УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
сентябрь–октябрь 2021 Том 21 № 5 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
September–October 2021 Vol. 21 No 5 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

УНИВЕРСИТЕТ ИТМО

doi: 10.17586/2226-1494-2021-21-5-738-747

Software restructuring models for object oriented programming languages using the fuzzy based clustering algorithm

Sarika Bobde[1]✉, Rashmi Phalnikar[2]

[1,2] MIT World Peace University, Pune, Maharashtra, 411038, India

[1] sarikabobde27@gmail.com✉, https://orcid.org/0000-0002-6693-1364
[2] rashmi.phalnikar@mitwpu.edu.in, https://orcid.org/0000-0002-2004-7944

Abstract
Advances in the domain of software-based technology pave the way for widespread use of object-oriented programs. There is a need to develop a well-established software system that will reduce maintenance costs and enhance the usability of the component. While designing a software system, its internal structure deteriorates due to prolonged or delayed maintenance activities. In such situations, restructuring of the software is a superior approach to improve the structure without changing external behaviour of the system. One approach to carry out restructuring is to use refactoring on the existing source code for improving the internal structure of the code. Code refactoring is an effective technique for software development that improves the software's internal structure without changing its external behaviour. The purpose of refactoring is to improve the cohesion of existing code and minimize coupling in the existing module of a software system. Among numerous methods, clustering is one of the effective approaches to increase the cohesion of the system. Hence in this paper, the authors suggest to extract member functions and member variables and propose to find their similarity by Frequent Usage Pattern approach. Next, the proposed fuzzy based clustering algorithm perform effective code refactoring. The proposed method utilizes multiple refactoring methods to increase the cohesion of the component without any change in the meaning of the software system. The proposed system will offer automated support to change low-cohesive to high-cohesive functions. Finally, the proposed model has been experimentally tested with object-oriented programs.

Keywords
refactoring, cohesion, clustering, member function, member variable, Frequent Usage Pattern, fuzzy c-means clustering, k-nearest neighbour

For citation: Bobde S., Phalnikar R. Software restructuring models for object oriented programming languages using the fuzzy based clustering algorithm. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2021, vol. 21, no. 5, pp. 738–747. doi: 10.17586/2226-1494-2021-21-5-738-747

УДК 004.438

Модели реструктуризации программного обеспечения для языка объектно-ориентированного программирования с использованием алгоритма нечеткой кластеризации

Сарика Бобде[1]✉, Рашми Пхальникар[2]

[1,2] Школа компьютерной инженерии и технологий, MIT — Всемирный университет, Пуна, Махараштра, 411038, Индия

[1] sarikabobde27@gmail.com✉, https://orcid.org/0000-0002-6693-1364
[2] rashmi.phalnikar@mitwpu.edu.in, https://orcid.org/0000-0002-2004-7944

Аннотация
Достижения в области программных технологий открывают путь для широкого использования объектно-ориентированных программ. Существует необходимость в разработке зарекомендовавшей себя системы программного обеспечения, которая снизит затраты на обслуживание и повысит удобство использования компонента. При проектировании программной системы ее внутренняя структура ухудшается из-за

© Bobde S., Phalnikar R., 2021

продолжительных или отложенных работ по техническому обслуживанию. В таких ситуациях реструктуризация программного обеспечения — лучший подход для улучшения структуры без изменения внешнего поведения системы. Один из подходов к реструктуризации — использование рефакторинга применяемого исходного кода для улучшения внутренней структуры кода. Рефакторинг кода — эффективный метод разработки программного обеспечения, который улучшает внутреннюю структуру программного обеспечения без изменения его внешнего поведения. Цель рефакторинга — улучшение связности используемого кода и минимизация связи в модуле программной системы. Кластеризация — один из эффективных подходов к увеличению сплоченности системы. В работе предложено извлечение функций-членов и переменных-членов и выполнение поиска их сходства с помощью подхода «Шаблон частого использования». Алгоритм нечеткой кластеризации дает эффективный рефакторинг кода. Предлагаемый метод использует несколько методов рефакторинга для повышения связности компонента без какого-либо изменения смысла программной системы. Представленная система рекомендует автоматическую поддержку для изменения функций с низким уровнем сцепления на функции с высоким уровнем сцепления. Предложенная модель экспериментально протестирована с объектно-ориентированными программами.

Ключевые слова
рефакторинг, связность, кластеризация, функция членства, переменная, часто используемый паттерн, нечеткая кластеризация методом c-средних, k-ближайший сосед

Ссылка для цитирования: Бобде С., Пхальникар Р. Модели реструктуризации программного обеспечения для языка объектно-ориентированного программирования с использованием алгоритма нечеткой кластеризации // Научно-технический вестник информационных технологий, механики и оптики. 2021. Т. 21, № 5. С. 738–747 (на англ. яз.). doi: 10.17586/2226-1494-2021-21-5-738-747

Introduction

Nowadays, object-oriented programs play an important role in software development scenarios and are extensively used by the research community. Also, software engineering always aims to provide low-maintenance software systems with low maintenance costs and time [1, 2]. Many researchers have reported that object-oriented practice is a beneficial method for developing software systems with good quality. At this point, good quality software defines that the system is manageable, reusable, easily modifiable and extensible based on the requirements [3, 4]. The changing requirement can lead to the continual development of real-world software. But the code becomes complicated when the software is modified and adapted. Then, its software quality diminishes, and the overall development cost also increases [3–5]. This generates the necessity for the methods that lessen the software complexity and therefore improve the quality and diminish the cost of maintenance. This type of technology is called refactoring, or refactoring in the development of object-oriented (OO) software. Refactoring has written the code into a simplified procedure for improving its internal operation without changing its external behaviour.

