Information Systems Frontiers
https://doi.org/10.1007/s10796-019-09948-4

Mining Reusable Software Components from Object-Oriented Source Code using Discrete PSO and Modeling Them as Java Beans

Amit Rathee[1] · **Jitender Kumar Chhabra**[1]

Abstract

Object-based software development in the IT-industry during the last decade has focused on reusing the already developed elements. Current software development models and technologies promote software reuse by utilizing component-oriented programming. Component-oriented programming provides more effective reuse as compared to its object-oriented language counterpart. This is because a component provides more abstractness in terms of the functionality in contrast to the object-oriented software element (e.g. class/ interface). The component-based development is highly dependent on the availability of the component library that contains well-defined and already tested functionality in terms of the available components. However, the suitable availability of such component library is still a problem. The lack of a suitable component library is tackled either by designing new components from scratch or by identifying them from the existing software systems. These identified components must be transformed as per component models, such as a Java Bean, in order to increase software reuse and faster software development, and little research has been carried out in this direction. Although many approaches have been proposed to identify components from existing systems and build a component library. But, such approaches lack in transforming the identified components as per specifications of a component model. Further identification of reusable elements also needs to be more accurate and efficient. Accuracy is dependent on the structural and semantic properties of a component. A component can consist of a single or a group of individual elements of the software, and thus identifying components from a set of all elements becomes an exponentially growing problem. Such problems can be more efficiently solved using metaheuristic algorithms. Therefore, this paper targets identifying accurately the components from a singular existing software system and transform those identified components to the Java Bean component model standard. The component identification approach of this paper is based on clustering the underlying software system by utilizing a metaheuristic search-based PSO algorithm. The clusters are identified as a group containing 2/ 3/ 4 software elements. Moreover, the used fitness function is based on the structural (based on usage for member variables) and semantic relations belonging to a software. Since the PSO algorithm is generally applied to continuous problems, whereas our formulated component identification problem is discrete (due to the representation of particles) in nature. Therefore, we have also provided a mechanism through which the PSO algorithm is modified to become applicable to discrete problems. From the obtained clusters (components), the interfaces of the components are identified by identifying independent chains in call-graphs of the underlying components. Finally, these components are transformed as per the Java Bean component model specifications. The proposed approach is validated experimentally on 7 open-source object-oriented Java software systems and results are compared with competitive approaches. The obtained results confirm the correctness of the transformed component and hence their reusability.

Keywords Reusable component · Frequent usage patterns · Discrete PSO · Component extraction · Java beans · Modelling · Interface identification

1 Introduction

Development of software using object-oriented languages has brought a revolution in the IT industry during the last few decades. The reasons behind this success are the object-oriented language features such as abstraction and encapsulation that help in reusability. These features provide

✉ Amit Rathee
amit1983_rathee@rediffmail.com

Extended author information available on the last page of the article.

Published online: 29 August 2019

flexibility and extensibility at the end of the developers. However, with the larger and complex software systems, the object-oriented languages fail to keep their expectation regarding the productivity in terms of the reuse. This is because such complex systems do not possess any explicit architecture that can be reused and generally have numerous complicated internal dependencies (Alshara et al. 2015). These limitations are solved by using another alternate development approach known as component technology or Component Based Software Engineering (CBSE). The key composing unit in CBSE is called a component and it possesses larger granularity as compared to classes/ objects in object-oriented languages. This component provides isolated and encapsulated functionality as a service to others. This service can be easily adapted and reused without much effort at the developer's end (Emmerich and Kaveh 2001).

