots, a_n\} \), where \( m_n \in M_i \) represents any method defined in class \( CL_i \), and \( a_n \in A_i \) denotes any attribute defined in class \( CL_i \), then it follows that \( CL_i = M_i \cup A_i \).

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
(1) Construct network \( G = G_1 \cup G_2 \) 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 \( G_D = (V, E_D) \), where \( V = V \setminus V_D \), and \( E_D = E_1 \setminus E_D \). The edge set connected to the node in \( V_D \).

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

(5) If we merge all the classes in each connected component into an entity set \( \Omega_k \) 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 \( \Omega_k = \cup CL_m \), 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 \( \Psi \) 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 \( G' = G_1 ' \cup G_2 ' \) 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 \( CL_{root} \). If there is a directed path from node \( CL_{mn} \) to \( CL_{mr} \), then we say that node \( CL_{mn} \) can reach \( CL_{mr} \). The nodes in \( G_{2'} \) that can reach the root node \( CL_{root} \) comprise an inheritance tree \( TR_k \). Obviously, one root node \( CL_{root} \) 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 \( G' = G_1 ' \cup G_2 ' \) 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 \( G_D = (V', E_D) \), where \( V = V \setminus V_D \) and \( E_D = E_1 \setminus E_D \). The edge set connected to the nodes in \( V_D \).

(3) We find all the connected components in \( G_D \), 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 \( TR_i \in TR \), and the relationships between its elements can be described as a method-level weighted undirected network. To consider all 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.
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_k, 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_i 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

\[
SAW(m_i, m_j) = \begin{cases} 
\frac{|M_i \cap M_j|}{|M_i| + |M_j|} & \text{if } |M_i \cup M_j| \neq 0 \\
0 & \text{otherwise},
\end{cases}
\]  

(1)

where M_i and 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.

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

(2)

\[
MIW(m_i, m_j) = \max(MIW_{ij}, MIW_{ji}).
\]  

(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

\[
FCW(m_i, m_j) = \begin{cases} 
\frac{ET_i}{ET_i + ET_j} & \text{if } ET_i + ET_j \neq 0 \\
0 & \text{otherwise},
\end{cases}
\]  

(4)

where ET_i represents the number of functional domains in which methods m_i and m_j appear together, and ET_j is defined as 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 \( \bar{m_i} \) and \( \bar{m_j} \), respectively. If we let \( ||\bar{m_i}|| \) be the Euclidean norm of vector \( \bar{m_i} \), 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.

\[
SSW(m_i, m_j) = \frac{\bar{m_i} \cdot \bar{m_j}}{||m_i|| \cdot ||m_j||}
\]  

(5)
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 $\alpha + \beta + \gamma + \eta = 1$.

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

An example code is shown in Fig. 3. Clearly, according to the refactoring preprocessing operations described in [14], the required code level information can be extracted. We use the thresholds $Th_1$ to remove the pseudo-semantic relationships with weights lower than $Th_1$.

An example code is shown in Fig. 3. Clearly, according to the refactoring preprocessing operations described in [14], the required code level information can be extracted. We use the thresholds $Th_1$ to remove the pseudo-semantic relationships with weights lower than $Th_1$.
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 $G_1$ by deleting superclasses $\text{DrawingPanel}$ and $\text{PertPanel}$. Furthermore, the connected components $cc_1$ and $cc_2$ can be found in $G_1$, where $cc_1 = \{\text{RestoreDataEdit}, \text{NetPanel}, \text{UndoRedoManager}\}$, and $cc_2 = \{\text{SVGPanel}\}$. Obviously, there are 10 attributes and 23 methods in entity set $V_1$, 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 $Th_j$ 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

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

where $M$ is the number of the edges in the network, $A = (a_{ij})$ 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 $A_{sas}$; Method invocation adjacency matrix $B_{inv}$; Simultaneous execution adjacency matrix $C_{exe}$; Semantic relevance adjacency matrix $D_{sem}$.

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 $A' = \alpha \cdot A_{sas} + \beta \cdot B_{inv} + \gamma \cdot C_{exe} + \eta \cdot D_{sem}$. The methods that should not be split are bound into one community, and each of the other nodes $n_d \in V$ is considered as a community. We assume that network $G$ is divided into $|C|1$ communities, and then the community set can be described as $C = \{CN_1, CN_2, ..., CN_{|C|1}\}$. Let the weighted modularity $Q = 0$.