Software refactoring objective is to improve efficiency, such as understanding many aspects of quality. This makes it convenient for developers by making the program faster and helping in finding errors. As refactoring approaches change the internal code structure, quality metrics like coupling and cohesion also change [7–9]. The good quality software represents maximum cohesion and minimum coupling measure. Whenever developers planned to augment software quality, its coupling and cohesion became more challenging or more satisfying. If the outcome of quality metrics was not satisfied, the entire system tends to fail with high complexity [10]. By keeping this in mind, designers planned for an automated approach that has emerged as a suitable solution. Among various methods, pattern recognition concepts are applicable for achieving high cohesion software components by employing a refactoring technique. Generally, in the software development field, the cohesion measure is elaborated as the degree to which several software components are related. These measures are implemented at the class level of the code [11].

Researchers and programmers are now focused on the high cohesion model. Because it possesses advantages such as reusability, maintainability, understandability, and ability to modify the code, but these characteristics are not supportable in the low cohesion component. Generally, in a software framework, a poorly structured class comprises member functions and member variables that are not related to a class. Thus, for enhancing the modularity of an object-oriented software system, refactoring is a well-suited approach. In refactoring models, various operations like Move Method, Extract Class / Method are applied to improve its effectiveness without changing code functionality or behaviour. Maintenance of the code becomes more complex, if there are larger numbers of complex classes with unrelated functions and members [12, 13]. This kind of difficulties is minimized using clustering approaches where relatedness among member functions and member variables are grouped based on similarity. Afterwards, refactoring procedures are applied to eliminate mistakes recognized from the clustering approach.

While restricting software, these steps are followed to get the efficiency of the code which means that the internal structure of code improves. Thus, the reliability of the software model improves, and the number of the efforts taken for maintenance decreases. The most effective model to measure and specify the relatedness between member functions is the FUP approach. Here the member function usage pattern refers to the set of member variables that are accessible by direct or indirect calls to other member functions. This improves the consistency of the software because if the member functions share the same FUP between them, they will be linked to the same internal data usage [14, 15]. So in this paper, we analyse the software program to identify classes with unrelated

member functions and propose appropriate refactoring to eliminate the identified code problem and improve the software program infrastructure. This helps to improve the cohesive quality of the system. This metric is based on information collected by using frequent usage patterns of member functions. The clustering algorithm aims to group the respective member functions. Based on these clusters, we propose a refactoring algorithm to provide a set of refactoring processes that developers should follow to improve cohesion without changing the specifications of the software system.

The contributions of the study are illustrated as follows:

- The restructuring of software code is done for reducing maintenance costs and improving reusability.
- Clustering approaches are included in the proposed work to enhance the software design for the purpose of refactoring.
- Different refactoring techniques are utilized to identify errors in the software design.
- The utilization of different refactoring methods improves the cohesion metric and execution time of software systems.

The manuscript is prepared as follows: section II describes the related work. Section III illustrates the motivation of the research. The planned methodology is elaborated in section IV. Results and discussion are done in Section V. Conclusion is provided in the final section.

Literature survey

This section details the review of existing literature in software restructuring models. Among several kinds of research, some of the latest works are illustrated as well.

Alkhalid et al. [16] exhibited the Adaptive K-Nearest Neighbour (A-KNN) technique to achieve clustering based on the similarity in attribute weights. This methodology helps to support software designers in the refactoring process at the method/function level. This was done by means of discovering bad code or ill-structured software entities. But the obtained accuracy was not up to the mark with poor cohesion measure. So that internal complexity, cost and maintenance efforts are not achieved in an effective manner.

Rao et al. [17] presented a software refactoring model using two approaches. At first, low cohesive classes were discovered. Then clustering approaches were developed based on supplemented agglomerative clustering and were validated by Weyuker's properties. For extracting class refactoring, clustering concepts were applicable based on Jaccard similarity metric values between class members.

Wang et al. [18] developed a multi model refactoring approach at the system level that helps to discover the move method, move field, and extract class refactoring prospects. The presented technique merges and splits the classes related among each other for acquiring ideal functionality distribution from the system level. For regrouping the entities, the weighted clustering method was utilized depending on merged method level networks. The bad codes obtained from inheritance and non-inheritance hierarchies were removed through pre-processing and pre-conditions approaches without modifying code behaviour.

Rathee et al. [19] examined the empirical evaluation of different dependency relationships in modelling dependency between different software components. Then a weighted dependency measurement scheme was developed by combining conceptual, structural, and change history-based relationships between software components. At last, various dependency models were estimated with different clustering techniques and were applied with open source Java code. These source codes are of dissimilar sizes and from different domains.

Khan et al. developed an approach named Distributed Object Oriented (DOO) based on the object-oriented concept in the point of the reference period. The core of DOO frameworks is the scattering of programming classes between different centres, and its main goal has no top-class circulation. Thus rebuilding is to be finished. To reinforce performance, DOO programming was restructured using a multifaceted strategy called the Neural Network (NN). First, a class dependency graph (CDG) was developed, in which hubs belonged to classes, and yet hubs associated with classes between situations. Currently, factors, lines, and import components were linked to the classes in the CDG presented to NN for preparation. Then, a set of prepared highlights was completed, using the OO Framework Class Dependency Based Clustering (CDBC) strategy, which was divided into subsystems with minimal coupling. Finally, grouped classes were embedded in bunch diagrams using the K-Medoid bunching method.

Alizadeh et al. [20] exhibited an interactive clustering approach for refactoring the software. The authors utilized different refactoring techniques with the help of multi-objective search functions for enhancing the software quality. They executed only the recommendation model for software refactoring so the received feedback from the developers may not be suitable for some refactoring strategies.

Sarika Bobde and Rashmi Phalnikar [21] presented a code refactoring model in an object-oriented software model to progress the cohesion metric of the code. The authors used clustering techniques for restructuring the software system and developed multiple refactoring approaches in the source code for well developed software design. But due to the lack of metric similarity selection, this presented method failed to reach the best performance level.