In CBSE, component and connectors are the key composing elements for defining software architecture and it fastens the application development cycle by reusing existing components rather than developing them from scratch. When reusing components they must be available in the form of a library (such as component on the shelves) or they must be identified from other already existing systems. Hence, the process of identification of reusable components (from the system's underlying source-code) is a key research area in CBSE (Birkmeier and Overhage 2009). This requires that the software be analyzed in order to extract information that represents the behavioral and structural perspectives of the software. This information is beneficial for the project manager during project management. Maximum cohesion and minimum coupling are the thumb rules for good quality of software design. The information about both of these characteristics of the software needs to be formally extracted and processed efficiently. All such information should be stored in some information system so that project managers can take appropriate decisions through such information. Using the structural dependency information, the project manager can determine the accurate group of software elements in a project that are functionally related. Similarly, the semantic information can help the project manager in maintaining semantic consistency within a component. Many approaches are proposed in the literature that focuses on recovering components from the existing legacy systems using its underlying source- code (Hamdouni et al. 2010; Shatnawi and Seriai 2013; Kebir et al. 2012; Allier et al. 2011; Alshara et al. 2015). These approaches focus on identifying a component as a set of different software elements that together provides a number of functions to other components. However, these approaches have several limitations. Firstly, some of the approaches do not specify how the interfaces of the identified components are identified (Hamdouni et al. 2010). Although, identification of the interface is mandatory for explicit specification of the services of the underlying component. Secondly, either they are relying on the selection of client software or they miss the semantic relations during component identification phase. Almost, all of these approaches specify a component, only informally. The informal description of a component includes what software elements belong to it and its corresponding interface description (Yacoub et al. 1999). As such, these components are not directly reusable. However, to make them directly reusable their specification must be transformed into some component models such as Java Beans, Enterprise Java Bean, Microsoft COM, and CORBA (Heisel et al. 2002). This transformation helps in binding the underlying object-oriented code to some component model specifications. The transformed component can be directly composed with other components in the system and these can be directly introspected by third-party tools. Such a transformed component is directly reusable for the leading IT developers and the pioneering researchers in academics.However, to the best of the author's knowledge, no approach targets modeling the identified component as Java Beans. A component that is modeled as a Java Bean and is available in the form of the component library is directly reusable during the development lifecycle of a new software system.

Therefore, this paper targets the two main aspects of CBSE. The first aspect is the identification of high-quality components by clustering different correlated software elements of an object-oriented software system. This paper gives due importance to the semantic integrity of a component along with other structural relations (Hamdouni et al. 2010). The clustering of correlated software elements is formulated as a metaheuristic search-based optimization problem and is solved using the metaheuristics PSO algorithm. The second aspect of this paper is towards the transformation of the object-oriented component to Java Bean component model specification. The proposed approach has both technical and practical contributions from information extraction, processing, and management viewpoint. Technically, we propose a new component identification approach that extracts, stores, and processes the information about the structural, semantic and evolutionary characteristics of the Java-based software. All of this information is dynamic in nature and varies with the maintenance of the software. This paper represents this information efficiently and groups them using discrete PSO based clustering technique. Practically the proposed approach has lots of industrial applications especially for project managers of IT industry, who can create or add to their own repository of reusable Java beans, which are reusable for future developments. Further, the proposed approach can be implemented in form of a professional tool, which any IT company can use in house forextracting useful components out of company's various Java-based software and transform them into Java

beans using the information extracted and processed form source code and change history repositories. The significant research additions of this paper are summarized below:

1. The problem of identifying the reusable component from the source code of the legacy system is represented as a metaheuristic search-based optimization problem. Here, for the search-based representation of the problem, both the structural and semantic dependency relations present in a software system are taken into account.
2. The underlying component identification problem is a discrete-parameter metaheuristic problem and this paper targets to solve it using the PSO algorithm, which is a continuous-parameter, based algorithm. Hence, this paper proposes a modified and discrete version of the search-based PSO algorithm.
3. A new technique is proposed here for determining the provides interface of the identified component based on the finding of independent chains in the call-graph of the identified component.
4. A new technique is designed to transform the obtained component structure as per the specifications of the Java Bean component model. This is a first attempt in the literature (to the best knowledge of the authors) to obtain reusable elements as Java Beans components.

The rest of the paper is organized into the following sections: Section 2 summarizes several related approaches already proposed in the literature. Section 3 mentions the details of the proposed approach of this paper. Section 4 gives details of how the experimental test is conducted so as to justify the significance of the proposed approach of this paper. Further, the Section 5 gives experimental details and presents the obtained results. Section 6 mentions various threats of the validity of the proposed approach. Finally, in the Section 7 we provide conclusions and relevant possible future work.