2. According to Equation (9), the element $\Delta Q_{ij}$ of the initial modularity increment matrix $A_Q$ is calculated as follows:

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

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

\[
\Delta Q_{mn} = \begin{cases} 
\frac{W_k}{W} - \left(\frac{S_k}{2W}\right)^2 & \text{if } CN_m \text{ connects with } CN_{min} \\
\frac{\Delta Q_{mn}}{2W} + \Delta Q_{mn} & \text{if } CN_m \text{ connects with } CN_{mn} \\
\frac{\Delta Q_{mn}}{2W} - \left(S_k \times S_i / 2W\right)^2 & \text{if } CN_m \text{ connects with } CN_{min} \text{ and not connects with } CN_{mn}
\end{cases}
\]

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 agglomerate the community pair containing the entities of different classes whose corresponding $\Delta Q_{mn}$ 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.

\[
Q = \frac{1}{2W} \sum_{ij} \left( w_{ij} - \frac{S_i \times S_j}{2W} \right) \times \delta(CN_i, CN_j)
\]

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 $n_{di}$, $w_{ij}$ is the weight of the edge between nodes $n_{di}$ and $n_{d_j}$, $n_c$ 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

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

where $S_{in}(CN_i, CN_j)$ 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 $\Delta Q_{ij}$ of the modularity increment matrix $A_Q$. If $\Delta Q_{mn}$ is the maximum element in $A_Q$, then communities $CN_{min}$ and $CN_{mn}$ 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 $0 \leq \Delta Q_{\text{mod}} \leq T_{\text{hi}}$, then we merge $CN_m$ and $CN_n$, where $T_{\text{hi}}$ represents the threshold used to identify more important move method/field opportunities. In the GUI of the proposed tool, these values can be assigned to $T_{\text{hi}}$ to allow developers to control the restructuring costs, i.e., $0$, $\text{avg}_0$, and $\text{max}_0$, where $\text{avg}_0$ and $\text{max}_0$ 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, $\text{avg}_0$ and $\text{max}_0$ are considered as two types of coupling strengths designed by original developers for the original system structure. If $T_{\text{hi}}$ is higher, then the clustering division is closer to the original design, and thus the refactoring expense is reduced.

3.3 Automatic Refactoring 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, i.e., $\text{avg}_0$. Similarly, if a method that overrides an inherited method is moved out of the superclass $CL_i$, then the method will be changed if the method that overrides an actual inherited method is moved out of the subclass $CL_j$.

(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 is 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 $CL_i$, 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 $CL_j$. 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 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 the entity set.
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, 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 $CL_{k}^{root} \in V_{root}$ represents any root node of an inheritance tree.

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

1. Create the new subclass set $V_{new}^{k}$ that corresponds to each new class extracted from the root node $CL_{k}^{root}$, and it follows that $|V_{new}^{k}| = |V_{new}^{k}|$. Initially, let $CL_{sub}^{k} = M_{i}^{0} \cup M_{i}^{0}$ for all $CL_{sub}^{k} \in V_{new}^{k}$, $i \in [1, 2, \ldots, k]$. 

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 $CL_{i}$ 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 $NL = 3$. 

Fig. 6. Refactoring process for the non-inheritance hierarchies.
Algorithm 2. The teasing apart inheritance algorithm

Input: Inheritance network $G_c$. 
Output: Restructured inheritance network $G'_c$.

1. All the entities defined in the root node $CL_{k_{1}}^{new} \in V_{root}$ and relations between them are considered as the method-level weighted undirected network, where $k \in [1, 2, \ldots, |V_{root}|]$. On this basis, the clustering analysis is used to regroup class $CL_{root}$, then:
   ① If class $CL_{k}^{new}$ was not decomposed, we should delete the node $CL_{k}^{root}$ and all the edges connecting to it from network $G_c$. Here, we denote $V_{str}^m$ as the direct subclass set of class $CL_{k_{1}}^{new}$. It follows that the direct subclasses in set $V_{str}^m$ become the root nodes.
   Let $V_{root} = V_{root} \setminus CL_{k}^{root}$, and repeat step 1.
   ② Class $CL_{k_{1}}^{root}$ should be subjected to Extract Class if it was split by clustering analysis. The new class set extracted from $CL_{k_{1}}^{new}$ is denoted as $V_{new}$, and the restructured root node is represented as $CL_{k_{1}}^{new}$. Obviously, each class of $V_{str}^m$ is considered as a root of the extracted inheritance tree.

2. Create the instance variables of classes belonging to set $V_{str}^m$ in class $CL_{k_{1}}^{new}$. Consequently, the extracted inheritance trees can be invoked by the main inheritance root node at class $CL_{k}^{new}$.