After analysing the literature review, each methodology contains some challenges. Based on that, some of the identified problems are given below:

- Program with duplicate logic is hard to read and modify.
- It is a common fact that a good software quality should possess high cohesion and low coupling measure. This objective lacks in the mentioned state-of-art techniques.
- If the quality metrics are achieved in existing approaches, then their computation time and accuracy were not maintained.

To outfit the current problem and lessen the computation amount, it is planned to implement a new clustering algorithm in the current research.

Motivation of the research

The development of software programs is a big business. While there is huge expense and efforts spent

for preparing a code for its release, a few investigations demonstrate that 40–65 % of the complete effort is spent for maintaining software after delivery with the goal that the maintenance is costly. These reasons motivate us to research in this field by attempting to diminish maintenance costs. Maintainability denotes making software easier which can be done via modifications. Here some of the essential modifications include bug fixes, integration of additional features and adaption of software functionalities suitable to different environments. When the maintainability of the software is affected, its whole quality will be corrupted. Modular and flexible software is generally easier to analyse and manage than poorly structured or complex software. Generally, programmers should possess knowledge about well-structured projects and can be able to understand the purpose of a particular code. Also, the code improvement should be made in a short period of time. But, not all the software "improvements" are worth doing. At every time of modification, the cost becomes higher. In addition to the cost, during the change in a program, there are potential additional costs when the external functionality of the software changes, e.g., updating test scripts and documentation. In addition, any changes to the software pose risks – new bugs may occur, or the code may be less manageable. Due to these issues, many software businesses choose to be conservative when making code changes. The perceived benefit of the change should outweigh the costs and risks considered.

Proposed methodology for software restructuring

Cohesion and coupling play a major role in designing a good software model — both the metrics measure the relatedness among different classes or modules. Generally, attaining a reusable and low maintenance software system is the desired goal, so the cohesion measure must always be high. However, most of the time, the cohesion measure for the particular module/class may lessen due to adding up new functionalities or maintenance activities in designing software. To overcome this drawback, the proposed methodology aims to enhance the cohesion of the module by exploiting member variables (MV) usage patterns and developing refactoring procedures on software.

— The method contained in the low cohesion class is transformed into a high cohesion class.
— The classes are separated into two or more cohesive classes.
— If the member function (MF) in different classes contains the same MVs means, then splitting is carried out under the basis of inheritance.

The upcoming section provides a detailed framework of the proposed methodology. The current implementation is carried out in Java, though the technique will also be applied to other programming languages. Initially, the constructed project, along with a total number of classes in it, (CL_1, CL_2, CL_3, …, CL_n) is generated. Immediately, the member variables (MV_1, MV_2, MV_3, …, MV_n) and member functions (MF_1, MF_2, MF_3, …, MF_n) are extracted from each class that is contained in a software system. Then the FUP (Frequent Usage Pattern) is calculated for each MV from dissimilar MF in the class. Then, the FUPs information is determined in the form of vectors. Here each and every vector signifies the usage pattern of MVs in place of the corresponding MF. This vector formation is utilized to define the similarity in the MV usage pattern between MFs. At once, the similarity measure is computed among MVs usage patterns from MFs with the help of these vectors. From the calculated similarity measure, the clustering of MF is carried out through the proposed fuzzy c-means clustering algorithm. The obtained cluster denotes a single cohesive set of MVs. Finally, each MF cluster is refactored using the proposed refactoring technique. The stepwise process of the proposed methodology is displayed in Fig.

The main objective of this proposed methodology is to reorganize the internal structure of the program in order

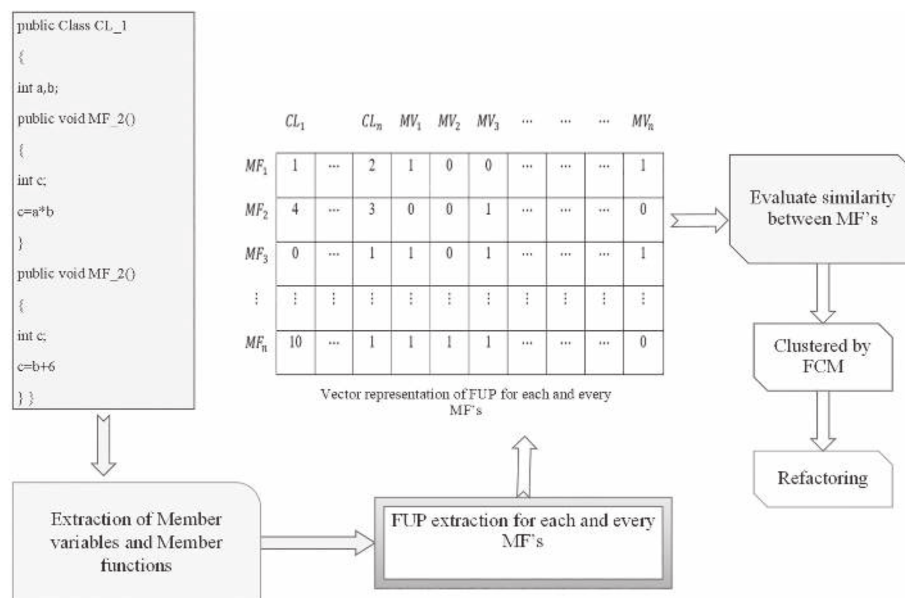

Figure. Layout of the proposed methodology

to reduce its complexity. To accomplish this program restructuring, we used the sample code written in Java for the experiment. Then, the member variable and member function that are present in the sample source code are extracted. The next step implies that the member variables directly or indirectly associated with the member function are extracted using frequent usage pattern extraction technique. The extracted and based on FUP member function is represented in vector form. Subsequently, the similarity existing between the member function is evaluated based on cohesion metrics. After that on the basis of the cohesion metric, the member functions which are responsible for similar activity are grouped into clusters using the fuzzy c-means clustering algorithm. Finally, the output obtained as a result of the clustering algorithm is fed as an input into refactoring approach to achieve an effective software system. The performance of the proposed software restructuring model can be evaluated using cohesion metrics. The steps involved in this proposed methodology are briefly explained below.