2 Literature Survey

Reusable component identification is a key step for the successful implementation of CBSE and hence is mandatory for the promotion of software reuse principle. Therefore, this section of the paper gives details about various techniques that are already proposed by different researchers. This paper considers summarizing the literature related to our proposed approach into the following three sub-sections:-

2.1 Software Remodularization and Dependency Estimation

In this sub-section, we consider summarizing those approaches and techniques of literature that are focused on dependency estimation among different software elements and its remodularization/ clustering. The process of software remodularization helps in improving maintenance efforts, quality and promotes software reuse (Ducasse and Pollet 2009). The authors in Praditwong et al. (2011) proposed a search-based approach for software module clustering by formulating two multi-objective approaches, namely MCA (Maximizing Cluster Approach) and ECA (Equal-size Cluster Approach). Both of these formulated approaches optimized two objectives related to high cohesion and low coupling quality requirement of a software system. Similarly, the authors in Bavota et al. (2012) proposed a software re-modularization approach by integrating human (developer's) knowledge during the genetic-based re-modularization solution. The authors used a fitness function extended from MQ (Modularization Quality) metric by introducing penalization score based on developer feedback. The authors in Prajapati and Chhabra (2017, 2019b) suggested an approach for improving the underlying modular structure of a software system by proposing a new many-objective discrete harmony search algorithms that are based on the structural and lexical dependency relations present in a software system. Similarly, they extended their work in Prajapati and Chhabra (2018a, b) and proposed many-objective Artificial Bee Colony (ABC) algorithms for the re-modularization of large-sized software systems. In Prajapati and Chhabra (2019a), the authors proposed an information-theoretic based software remodularization approach. They formulated their proposed approach as many-objective search-based approach. Mu et al. proposed an hybrid GA (Genetic Algorithm) based software remodularization approach (Mu et al. 2019). Here, the authors proposes a new mathematical programming model to overcome the problem of over-cohesiveness during the remodularization process. Rathee et al. proposed the cohesion improvement of software using a new technique for the-measurement of the structural dependencies in a software system. This new technique is called Frequent Usage Pattern (FUP) (Rathee and Chhabra 2018b).

2.2 Software Component Identification

In this sub-section, we consider summarizing those approaches that are directly aimed at reusable component identification or recovering component-based architecture of the software system. The authors in von Detten et al. (2014) proposed an architecture mining approach for the legacy software systems using the Archimetrix approach. The proposed approach aimed at clustering the underlying classes into components. Shatnawi et al. proposed a mining approach for component identification using the source code of a set of similar domain software systems. Similarly, the authors in Chardigny et al. (2008) proposed

a quasi-automatic component-based architecture recovery approach called ROMANTIC. The proposed approach is dependent on the structural and semantic characteristics of the underlying software system. The authors in Shatnawi et al. (2018) proposed an approach for identifying object-oriented reusable component from underlying APIs using the dynamic analysis of the API. Here, the dynamic analysis is based on the synergy between the corresponding API and different considered client software that is directly based on the API. Similarly, Seriai et al. also proposed a component identification approach based on the dynamic information extracted from the execution traces of the system (Seriai et al. 2014). The authors also proposed a mechanism to identify the required interfaces (provides and requires) of the identified components. The authors claim that the identified software is directly reusable in the concrete component framework. Kessel et al. proposed a novel approach for ranking software components by utilizing non-functional properties (Kessel and Atkinson 2016). Rathee et al. proposed a multi-objective search-based reusable component identification approach based on NSGA-III clustering algorithm (Rathee and Chhabra 2019a). Ilk et al. stressed that existing software components can be efficiently utilized if the business semantic knowledge of the component is known (Ilk et al. 2011). Based on this, the authors proposed an automated approach that generated the business semantics for the source code components. The authors in Rathee and Chhabra (2019b) provided an approach for the identification of multimedia components using two kinds of relations namely FUP and semantic. The authors in Garriga et al. (2018) presented a structural-semantic based service retrieval and selection approach that can be used in the service-oriented computing environment.

2.3 Modeling Components to Component Model Standard

In this sub-section of the paper, our focus is to summarize the literature work that aimed at modeling components as Java Beans specification or any other component models. Song et al. proposed a specific approach for extracting the Enterprise Java Beans (EJB) components only from the Java Servlet source code (Song et al. 2002). The authors in Lorenz and Vlissides (2001) suggested an architectural transformation approach that maps a Java class into its corresponding Java Bean component. However, the authors considered a single class as a single component and did not consider the grouping of classes that can be more reusable as per our consideration. The authors in Washizaki and Fukazawa (2005) pproposed a refactoring based approach that converts a Java software system into a Java Bean framework. They represented the software as a dependency graph based on class relations. Further, they applied the Façade interface for defining the interface of a component. Here, it can be noted that their approach did not consider all kinds of dependency relation while refactoring the software into components. Allier et al. proposed a transformation approach that converts a Java application to OSGi standard (Allier et al. 2011). They used Facade and Adapter design patterns for defining the component interfaces.