3. We split the direct subclasses of $CL_{k_{1}}^{new}$ according to the structure of the new extracted classes and the principle of “high cohesion and low coupling”.

4. If $CL_{k_{1}}^{new} \in V_{str}^m$ has direct subclasses, then we repeat step 3, where $m \in [1, 2, \ldots, |V_{str}^m|]$. The process continues, and the refactoring operations stop when all the leaf nodes have been analyzed. Finally, we obtain the restructured inheritance network $G'_c$ by updating $G_c$.

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

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 should be subjected 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 $Th_2 = av_{go}$. The values of $\alpha$, $\beta$, $\gamma$, and $q$ in Eq. (6) were assigned as 0.5, 12, 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 0.28 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$new_2$. Furthermore, the entities had closer relationships with the UndoRedo.
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 RestoreData\_redoEdit\_{...}, respectively. Consequently, these three methods were clustered together in the first few steps.

Fig. 8. Refactoring process for the inheritance hierarchies.
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 \( \text{UndoRedoManager.undoOrRedo(...)}, \text{UndoRedoManager.undoOrRedoInProgress} \), and \( \text{UndoRedoManager.updateActions(...)} \) 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 \( \text{NetPanel} \) 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 large value.
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 $a_{_j}.get\_set()$ 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
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 \( \alpha, \beta, \gamma, \) and \( \eta \), which represent the coefficients of \( SAW, MIW, FCW, \) and \( SSW \) in Eq. (6), respectively. We addressed the following research question in our case study.

- **RQ1**: How can we assign the values of \( \alpha, \beta, \gamma, \) and \( \eta \) to make them applicable to various software systems?

To address the research question (RQ1), 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 \( PA_{\text{new}} \) is the partition obtained after the clustering analysis and \( PA_{\text{ori}} \) is the partition of the originally selected classes, then MoJoFM is calculated using

\[
\text{MoJoFM}(PA_{\text{new}}, PA_{\text{ori}}) = 1 - \frac{\text{mno}(PA_{\text{new}}, PA_{\text{ori}})}{\max(\text{mno}(\forall PA_{\text{new}}, PA_{\text{ori}}))},
\]

Fig. 12. The structure of the example system before and after refactoring.
Assumed that coefficients were not sensitive to the value of the value of more general coefficients were effective even when component of the residual network obtained by merging the classes in each connected ever, in our study, we regrouped the entity set split into two or three new classes, where for decomposition. In general, a god class should be contained sets of classes from each system, where each class set classes have cohesion or coupling problems. To ensure ses in the regrouping process if the selected original the god class may be split into more than approach does not specify the number of clusters, so the system. The clustering algorithm used by our the higher than the average cohesion of all the classes in and the cohesion of each selected class should be (1) The we selected had to satisfy the following conditions. To obtain the best configuration, the merged classes that were 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 ∈ [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 G1. 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 ∈ {2, 3, ..., |CSmax|}, where |CSmax| represents the number of classes in the largest connected components of G1.

(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 η = 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 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 G1, 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.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
<th>Introduction</th>
<th>TNC</th>
<th>TNE</th>
<th>ANM</th>
<th>ANA</th>
<th>ALC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elasticsearch</td>
<td>1.6.0</td>
<td>Elasticsearch is a distributed restful search engine built for the cloud</td>
<td>4,152</td>
<td>12,748</td>
<td>7.05</td>
<td>2.21</td>
<td>161.78</td>
</tr>
<tr>
<td>Android-async-http</td>
<td>1.4.7</td>
<td>An asynchronous, callback-based http client for Android built on top of Apache’s “HttpClient” libraries</td>
<td>73</td>
<td>126</td>
<td>6.75</td>
<td>4.27</td>
<td>157.68</td>
</tr>
<tr>
<td>Libgdx</td>
<td>1.6.2</td>
<td>LibGDX is a cross-platform Java game development framework based on OpenGL (ES)</td>
<td>1,988</td>
<td>4,572</td>
<td>7.22</td>
<td>2.67</td>
<td>105.60</td>
</tr>
<tr>
<td>Android-Annotations</td>
<td>3.3.1</td>
<td>Android-Annotations is an Open Source framework that speeds up Android development</td>
<td>696</td>
<td>1,948</td>
<td>4.79</td>
<td>3.94</td>
<td>79.20</td>
</tr>
<tr>
<td>Okhttp</td>
<td>2.4.0</td>
<td>An HTTP &amp; SPDY client for Android and Java applications.</td>
<td>239</td>
<td>522</td>
<td>4.70</td>
<td>3.04</td>
<td>249.83</td>
</tr>
<tr>
<td>RxJava</td>
<td>1.0.12</td>
<td>RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs using observable sequences</td>
<td>436</td>
<td>951</td>
<td>7.65</td>
<td>3.52</td>
<td>220.88</td>
</tr>
<tr>
<td>Spring-framework</td>
<td>4.2.0</td>
<td>The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications – on any type of deployment platform</td>
<td>6,139</td>
<td>16,014</td>
<td>6.32</td>
<td>4.55</td>
<td>155.54</td>
</tr>
<tr>
<td>Zxing</td>
<td>4.7.3</td>
<td>ZXing is a multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages</td>
<td>4,703</td>
<td>9,241</td>
<td>5.71</td>
<td>4.01</td>
<td>138.26</td>
</tr>
<tr>
<td>MPAndroidChart</td>
<td>2.1.0</td>
<td>MPAndroidChart is an easy to use chart library for Android</td>
<td>149</td>
<td>297</td>
<td>10.54</td>
<td>3.62</td>
<td>202.70</td>
</tr>
</tbody>
</table>
Algorithm 3. The algorithm of selecting classes to be merged.