Extraction of member variables and member functions.

- Extract MVs like public, protected and private defined in the class of software system. The extracted variables help to identify frequent usage pattern for different MF present in the software model.
- Extract MFs of all private and public variables apart from the constructor, setter or getter functions defined in the class of software.

Here constructors type MF is not at all extracted because it initializes the objects as well as it belongs to specific classes. So they cannot be moved or defined among other classes. Likewise, the setter and getter are unique functions in object-oriented software development standards that permit access to the private MV of a class (getter) and their initialization, i.e., setting MV to a known value (setter). They help to retrieve and set public MVs wherever in the software model. The private MFs cannot be accessed directly outside the class, but they can be directly called in the same class of public MFs. Because of this, they may be functionally associated with further public MFs. The extracted MVs and functions and their corresponding class details are documented in a data structure format. This aids to identify the corresponding class during refactoring.

Frequent Usage Pattern extraction. At the second phase of the proposed work, the algorithm evaluates FUPs for each and every MF for all MVs. This FUP plays a significant role in recognizing dependencies between MFs. While refactoring the code, this form of dependency information is utilized for improving the cohesion measure of classes in a software system. The FUP is recognized by means of a subset of MVs, which are frequently utilized together by MFs in the designed software model. The subset of FUP of a given member variable is denoted as M_i; through this, we can identify all other MVs directly or indirectly. The indirect usage comprises MVs accessed throughout the other MFs, which are called by M_i. This indirect usage does not contain recursive calls so that the outcome is in the form of an infinite loop because of the static system analysis, and their uses have been observed.

llustration of vector for member functions FUP. The FUP is initialized as a vector of size ($Tc + n$) for fining MVs usage similarity between MFs. Here Tc represents the total number of class and n is the total number of MFs present in the designed software model. The vector format is represented as a row-wise and column-wise matrix form. In the vector format, Tc slots symbolize the classes and define the total number of MVs of a class utilized by M_i. Similarly, n represents the MVs which may be either 0 or 1. Where 1 in the MV slot denotes consequent MFs, which use its consequent MV, and 0 indicates MFs, which do not use its consequent MV. This must be determined under the basis of the identified FUP for their consequent MFs. The value of the vector becomes 0 when MF does not use any of the MV of the classes accordingly in the software system. The overall information will be utilized by the refactoring mechanism.

Evaluating similarity based cohesive metrics. As per knowledge, the clustering algorithm helps to cluster the segments of codes. In clustering approaches, any two clusters are grouped together on the basis of similarity measures. Therefore, similarity measures should be available for cluster analysis. So proposing the similarity metric helps to point out the closeness measures of two MFs contained in the value of vector space and this information is used for the next step. Two MFs that have a higher similarity metric are regarded to be interrelated with each other. In other words, to find similarity among MF pairs, the similarity metric is introduced, which measures the relatedness between MFs. In our case, a new similarity measure FUP is studied and evaluated. The following paragraph describes the cohesion metric working principle.

For improving the cohesion measure, FUP is formulated as:

$$\textit{Similarity mytric}\ (M_i, M_j) = \frac{|\textit{use pattern}\ (M_i) \cap \textit{use pattern}\ (M_j)|}{|\textit{use pattern}\ (M_i) \cup \textit{use pattern}\ (M_j)|}.$$

Thus, the similarity metric was evaluated based on the FUP set defined above. It is represented as the proportion of the total number of the same MVs utilized to the total number of unique MVs retrieved by both MFs. This type of information can be used for refactoring to improve the cohesion of classes in a software framework. The obtained results are used as the input to the proposed clustering algorithm. Increasing cohesion of the class will eventually reduce the coupling due to refactoring.

Coding with an example. To the best of our knowledge, the FUP metric analysed in the proposed solution performs better than all other metrics. So, an illustration is presented below to investigate the computation of the FUP strategy that helps to verify and validate our cohesion improvement approach. It also defines how FUP is interrelated by modularization and shows improved cohesion in object-oriented classes of software design. The sample source code written in Java programming language is given below in Table 1.

The original sample code given in the above table comprises four classes, namely CL_1, CL_2, CL_3 and CL_4. At the method level, the FUP is processed, and it includes

Table 1. Sample source code

Class 1	Class 2	Class 3	Class 4

public Class CL_1
{
int a, b
public void MF_1()
{
int c;
$c = a \times b$;
}
public void MF_2()
{
int c;
$c = b + 6$;
}
}
	public Class CL_2
	{
	int d, e, f;
	public void MF_3()
	{
	int c
	$c = d + e$;
	}
	public void MF_4()
	{
	int c;
	$c = \frac{e \times f}{2}$;
	}
	}
		public Class CL_3
		{
		int g, h;
		public void MF_5()
		{
		int c;
		$c = g + 4$;
		}
		}
			public Class CL_4
			{
			int i, j;
			public void MF_6()
			{
			CL_1 $c = newCL_1$();
			$c.MF_3$();
			}
			public int MF_7()
			{
			return $(i + j)$;
			Public int MF_8()
			{
			return $(i - j)$;
			}
			}

a sequence of the unordered MV names utilized by its equivalent MF indirectly or directly called to different methods of the classes. The method MF_1 from CL_1 class is described and the FUP is a set$\{a, b\}$; the same class consists of another one method MF_2(), and its FUP is set$\{b\}$ and considering the second class CL_2 which consists of two methods MF_3() and MF_4(), for that FUP is s set of $\{d, e\}$ and $\{e, f\}$ respectively. Similarly, the FUP of every MF is computed in the software framework. The ideology adopted here is that if any MF shares a set of common MFs, they should be put together to fulfil conceptual dependency. Afterwards, merging and splitting (bad code) is done for the internal structure of classes which is shown in the subsequent Table 2.