This paper identifies various research gaps based on the literature study and these are as follows:

1. Accurate computation and integration of FUP and semantic relation is an open research problem in component identification and no existing approaches, test the feasibility of FUP and semantic relations together.
2. The existing approaches identify the interface of a component as a set of all the methods present in the classes belonging to the component. This is a more or less common approach and it is the belief of the authors that the actual interface set is generally a subset of it.
3. The existing approaches that model the identified component as Java Bean are based on the use of Façade and Adapter design patterns only. None of the existing approaches considers transforming the component to Java Bean API specifications and targets the introspection, customization and persistence feature of Java Bean.
4. Many component-based architecture recovery approaches are directly based on the good choice of client software. Here, inappropriate selection can highly deviate the result of the underlying approach.

Based on the literature study, to the best of our knowledge, we conclude that no direct reusable component identification approach is available in the literature that is based on FUP and semantic dependency relations together and models the identified components into the Java Beans.

3 Proposed Approach

This section gives details about the proposed approach that focuses on mining/ identifying reusable components from an object-oriented software and modeling them as Java Beans. The proposed approach aims to reengineer the object-oriented system into its corresponding component-based system. The identified components fulfill the Java Beans component model standards. The proposed approach is beneficial for the project manager as it extracts and efficiently processes various structural and semantic information about the underlying project. This information can be used for effective management of theproject. Figure 1 diagrammatically depicts the proposed approach of this paper. The proposed approach consists of four steps, namely

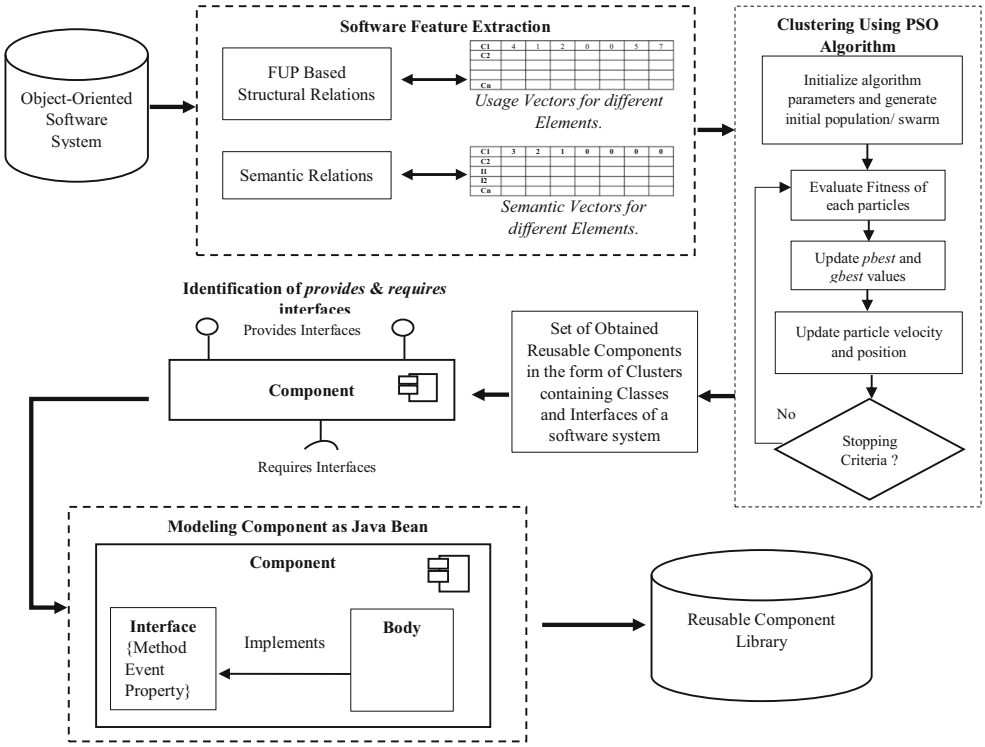

Fig. 1 Proposed approach for reusable component mining and modeling them as Java Beans