Input: Class-level dependency network \(G_c\) of the open source system.

Output: The merged class set \(VM = \{V_1, V_2, \ldots, V_{NM}\}\), and \(NM = 100\);

Do[
1. Initialization: Suppose \(NU \in \{2, 3, \ldots, |CS_{seed}|\}\) is the total number of classes in set \(V_i \in VM\). Let \(V_{seed} = \varnothing\), \(V_i = \varnothing\), \(V_i = \varnothing\), 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 \(V_i\) is the class set that has been selected as seeds.
2. In network \(G_c\), 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 \(V_i = V_i \cup CL_{seed}\), \(V_i = V_i\) \(CL_{seed}\).
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 \(V_{seed}^{NB}\). Let \(V_{seed} = V_{seed} \cup (V_{seed}^{NB} \setminus V_i)\).
4. If \(V_{seed}^{NB} \geq NU - |V_i|\), then we randomly delete \(|V_{seed}^{NB} - NU| + |V_i|\) classes from set \(V_{seed}^{NB}\) to \(V_{seed}^{NB} = \varnothing\), and repeat step (3).
   If \(V_{seed}^{NB} < NU - |V_i|\) \& \& \(V_{seed} \neq \varnothing\), then we randomly select a class from \(V_{seed}\) as the new seed \(CL_{seed}\). Let \(V_i = V_i \cup CL_{seed}\), and repeat step (3).]

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 \(CLA\) is defined as the number of classes that have dependency relationships with class \(CLA\). MPC represents the total number of methods that do not belong to class \(CLA\) but that have dependencies with the methods in class \(CLA\). 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 percent of the system in cohesion measurement were filtered out beforehand. Table 2 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 \(\theta_1\) to eliminate the lower similarity values. For the three cases of \(\theta_1 = SSW_{med}\), \(\theta_1 = 0.1\), and \(\theta_1 = 0.2\), 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. After analyzing the results, we can make the following conclusions.

(1) Clearly, by setting the threshold \(\theta_1\) equal to the average semantic similarities, we could achieve higher refactoring accuracy. Fig. 15 shows that when \(\alpha\) and \(\beta\) were kept constant, the accuracy tended to increase rapidly with the decrease in the semantic coefficient \(\eta\). 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 \(\gamma\) and \(\eta\) 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

When we 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 \(\alpha + \beta + \eta = 1\), 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 \(\text{Th}_1 = SSW_{med}\). As shown in Fig. 15, the average accuracy had a significant positive correlation with the parameter \(\alpha\). 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 \(\text{Th}_1 = SSW_{med}\). 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 (RQ1), 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.

![Fig. 14. Illustration of the creation of an artificial god class.](image)

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>Cohesion and Coupling Metric Values of the Top 5 Selected Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cohesion /Coupling</strong></td>
<td><strong>Metrics</strong></td>
</tr>
<tr>
<td><strong>CBO</strong></td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td></td>
<td>Std.dev</td>
</tr>
<tr>
<td><strong>MPC</strong></td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td></td>
<td>Std.dev</td>
</tr>
<tr>
<td><strong>C3</strong></td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td></td>
<td>Std.dev</td>
</tr>
<tr>
<td><strong>LOCM</strong></td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td></td>
<td>mean</td>
</tr>
<tr>
<td></td>
<td>Std.dev</td>
</tr>
</tbody>
</table>
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 \( \alpha = 0.7, \beta = 0.2, \gamma = 0.1, \eta = 0.0, \) 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.