In the modified Java code, classes CL_2 and CL_3 are merged together which forms CL_2_CL_3. Next to this, the final class CL_4 is split into Class CL_4_1 and Class CL_4_2. This modified code comprises eight MFs and nine MVs. MF_1() uses a and b member variables, whereas MF_2() utilizes member variable b. Likewise, all MFs contain their corresponding MVs, and all other FUP are computed for the rest of MFs, which vary from MF_3() to MF_8(). This FUP is denoted in the vector form where 0 denotes the non-usage of MV and 1 denotes the usage of MV. This vector encodes the information of a total number of MVs accessed for the corresponding class by means of MF. This information is useful for restricting the software model in the final stage. After the vector representation, a similarity measure is computed between the pair of MFs. It helps to calculate the relatedness between MFs as the ratio of similar accessed MVs (FUP intersection) to the sum MVs (FUP union) accessed uniquely by both MFs.

From the example code, the similarity between a member function MF_1() and MF_2() is ½ (0.5). Likewise, the similarity between the other two member functions MF_1() and MF_3() is 0/4 (0). These similarity values are evaluated using the metric formula and are denoted in a square matrix M of size 8 × 8. This obtained matrix is given

Table 2. Modified sample source code

Class 1	Class 2	Class 3	Class 4

public Class CL_1
{
int a, b;
public void MF_1()
{
int c;
$c = a \times b$;
}
public void MF_2()
{
int c;
$c = b + 6$;
}
}
	public Class CL_2_CL_3
	{
	int d, e, f, g, h;
	public void MF_3()
	{
	int c;
	$c = d + e$;
	}
	public void MF_4()
	{
	int c;
	$c = \frac{e \times f}{2}$;
	}
	public void MF_5()
	{
	int c;
	$c = g + 4$;
	}
	}
		public Class CL_4_1
		{
		int i, j;
		public void MF_6()
		{
		CL_1 $c = newCL_1$();
		$c.MF_3$();
		}
		public int MF_7()
		return $(i + j)$;
		}
		}
			public Class CL_4_2
			{
			int i, j;
			Public int MF_8()
			{
			CL_4_1 $c = newCL_4$_1();
			return ($c.MF_7$());
			}
			}

as the input to the next phase of the fuzzy based clustering. The proposed clustering algorithm gives four clusters as outputs which are $\{MF_1, MF_2\}$, $\{MF_3, MF_4, MF_6\}$, $\{MF_5\}$, $\{MF_7, MF_8\}$. In the clustering outcome, MF_5 is the method to be found in class $CL_2_CL_3$, and it is placed separately after performing the merging operation. Likewise, MFs of split class CL_4_1 and class CL_4_2 are located in the same cluster as $\{MF_7, MF_8\}$. Therefore, the FUP plays a significant role in dependency identification between MFs. This identified dependency data is used during refactoring for improving classes' cohesion in the software model.

Clustering based on fuzzy c-means approach. The created matrix is fed as the input to the clustering strategy. The clustering phase implements the proposed fuzzy c-means clustering algorithm (FCM). It reduces the chance of reusability and maintenance; then it will obtain low cohesion. FCM is a suitable approach for obtaining high cohesion where a separate class is defined for all jobs to execute a specific job. Also, its clustering process depends directly on the distance measure. It is the combination of the c-means strategy by handling fuzzy data. In this clustering technique, at first, each MF is assigned to a separate cluster which forms m number of clusters on the basis of distance among the cluster centre and MFs. The reason behind choosing the FCM technique is that it has a high grouping capacity and offers the best outcome for the overlapped project. In the k-means clustering approach, data points belong to one cluster centre, whereas in the proposed FCM method, the data point may belong to more than one cluster through the membership function. Therefore, FCM performs comparatively better than the k-means algorithm.

The working principle is shortly given as follows:

Step 1: Initialize the created matrix, which is represented as $Y = \{y_1, y_2, y_3, \ldots, y_n\}$ and the set of clusters is also initialized as $V = \{v_1, v_2, v_3, \ldots, v_c\}$.

Step 2: Select the cluster centres randomly.

Step 3: Evaluate the fuzzy membership function by $\mu_{ij} = \dfrac{1}{\sum_{l=1}^{cc}\left(\dfrac{d_{ij}}{d_{il}}\right)^{(2/m-1)}}$.

Step 4: The fuzzy centre is calculated by $v_{j=1,2,\ldots c} = \dfrac{\left(\sum_{i=1}^{n}(\mu_{ij})^m y_i\right)}{\left(\sum_{i=1}^{n}(\mu_{ij})^m\right)}$.

Step 5: Repeat the above steps (3) and (4) until the minimum objective is reached.

The objective function is formulated as

$$J(u, v) = \sum_{i=1}^{n}\sum_{j=1}^{c}(\mu_{ij})^m\|y_i - v_j\|^2,$$

where $\|y_i - v_j\|^2$ defines Euclidean distance between $i - th$ and $j - th$ cluster centre.

After this, merge any two close MFs showing the highest similarity value into a single cluster which forms $m - 1$ total clusters. Until we get the single cluster with all MFs, the process of merging takes place. The output of the clustering technique is a set of clusters. This set of clusters helps to aid the software designers in discovering a low cohesive measure. Then software designers decompose them into numerous fragments of code and compose them into new functions. The objects in each cluster have higher relatedness than the objects in dissimilar clusters. As information gained from the proposed research, it is important to improve the software systems design by refactoring during the emergence of object-oriented software systems. Refactoring aims to change a software system so that it not only modifies the performance of the external code but also improves its internal structure.