(1) Extraction of different Software features at the software element level, (2) Clustering different software elements using PSO, (3) Identification of different interfaces for each of the obtained components, and (4) Modeling different components as Java Beans standard components. Each of these steps is detailed in the following sub-sections below:

3.1 Software Features Extraction

Software feature extraction is the primitive and most important step in the approach depicted in Fig. 1. In our proposed approach, we consider a reusable component as a group of software elements (class/ interface) that belongs to the software and this group together provides a more coarse-grained functionality in the system. Such a group is supposed to possess high cohesion and low coupling. This group, containing functionally related classes, is identified based on two features present between the software elements, namely (1) Frequent Usage Pattern (FUP) relation and (2) the semantic relation (Rathee and Chhabra 2018b; Shatnawi et al. 2017; Rathee and Chhabra 2017). The FUP based relation (FUPR) is basically the structural relation and it denotes the group of software elements that are functionally related to a given element. The FUPR is identified as the set of member variable names that are used by a software element either directly or through function calls. The FUPR helps us group software elements such that they are functionally related. The semantic relation (SR) is a non-structural relation and is used to group semantically related software elements. The SR is measured by tokenizing the source-code and extracting the tokens from two parts, namely (1) the class name declaration statements, (2) the function definition statements. The class name declaration statements give the semantic information about the class names and usage of the inheritance feature if any. The function definition statements give semantic information about the corresponding functions, types of their parameters and the return types. The FUPR and SR are identified as the set of member variable names and the token names respectively. In our proposed approach, these two types of dependency relations are determined at the level of software element i.e. class/ object level.

The next step after determining these two relations is to represent them in the form of vectors namely FUPR Vector and SR Vector separately for each software element. The FUPR vector is the frequency-based representation of the corresponding FUPR set. The size of FUPR vector is and denotes count of unique features (member variables) belonging to the source-code of the software. Similarly, the SR vector represents the SR set and its size is. Here, represents the size of the unique tokens identified at the software level. If a software element does not use any of the other element (s), then, the corresponding value in the vector for them is zero. Figure 1 diagrammatically represents

this idea and this representation is used to compute the underlying quality (in terms of cohesion and coupling) of the identified component during the later stages of the proposed approach.

3.2 Clustering using PSO Algorithm

In the proposed approach, the grouping of various software elements that ultimately constitutes the reusable component is done using a non-deterministic meta-heuristic search algorithm called Particle Swarm Optimization (PSO) [Prajapati2018]. PSO is a swarm based optimization algorithm and it is inspired by the bird flocking behavior. For our reusable component identification from source-code, the PSO algorithm, especially suits because of the following reasons:

1. Reusable component identification is basically a combinatorial problem that involves grouping of individual software elements. Moreover, the curse of dimensionality of the system further worsens the situation. The PSO algorithm is simple to implement and requires little memory for implementation (Gong et al. 2017).
2. In reusable component identification, the software elements belonging to a component must be related to each other. This requires that the clustering is dependent on other neighboring elements present in the system (whether or not they are related to each other). In the PSO algorithm, an individual particle does not directly solve the problem and the population is organized as a communication structure/ topology. In this topology, communication takes place through the bidirectional edges present among a pair of neighboring particles (Shan et al. 2006). Hence, it suits our component identification problem.

Further, the reason for choosing the PSO algorithm over other evolutionary algorithms is that it has shown promising performance, is easy to understand as well as implement and also provides balanced exploration and exploitation (Nebro et al. 2008; Tydrichova 2017; Prajapati and Chhabra 2018c). The two main factors that differentiate it from others are population size (generally chosen to be smaller than competitive approaches such as NSGA-III), and a good distribution of the solutions (more diversity due to two kinds of populations namely pBests and current positions). Moreover, the PSO algorithm possesses a faster convergence speed. In addition, the computational time of the PSO algorithm is lower due to its limited number of parameters and also no crossover and mutation operators are involved. Figure 1 also depicts the algorithmic steps that are followed in the PSO based optimization for clustering various software elements in groups. Each of such obtained clusters is considered as a different identified reusable software component. The various contents belonging to the component are further used to identify *provides* and *requires* interfaces. Different steps involved in the PSO based clustering and discretization of the PSO algorithm are discussed in below sub-sections:

3.2.1 PSO Particle Initialization

In the PSO based solution of the clustering problem, the individual particle is expressed as a linear vector of size N. Here, N represents the total individual elements associated with the considered software. Figure 2 depicts the basic idea behind the PSO based clustering along with how the individual particle is represented in the search space. Here, the considered hypothetical system contains say e.g. six software elements, namely $E_1, E_2, ..., E_5, E_6$ that are represented as nodes in the graph along with the FUPR (shown as solid lines) and SR (shown as dotted lines) represented as edges between the nodes. In the typical representation used for the particle, each of the software elements is assigned a unique fixed position in the array indexes and the corresponding value at that index denotes the cluster number to which it belongs. Since, coarse-grained components may be of different sizes and hence, they contain an unequal number of classes. Based on the literature, the average size of an identified component varies from two to four (Shatnawi et al. 2017, 2018;

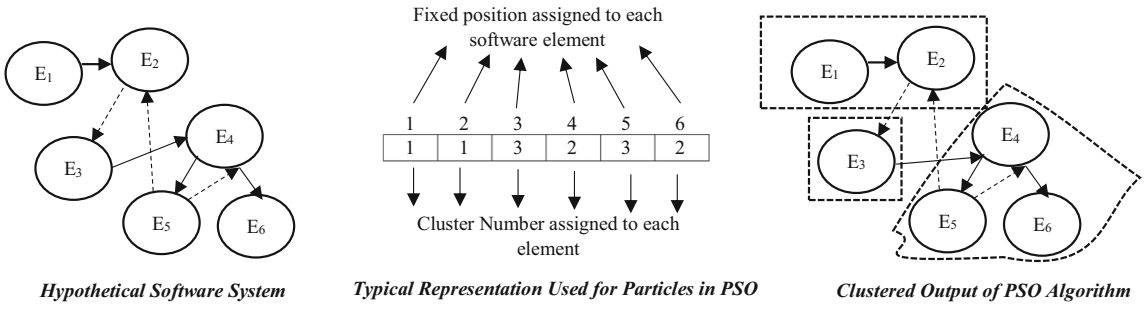

Fig. 2 Representation of particle and typical working of PSO algorithm

Washizaki and Fukazawa 2005). This motivates us to perform variable sized clustering. Therefore, in our PSO based clustering solution, the cluster number of different elements are assigned in the pair of 2, 3, or 4 software elements. For example, in the hypothetical example of Fig. 2, the particle is initialized by considering the cluster numbers assigned as a pair of two software elements. Hence, there are three total obtained clusters in the PSO algorithm output. Similar is the case of particle initialization for the pair of three (containing two obtained clusters), and four (again containing two obtained clusters). In our proposed approach, the association of different elements to different groups is decided randomly initially. This randomization helps in better search space exploration. Each of these pairwise initialization is done in the different run of the PSO algorithm. Finally, the *gbest* particles of each run are analyzed and the final clustering solution is obtained that contains reusable components as a pair of 2, 3, and/ or 4 software elements.

3.2.2 Particle Position Updating Rules

The PSO algorithm is originally designed to solve continuous problems. However, the proposed representation of particles considers discrete integer values. These values represent the cluster numbers. Therefore, it is important to map the continuous search space domain to a discrete domain. In our proposed approach, this mapping is done while updating the position of different particles after evaluating their fitness values. The position (S) and the velocity (V) of a particle at index t is updated using the Eqs. 1 and 2.

$$S_i(t) = Math.Round(S_i(t) + V_i(t)) \tag{1}$$

$$\begin{aligned} V_i(t) \;=\; & W * V_i(t) + C_1 * r_1 * (S_{lbest}(t) - S_i(t)) \\ & + C_2 * r_2 * (S_{gbest}(t) - S_i(t)) \end{aligned} \tag{2}$$

Here, the velocity of a particle is updated using the same formula as proposed in Sierra and Coello Coello (2005) and the variables $W, C_1, C_2, r_1,$ and r_2 are chosen as per the recommendation given by the authors in Sierra and Coello Coello (2005). The $S_{lbest}(t)$ and $S_{gbest}(t)$ are the corresponding position values for the local and global best particles in the PSO algorithm. Since, velocity value is always a real number, therefore, *Math.Round* function is used while updating the position of a particle. This function converts the real value to the closest whole number either up or down. This helps us to map the continuous search domain to the discrete domain. The basic requirement of our proposed approach is that the particles at any stage must contain software elements only in the pairs of either 2, 3, or 4 and not a mixture of these values. Therefore, it is always checked whether after generating new position values for a particle, the values adhere to the basic requirement or not. If the basic requirement is not met, then, the two actions are taken. Firstly, if the cluster possesses more than the needed paired number of elements, then the exact number of paired elements are randomly chosen and rest are left out. Secondly, if the cluster possesses less than the required paired number of elements, then the exact number of paired elements is completed by randomly choosing the left out software elements.