![Fig. 15. Comparison of the accuracy values under different coefficients and thresholds.](image1)

![Fig. 16. Contrast analysis of the quality for the 50 systems.](image2)
### 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 ($Th_1 = SSW_{med}$)

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

- **RQ1**: What are the advantages of our approach compared with clustering analysis using other refactoring algorithms?
- **RQ2**: Do the refactoring suggestions obtained using our approach actually make sense from the perspective of the developers?

To address research question **RQ2**, 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 described 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 \( k' < k \) 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 **RQ1**, 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 RQ2

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
The 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 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.

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 \(O(fg)\) and \(O(V^3)\) [38], \(O(fg)\) 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: \(O(fg) < O(V\log^2 V) < O(V^3)\). 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

\[\text{Fig. 17. Comparison of the MoJoFM metric values obtained using the four approaches.}\]
### 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)

<table>
<thead>
<tr>
<th>Class</th>
<th>Pre-refactoring</th>
<th>Our approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$L$</td>
<td>$Z$</td>
</tr>
<tr>
<td>Database</td>
<td>638</td>
<td>0.15</td>
</tr>
<tr>
<td>Select</td>
<td>43</td>
<td>0.27</td>
</tr>
<tr>
<td>UserManager</td>
<td>20</td>
<td>0.21</td>
</tr>
<tr>
<td>FileGeneratorAdapter</td>
<td>8</td>
<td>0.31</td>
</tr>
<tr>
<td>Import</td>
<td>29</td>
<td>0.10</td>
</tr>
<tr>
<td>JEditTextArea</td>
<td>12,876</td>
<td>0.11</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>236</td>
<td>0.09</td>
</tr>
<tr>
<td>NumberAxis</td>
<td>102</td>
<td>0.12</td>
</tr>
<tr>
<td>DefaultApplicationModel</td>
<td>81</td>
<td>0.14</td>
</tr>
<tr>
<td>XMLElementValidator</td>
<td>6,695</td>
<td>0.11</td>
</tr>
<tr>
<td>XMLSerializer</td>
<td>95</td>
<td>0.20</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE 6
(Class 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)

<table>
<thead>
<tr>
<th>Class</th>
<th>Bavota’s approach [15]</th>
<th>Fokaee’s approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$L$</td>
<td>$Z$</td>
</tr>
<tr>
<td>Database</td>
<td>155</td>
<td>0.29</td>
</tr>
<tr>
<td>Select</td>
<td>10</td>
<td>0.37</td>
</tr>
<tr>
<td>UserManager</td>
<td>6.5</td>
<td>0.31</td>
</tr>
<tr>
<td>FileGeneratorAdapter</td>
<td>5</td>
<td>0.43</td>
</tr>
<tr>
<td>Import</td>
<td>5.5</td>
<td>0.26</td>
</tr>
<tr>
<td>JEditTextArea</td>
<td>721</td>
<td>0.26</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>61</td>
<td>0.27</td>
</tr>
<tr>
<td>NumberAxis</td>
<td>35</td>
<td>0.28</td>
</tr>
<tr>
<td>DefaultApplicationModel</td>
<td>20.5</td>
<td>0.26</td>
</tr>
<tr>
<td>XMLDTDValidator</td>
<td>2,313</td>
<td>0.22</td>
</tr>
<tr>
<td>XMLSerializer</td>
<td>34</td>
<td>0.26</td>
</tr>
<tr>
<td>Average</td>
<td>64%</td>
<td>17%</td>
</tr>
</tbody>
</table>
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 RQ3

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 $TH_2 = avg_2$. 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, with the aim of further addressing research question RQ3.

#### 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

---

**TABLE 7**

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Database(41)</td>
<td>MoJoFM MJ</td>
<td>MoJoFM MJ</td>
<td>MoJoFM MJ</td>
</tr>
<tr>
<td>Select(14)</td>
<td>0.91 2</td>
<td>0.97 1</td>
<td>0.94 4</td>
</tr>
<tr>
<td>UserManager(13)</td>
<td>0.86 2</td>
<td>0.93 1</td>
<td>0.87 3</td>
</tr>
<tr>
<td>FileGeneratorAdapter(9)</td>
<td>0.86 1</td>
<td>0.86 1</td>
<td>0.76 2</td>
</tr>
<tr>
<td>Import(10)</td>
<td>1 0</td>
<td>1 0</td>
<td>0.65 9</td>
</tr>
<tr>
<td>JEditTextArea(214)</td>
<td>0.94 15</td>
<td>0.84 34</td>
<td>0.94 15</td>
</tr>
<tr>
<td>JFreeChart(24)</td>
<td>1 0</td>
<td>0.95 1</td>
<td>0.63 12</td>
</tr>
<tr>
<td>NumberAxis(20)</td>
<td>1 0</td>
<td>0.94 1</td>
<td>0.75 8</td>
</tr>
<tr>
<td>DefaultApplicationModel(14)</td>
<td>0.92 1</td>
<td>0.92 1</td>
<td>0.82 4</td>
</tr>
<tr>
<td>XMLDTDValidator(69)</td>
<td>0.98 2</td>
<td>0.88 8</td>
<td>0.79 23</td>
</tr>
<tr>
<td>XMLSerializer(69)</td>
<td>0.97 1</td>
<td>0.91 2</td>
<td>0.89 4</td>
</tr>
</tbody>
</table>
| Average                    | 0.95 2.2         | 0.91 4.7                        | 0.80 8.09                       