Refactoring model. The final stage of the proposed work focuses on the refactoring for the obtained clusters. The obtained cluster is taken as the input for refactoring, which is a highly related set of MFs. As much as possible, the highly related clusters should be present together for enhancing the cohesion measure and programming frameworks internal structure. By moving MFs of corresponding MVs, the refactoring of the source code is done efficiently. For refactoring, different approaches are utilized, which are move methods, class extraction and class extraction with its inheritance, etc. The main advantage of the proposed model is that refactoring can be achieved in multiple ways. So combing different methods, our refactoring model works based on the following three cases:

Case 1: First, check if each method in a cluster can be grouped into a new class based on the sharing of MVs with other methods.

Case 2: Or else, it attempts to move methods to previous classes depending on the relatedness value for achieving the cohesion of the classes concerned.

Case 3: At last, if the defined two cases are not feasible, due to high sharing of MVs between the methods and their non-movement nature (because MVs cannot be copied directly from one class to another with the methods), therefore the technique groups methods of a cluster in a new cluster along with inheritance by defining a new base class. That base class comprises all the MVs which are shared. This can be done for preserving the software integrity. These steps are illustrated in the following table.

Thus, the benefit of the proposed method is likely to increase the effectiveness of the software system by improving its cohesion measure.

Experiments and discussion

Several experiments were conducted to verify the proposed procedure and explain how the sample projects can be written differently. The entire methodology was tested with the metrics like cohesion, accuracy and time. As mentioned earlier, when the cohesion of a class increases, its coupling automatically decreases. The present implementation is written in Java. Moreover, the technology can also be supported by other programming languages.

Performance metrics. The cohesion metric used in this methodology is the Lack cohesion in methods (LCOM), Tight Class Cohesion (TCC), Loose Class Cohesion (LCC),

Accuracy and time. The following notes describe the working principle of those metrics:

LCOM: LCOM metric defines the number of pair of methods in the class using several instance variables in common. Assume that M represents the pairs of methods without shared instance variables and N defines the pairs of methods with shared instance variables. Then, LCOM is formulated as:

$$LCOM = \begin{cases} |M| - |N|; & if\,|M| > |N| \\ 0; & if\,|M| - |N|\ is\ negative \end{cases}.$$

TCC: TCC is defined as the relative number of common methods connected directly. Assume that a class contains M number of common methods. Let MP represent the maximum number of common pairs of methods and is given as $MP = [M \times (M - 1)]/2$. The mathematical notation of TCC is given as follows:

$$TCC = MDC/MP.$$

Here MDC defines the number of direct connections among common methods.

LCC: LCC is defined as the relative number of common methods connected directly or indirectly and is formulated by:

$$LCC = MIC/MP.$$

Here MIC term denotes the number of direct or indirect connections among common methods.

The following sub-sections elaborate on the outcome of these experiments.

Evaluation based on doc MGR project.

Meanwhile, the values obtained after clustering are higher than the ones before clustering for total lines, representing the total number of member functions and member variables. Also, the cohesion value obtained by LCC and TCC is high for the proposed approach, and the LCOM value obtained is lesser for the proposed method. As we know, TCC and LCC scores range from 0 (the least cohesive) to 1 (the most cohesive). And if the LCOM measure obtains lesser value means, the cohesion measure will be better. Likewise, the LCC value is always greater than TCC. Here it is clear that cohesion is improved after the refactoring, then automatically coupling gets minimized after performing the refactoring. Following this performance evaluation, we performed a comparison between proposed refactoring and existing refactoring method. Table 4 illustrates the comparative analysis carried out between the proposed and conventional refactoring techniques.

The comparison carried out between the proposed and conventional refactoring method is displayed in Table 3. The analysis revealed that cohesion or coupling measure are improved using the proposed refactoring approach compared with conventional methods. This is proved through evaluating some of the cohesion metrics like LCC, TCC and LCOM. The obtained values for these cohesion metrics are lower than the ones for the existing approaches. Finally, based on this study, it is shown that the improved cohesion is attained using the proposed refactoring model for program restructuring.

Conclusion and future work

This work aims to provide automated assistance to identify low-cohesive or poorly-structured functions and to provide refactoring recommendations to software designers to guide their refactoring operations. For improving cohesion performance, initially, member variables and member functions were extracted, and then their frequent usage pattern structures based on the dependencies between member functions were extracted. Then fuzzy c-means based clustering technique was proposed for refactoring to acquire a good quality software system. The main advantage of the proposed clustering algorithm is that each MF is assigned to a separate cluster. It results in better usability and maintenance, which causes the lowest time, cost, and effort. The proposed methodology is executed

Table 3. Comparison before and after clustering

	Performance before clustering	Clustering by the proposed FCM
Total Lines	11 862	16 567
Total No. of Member Function	244	332
Total No. of Member Variable	62	84
Lack cohesion in methods (LCOM)	2448.0	1871.0
Loose Class Cohesion (LCC)	0.842	1.0
Tight Class Cohesion (TCC)	0.817	0.936

Table 4. Analysis carried out between the proposed and conventional refactoring approach for program restructuring

Performance metric	Move Method Refactoring [22]	Extract Subclass Refactoring [23]	Refactoring based on modularity [15]	Refactoring based on the proposed fuzzy clustering algorithm
TCC	0.658	0.63	0.33	0.817
LCC	0.699	0.66	0.78	0.842
LCOM	3585	2566	2356	1871

in object-oriented programming languages and was tested with existing algorithms to show the superiority of the proposed method.

Future scope

In future, we plan to use hybrid clustering approaches that will help further to decrease computation time. Another research direction includes various types of refactoring, such as Rename, to be automatically handled by the proposed model.

Compliance with ethical standards

Funding: There is no funding provided to prepare the manuscript.

Conflict of Interest: There is no conflict of interest between the authors regarding the manuscript preparation and submission.

Ethical Approval: This article does not contain any studies with human participants or animals performed by any authors.