---

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, where $B_f/A_f$ 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 hierarchy.

| System      | $B_f$ | $A_f$ | $N_{nh}$ | $N_{noih}$ | $N_{leaf}$ | $|E_1|$ | $|E_2|$ | $|V_1|$ | $|V_2|$ | $|E_1|$ | $|E_2|$ | $TR$ | $N_{extra}$ | $N_{novice}$ | $N_{novice}$ |
|-------------|-------|-------|----------|------------|------------|--------|-------|--------|-------|--------|-------|------|-----------|------------|-------------|
| JHotDraw 7.0.6 | 309   | 205   | 104      | 114        | 1,435      | 182    | 175   | 479    | 33    | 11     | 13    | 1    | 20        | 19         | 8           |
| JFreeChart 0.9.7 | 551   | 309   | 242      | 140        | 1,344      | 311    | 159   | 293    | 50    | 16     | 14    | 8    | 20        | 19         | 8           |
| jEdit 2.7    | 251   | 156   | 95       | 112        | 1,154      | 128    | 183   | 739    | 36    | 20     | 19    | 8    | 20        | 17         | 1           |
| HSQLDB 1.8.1.4 | 301   | 132   | 119      | 57         | 1,419      | 84     | 153   | 745    | 37    | 20     | 17    | 1    | 20        | 17         | 1           |
| Jmol 9.0     | 169   | 78    | 91       | 44         | 558       | 47     | 133   | 363    | 14    | 11     | 8     | 0    | 11        | 8          | 0           |

10. https://github.com/wangying8052/Raw-Data-in-Section-7
1700 restructuring operation for the class
1699 jects. Significantly, one evaluator commented that the
1698 JFreeChart 0.9.7 system were mostly approved by the sub-
1697 Macros were frequently invoked by the methods defined in the class
1696 to deal with operations related to macros, as a result, they
1694 ods, we found that these moved entities were used mainly
1693 by analyzing the coupling relationships between the meth-
1692 should be moved to the class
1691 bute
1690 In particular, most of the participants agreed that the attri-
1689 method/Field refactoring operations received high marks.
1688 the feedback in the questionnaires, some of the move
1687 form these two types of operations, respectively. Based on
1686 and 84 and 63 percent of the evaluators were willing to per-
1685 toring operations in the inheritance of jEdit 2.7 made sense,
1684 inheritance related classes of JFreeChart 0.9.7 and the refac-
1683 that 100 percent of the extract class suggestions in the non-
1682 who chose “No” and “Maybe”. All of the participants stated
1681 ber who selected “Yes” was far higher than that of those
1680 the five systems were quite similar. In all the cases, the num-
1679 As shown in Table 9, the evaluation results obtained for
1678 by our approach, respectively.

<table>
<thead>
<tr>
<th>System</th>
<th>Refactoring</th>
<th>$Q_1$</th>
<th>$Q_2$</th>
<th>$Q_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>JHotDraw 7.0.6</td>
<td>Non-inheritance</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Inheritance</td>
<td>0.76</td>
<td>0.09</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>Move Method/Field</td>
<td>0.80</td>
<td>0.04</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>Non-inheritance</td>
<td>0.84</td>
<td>0.03</td>
<td>0.13</td>
</tr>
<tr>
<td>JFreeChart 0.9.7</td>
<td>Inheritance</td>
<td>0.93</td>
<td>0</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>Non-inheritance</td>
<td>0.85</td>
<td>0.05</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>Move Method/Field</td>
<td>0.99</td>
<td>0.01</td>
<td>0</td>
</tr>
</tbody>
</table>

As shown in Table 9, the evaluation results obtained for
the five systems were quite similar. In all the cases, the num-
ber 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 refac-
toring operations in the inheritance of jEdit 2.7 made sense,
and 84 and 63 percent of the evaluators were willing to per-
form 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 attribu-
tes 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 meth-
ods, 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 sub-
jects. Significantly, one evaluator commented that the restructure
operation for the class chartUtilites could be
considered as the perfect division. As shown in Fig. 20, after
the clustering analysis, the classes obtained,
chartUtilities_new_1 and chartUtilities_new_2, encapsulated
the functionalities of the PNG and JPEG image format set-
tings, 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 rela-
tionships 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 func-
tions 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 corre-
sponding 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.
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 participant 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 `BandPlotG2DRender` and `BandPlotEPSRender` 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, KeyedValues2D, 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. 19. Coupling relationships between the entities in the `View` and `Macros` classes.

![Diagram showing coupling relationships between entities in View and Macros classes.]

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.

![Diagram showing division of `chartUtilites` class.]

Fig. 20. Division of the `chartUtilites` class suggested by the proposed approach.
Reusability is equal to the maximum length from the node to the root in the inheritance tree of a design. As defined in Eqs. (11), (12), and (13), the ability model [43] and the ability model [25, 26, 27, 36, 43] are applied in the maintainability model. Both of the metrics are equal to the total number of classes that have integer relationships with the class.

The DCC metric 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.

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 sharing attributes subtracted from the total number of method pairs not sharing attributes. 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.

\[
\text{Reusability} = -0.25 \times \text{Coupling} + 0.25 \times \text{Cohesion} + 0.5 \times \text{Messaging} + 0.5 \times \text{DesignSize}
\]

\[
\text{Flexibility} = 0.25 \times \text{Encapsulation} - 0.25 \times \text{Coupling} + 0.5 \times \text{Composition} + 0.5 \times \text{Polymorphism}
\]
Understandability = \(-0.33 \times \text{Abstraction} - 0.33 \\
\times \text{Coupling} + 0.33 \times \text{Encapsulation} + 0.33 \times \text{Cohesion} \\
- 0.33 \times \text{Polymorphism} - 0.33 \times \text{Complexity} \\
- 0.33 \times \text{Design Size}\) (13)

Maintainability = \(k_0 \times \frac{1}{(\text{LCOM} \times \text{CBO} \times \text{WMC})} \\
\times \text{DIT} \times \text{NOC} \times \text{RFC}\) (14)

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

\[
E_{\text{System}} = \sum_{i=1}^{n} \frac{|CL_i|}{|V|} \times E_{PCL_i} \tag{15}
\]

\[
E_{PCL_i} = \frac{\sum_{et \in CL_i} \text{distance}(et, CL_i)}{|CL_i|} / \frac{\sum_{et \notin CL_i} \text{distance}(et, CL_i)}{|\text{entities} \notin CL_i|} \tag{16}
\]

where \(\text{distance}(et, CL_i)\) is the Jaccard distance between entity \(et\), and class \(CL_i\), and \(|\text{entities} \notin CL_i|\) 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 \(E_{\text{System}}\) 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 \(M_e\) 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
The class DrawingPanel 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 refactoring operations were suggested by our approach because the threshold \( T_{H2} \) 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 duplicative 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 11**