Informal Consent: Informed consent was obtained from all individual participants included in the study.

References

1. Keshta I.M. Software refactoring approaches: A survey. *International Journal of Advanced Computer Science and Applications*, 2017, vol. 8, no. 11, pp. 542–547.
2. Alkhalid A., Alshayeb M., Mahmoud S.A. Software refactoring at the class level using clustering techniques. *Journal of Research and Practice in Information Technology*, 2011, vol. 43, no. 4, pp. 285–306.
3. Srinivas C., Radhakrishna V., Rao C.V.G. Clustering software components for program restructuring and component reuse using hybrid XNOR similarity function. *Procedia Technology*, 2014, vol. 12, pp. 246–254. https://doi.org/10.1016/j.protcy.2013.12.482
4. Fokaefs M., Tsantalis N., Chatzigeorgiou A., Sander J. Decomposing object-oriented class modules using an agglomerative clustering technique. *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2009, pp. 93–101. https://doi.org/10.1109/ICSM.2009.5306332
5. Lung C.H., Xu X., Zaman M., Srinivasan A. Program restructuring using clustering techniques. *Journal of Systems and Software*, 2006, vol. 79, no. 9, pp. 1261–1279. https://doi.org/10.1016/j.jss.2006.02.037
6. Srinivas C., Rao C.V.G. A feature vector based approach for software component clustering and reuse using k-means. *Proc. International Conference on Engineering and MIS (ICEMIS)*, 2015, pp. 1–5. https://doi.org/10.1145/2832987.2833080
7. Naseem R., Maqbool O., Muhammad S. Improved similarity measures for software clustering. *Proc. 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 45–54. https://doi.org/10.1109/CSMR.2011.9
8. Bavota G., Gethers M., Oliveto R., Poshyvanyk D., De Lucia A. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology*, 2014, vol. 23, no. 1, pp. 2559935. https://doi.org/10.1145/2559935
9. Al Dallal J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 2012, vol. 54, no. 10, pp. 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004
10. Singh S., Kaur S. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 2018, vol. 9, no. 4, pp. 2129–2151. https://doi.org/10.1016/j.asej.2017.03.002
11. Wang Y., Yu H., Zhu Z., Zhang W., Zhao Y. Automatic software refactoring via weighted clustering in method-level networks. *IEEE Transactions on Software Engineering*, 2018, vol. 44, no. 3, pp. 202–236. https://doi.org/10.1109/TSE.2017.2679752
12. Hegedűs P., Kádár I., Ferenc R., Gyimóthy T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 2018, vol. 95, pp. 313–327. https://doi.org/10.1016/j.infsof.2017.11.012
13. Kebir S., Borne I., Meslati D. A genetic algorithm-based approach for automated refactoring of component-based software. *Information and Software Technology*, 2017, vol. 88, pp. 17–36. https://doi.org/10.1016/j.infsof.2017.03.009
14. Han A.-R., Bae D.-H., Cha S. An efficient approach to identify multiple and independent Move Method refactoring candidates.

Литература

1. Keshta I.M. Software refactoring approaches: A survey // International Journal of Advanced Computer Science and Applications. 2017. V. 8. N 11. P. 542–547.
2. Alkhalid A., Alshayeb M., Mahmoud S.A. Software refactoring at the class level using clustering techniques // Journal of Research and Practice in Information Technology. 2011. V. 43. N 4. P. 285–306.
3. Srinivas C., Radhakrishna V., Rao C.V.G. Clustering software components for program restructuring and component reuse using hybrid XNOR similarity function // Procedia Technology. 2014. V. 12. P. 246–254. https://doi.org/10.1016/j.protcy.2013.12.482
4. Fokaefs M., Tsantalis N., Chatzigeorgiou A., Sander J. Decomposing object-oriented class modules using an agglomerative clustering technique // Proc. IEEE International Conference on Software Maintenance (ICSM). 2009. P. 93–101. https://doi.org/10.1109/ICSM.2009.5306332
5. Lung C.H., Xu X., Zaman M., Srinivasan A. Program restructuring using clustering techniques // Journal of Systems and Software. 2006. V. 79. N 9. P. 1261–1279. https://doi.org/10.1016/j.jss.2006.02.037
6. Srinivas C., Rao C.V.G. A feature vector based approach for software component clustering and reuse using k-means // Proc. International Conference on Engineering and MIS (ICEMIS). 2015. P. 1–5. https://doi.org/10.1145/2832987.2833080
7. Naseem R., Maqbool O., Muhammad S. Improved similarity measures for software clustering // Proc. 15th European Conference on Software Maintenance and Reengineering. 2011. P. 45–54. https://doi.org/10.1109/CSMR.2011.9
8. Bavota G., Gethers M., Oliveto R., Poshyvanyk D., De Lucia A. Improving software modularization via automated analysis of latent topics and dependencies // ACM Transactions on Software Engineering and Methodology. 2014. V. 23. N 1. P. 2559935. https://doi.org/10.1145/2559935
9. Al Dallal J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics // Information and Software Technology. 2012. V. 54. N 10. P. 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004
10. Singh S., Kaur S. A systematic literature review: Refactoring for disclosing code smells in object oriented software // Ain Shams Engineering Journal. 2018. V. 9. N 4. P. 2129–2151. https://doi.org/10.1016/j.asej.2017.03.002
11. Wang Y., Yu H., Zhu Z., Zhang W., Zhao Y. Automatic software refactoring via weighted clustering in method-level networks // IEEE Transactions on Software Engineering. 2018. V. 44. N 3. P. 202–236. https://doi.org/10.1109/TSE.2017.2679752
12. Hegedűs P., Kádár I., Ferenc R., Gyimóthy T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset // Information and Software Technology. 2018. V. 95. P. 313–327. https://doi.org/10.1016/j.infsof.2017.11.012
13. Kebir S., Borne I., Meslati D. A genetic algorithm-based approach for automated refactoring of component-based software // Information and Software Technology. 2017. V. 88. P. 17–36. https://doi.org/10.1016/j.infsof.2017.03.009
14. Han A.-R., Bae D.-H., Cha S. An efficient approach to identify multiple and independent Move Method refactoring candidates // Information and Software Technology. 2015. V. 59. P. 53–66. https://doi.org/10.1016/j.infsof.2014.10.007