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

<table>
<thead>
<tr>
<th>Software</th>
<th>Approach</th>
<th>DSC</th>
<th>ANA (DIT)</th>
<th>DAM</th>
<th>MPC</th>
<th>CAM</th>
<th>MOA</th>
<th>NOP</th>
<th>CIS</th>
<th>NOM (WMC)</th>
<th>RFC</th>
<th>LCOM</th>
<th>NOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw 7.0.6</td>
<td>Before 309</td>
<td>0.62</td>
<td>0.73</td>
<td>18.78</td>
<td>0.11</td>
<td>0.53</td>
<td>1.83</td>
<td>11.96</td>
<td>16.31</td>
<td>23.15</td>
<td>218.73</td>
<td>1.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Snelting’s 428</td>
<td>1.42</td>
<td>0.50</td>
<td>12.38</td>
<td>0.19</td>
<td>0.60</td>
<td>1.07</td>
<td>7.54</td>
<td>11.80</td>
<td>17.01</td>
<td>201.32</td>
<td>6.98</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moore’s 317</td>
<td>0.73</td>
<td>0.69</td>
<td>18.83</td>
<td>0.13</td>
<td>0.57</td>
<td>1.49</td>
<td>11.10</td>
<td>16.02</td>
<td>22.47</td>
<td>207.55</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ours 343</td>
<td>0.61</td>
<td>0.63</td>
<td>17.61</td>
<td>0.25</td>
<td>0.72</td>
<td>1.63</td>
<td>9.92</td>
<td>14.95</td>
<td>18.95</td>
<td>142.85</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td>JFreeChart 0.9.7</td>
<td>Before 551</td>
<td>0.60</td>
<td>0.63</td>
<td>14.02</td>
<td>0.13</td>
<td>0.34</td>
<td>1.34</td>
<td>12.20</td>
<td>16.72</td>
<td>19.66</td>
<td>413.50</td>
<td>1.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Snelting’s 801</td>
<td>1.76</td>
<td>0.44</td>
<td>7.99</td>
<td>0.19</td>
<td>0.57</td>
<td>0.96</td>
<td>8.04</td>
<td>10.75</td>
<td>13.71</td>
<td>298.68</td>
<td>6.56</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moore’s 573</td>
<td>0.71</td>
<td>0.60</td>
<td>14.07</td>
<td>0.14</td>
<td>0.36</td>
<td>1.17</td>
<td>11.42</td>
<td>16.01</td>
<td>18.32</td>
<td>390.43</td>
<td>1.59</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ours 599</td>
<td>0.58</td>
<td>0.53</td>
<td>12.74</td>
<td>0.26</td>
<td>0.50</td>
<td>1.12</td>
<td>10.15</td>
<td>15.25</td>
<td>15.08</td>
<td>270.64</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>jEdit 2.7</td>
<td>Before 251</td>
<td>0.40</td>
<td>0.72</td>
<td>21.02</td>
<td>0.15</td>
<td>0.56</td>
<td>1.52</td>
<td>14.39</td>
<td>20.19</td>
<td>23.55</td>
<td>235.82</td>
<td>1.09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Snelting’s 414</td>
<td>1.03</td>
<td>0.44</td>
<td>10.43</td>
<td>0.23</td>
<td>1.05</td>
<td>0.90</td>
<td>9.67</td>
<td>9.02</td>
<td>12.97</td>
<td>157.90</td>
<td>6.82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moore’s 256</td>
<td>0.29</td>
<td>0.64</td>
<td>19.09</td>
<td>0.12</td>
<td>0.59</td>
<td>1.40</td>
<td>13.46</td>
<td>20.34</td>
<td>21.94</td>
<td>301.94</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ours 278</td>
<td>0.27</td>
<td>0.65</td>
<td>18.35</td>
<td>0.35</td>
<td>1.12</td>
<td>1.30</td>
<td>12.36</td>
<td>17.98</td>
<td>18.73</td>
<td>203.62</td>
<td>1.05</td>
<td></td>
</tr>
<tr>
<td>Jmol 9.0</td>
<td>Before 169</td>
<td>0.41</td>
<td>0.83</td>
<td>17.88</td>
<td>0.14</td>
<td>0.89</td>
<td>1.28</td>
<td>18.46</td>
<td>24.39</td>
<td>22.65</td>
<td>450.71</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Snelting’s 281</td>
<td>1.56</td>
<td>0.56</td>
<td>11.06</td>
<td>0.33</td>
<td>2.09</td>
<td>0.85</td>
<td>13.21</td>
<td>12.03</td>
<td>13.09</td>
<td>328.14</td>
<td>6.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Moore’s 176</td>
<td>0.43</td>
<td>0.81</td>
<td>17.20</td>
<td>0.20</td>
<td>0.90</td>
<td>1.20</td>
<td>17.50</td>
<td>23.96</td>
<td>21.86</td>
<td>437.00</td>
<td>1.08</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ours 187</td>
<td>0.34</td>
<td>0.81</td>
<td>15.75</td>
<td>0.30</td>
<td>1.71</td>
<td>1.02</td>
<td>16.39</td>
<td>23.08</td>
<td>18.35</td>
<td>417.18</td>
<td>0.93</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 12**

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

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

The class DrawingPanel 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 refactoring operations were suggested by our approach because the threshold \( T_{H2} \) 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 duplicative 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.
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. 22. Metric quotient changes after performing refactoring operations.](image)

![Fig. 23. Quality changes after performing refactoring operations.](image)
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 $O(2^{\text{Ent}} - 1|\text{Obj}||\text{Ent}|^2)$ [51] in the worst case, where $|\text{Ent}|$ is the total number of the entities in the system and $|\text{Obj}|$ 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 502 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.
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


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.

Zhiyong Li received the MS degree in computer science from Northeastern University, China, in 2013. He currently works as an associate professor in the Software College, Northeastern University. His main research interests include algorithm design and analysis, software refactoring, software architecture, and software testing.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.