Information and Software Technology, 2015, vol. 59, pp. 53–66. https://doi.org/10.1016/j.infsof.2014.10.007

15. Rathee A., Chhabra J.K. Restructuring of object-oriented software through cohesion improvement using frequent usage patterns. *ACM SIGSOFT Software Engineering Notes*, 2017, vol. 42, no. 3, pp. 1–8. https://doi.org/10.1145/3127360.3127370
16. Alkhalid A., Alshayeb M., Mahmoud S. Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm. *Advances in Engineering Software*, 2010, vol. 41, no. 10–11, pp. 1160–1178. https://doi.org/10.1016/j.advengsoft.2010.08.002
17. Rao A.A., Reddy K.N. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. *International Journal of Computer Science Issues*, 2011, vol. 8, issue 5, no. 2, pp. 185–194.
18. Wang Y., Yu H., Zhu Z., Zhang W., Zhao Y. Automatic software refactoring via weighted clustering in method-level networks. *IEEE Transactions on Software Engineering*, 2017, vol. 44, no. 3, pp. 202–236. https://doi.org/10.1109/TSE.2017.2679752
19. Rathee A., Chhabra J.K. Clustering for software remodularization by using structural, conceptual and evolutionary features. *Journal of Universal Computer Science*, 2018, vol. 24, no. 12, pp. 1731–1757.
20. Alizadeh V., Kessentini M. Reducing interactive refactoring effort via clustering-based multi-objective search. *ASE 2018 – Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 464–474. https://doi.org/10.1145/3238147.3238217
21. Bobde S., Phalnikar R. Restructuring of object-oriented software system using clustering techniques. *Proceeding of International Conference on Computational Science and Applications*, Springer, 2020, pp. 419–425. https://doi.org/10.1007/978-981-15-0790-8_41
22. Al Dallal J. Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology*, 2017, vol. 92, pp. 105–120. https://doi.org/10.1016/j.infsof.2017.07.013
23. Al Dallal J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology*, 2012, vol. 54, no. 10, pp. 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004

15. Rathee A., Chhabra J.K. Restructuring of object-oriented software through cohesion improvement using frequent usage patterns // ACM SIGSOFT Software Engineering Notes. 2017. V. 42. N 3. P. 1–8. https://doi.org/10.1145/3127360.3127370
16. Alkhalid A., Alshayeb M., Mahmoud S. Software refactoring at the function level using new Adaptive K-Nearest Neighbor algorithm // Advances in Engineering Software. 2010. V. 41. N 10–11. P. 1160–1178. https://doi.org/10.1016/j.advengsoft.2010.08.002
17. Rao A.A., Reddy K.N. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique // International Journal of Computer Science Issues. 2011. V. 8. Issue 5. N 2. P. 185–194.
18. Wang Y., Yu H., Zhu Z., Zhang W., Zhao Y. Automatic software refactoring via weighted clustering in method-level networks // IEEE Transactions on Software Engineering. 2017. V. 44. N 3. P. 202–236. https://doi.org/10.1109/TSE.2017.2679752
19. Rathee A., Chhabra J.K. Clustering for software remodularization by using structural, conceptual and evolutionary features // Journal of Universal Computer Science. 2018. V. 24. N 12. P. 1731–1757.
20. Alizadeh V., Kessentini M. Reducing interactive refactoring effort via clustering-based multi-objective search // ASE 2018 – Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018. P. 464–474. https://doi.org/10.1145/3238147.3238217
21. Bobde S., Phalnikar R. Restructuring of object-oriented software system using clustering techniques // Proceeding of International Conference on Computational Science and Applications. Springer, 2020. P. 419–425. https://doi.org/10.1007/978-981-15-0790-8_41
22. Al Dallal J. Predicting move method refactoring opportunities in object-oriented code // Information and Software Technology. 2017. V. 92. P. 105–120. https://doi.org/10.1016/j.infsof.2017.07.013
23. Al Dallal J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics // Information and Software Technology. 2012. V. 54. N 10. P. 1125–1141. https://doi.org/10.1016/j.infsof.2012.04.004

Authors

Sarika Bobde — Associate Professor, MIT World Peace University, Pune, Maharashtra, 411038, India, SC 57193133803, https://orcid.org/0000-0002-6693-1364, sarikabobde27@gmail.com

Rashmi Phalnikar — Research Scholar, MIT World Peace University, Pune, Maharashtra, 411038, India, SC 26436009300, https://orcid.org/0000-0002-2004-7944, rashmi.phalnikar@mitwpu.edu.in

Авторы

Бобде Сарика — доцент, Школа компьютерной инженерии и технологий, MIT — Всемирный университет, Пуна, Махараштра, 411038, Индия, SC 57193133803, https://orcid.org/0000-0002-6693-1364, sarikabobde27@gmail.com

Пхальникар Рашми — исследователь, Школа компьютерной инженерии и технологий, MIT – Всемирный университет, Пуна, Махараштра, 411038, Индия, SC 26436009300, https://orcid.org/0000-0002-2004-7944, rashmi.phalnikar@mitwpu.edu.in

Received 14.05.2021
Approved after reviewing 16.09.2021
Accepted 11.10.2021

Статья поступила в редакцию 14.05.2021
Одобрена после рецензирования 16.09.2021
Принята к печати 11.10.2021

Р бот доступн по лицензии
Creative Commons
«Attribution-NonCommercial»