3.2.3 Particle Fitness Functions

During meta-heuristic search-based clustering, the determination of clustering evaluation criteria is a crucial issue. This choice directly affects the obtained clustering results of the search-based algorithm (e.g. PSO). In our proposed approach, the PSO based clustering focuses on clustering elements such that the obtained clusters possess higher cohesion and least coupling characteristics. In our approach, during any iteration of the PSO algorithm, the evaluation of the underlying clustering quality (as indicated by particle representation) is based on three conflicting component quality criteria. The first quality criteria is the number of intra-cluster usage based dependency (to be maximized). The second quality criteria is the number of inter-cluster usage based dependency (to be minimized). These wo quality criteria are computed from the *Usage Vectors* based on FUPR. Finally, the third quality criteria is the semantic-based cohesion computed for the identified cluster/ component. This semantic based cohesion is measured using the *semantic vectors* and the modified cosine similarity measure as depicted in Eq. 3. Here, n is the total number of elements belonging to the component C_j and $w_{k,i}$ is the weight of k^{th} unique token belong to the i^{th} element.

$$SemCoh(C_j) = \frac{\sum_{i=1}^{n}(w_{(j,i)} * w_{(k,i)} * ..... * w_{(t,i)})}{\sqrt{\sum_{i=1}^{n} w_{(j,i)}^2}\sqrt{\sum_{i=1}^{n} w_{(k,i)}^2}\sqrt{\sum_{i=1}^{n} w_{(t,i)}^2}} \tag{3}$$

Based on the three quality criteria, we defined a multiplicative aggregative fitness function based on the structural and semantic objectives. The proposed fitness function for a solution containing N clusters is depicted in Eq. 4.

$$fitness(f) = Average\left(\sum_{i=1}^{N}\left(\frac{UD_{INTRA}(i)}{UD_{INTER}(i)+UD_{INTRA}(i)} * SemCoh(C_i)\right)\right) \tag{4}$$

Here, $UD_{INTRA}(i)$ and $UD_{INTER}(i)$ are the total number of member variable references within and outside the given cluster i. The $Average$ function helps determine the fitness function value at the component level by averaging it on the component level.

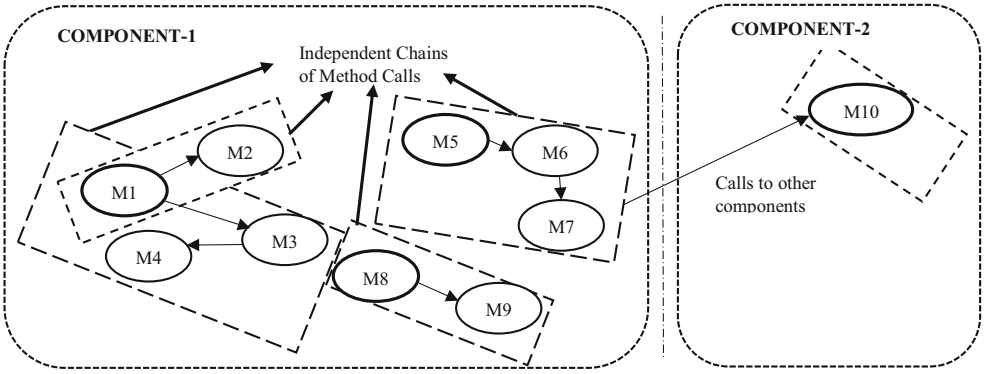

Fig. 3 Call graph and independent chains of method calls

3.3 Identification of Provides and Requires interfaces of a Component

In a component-based software development approach, a software system is developed by integrating loosely coupled independent reusable components. This composition is based on the *p*rovides and *r*equires interfaces provided by the component. These interfaces define which services a component serves to others and which of the services it needs from other components in the system. In the proposed approach, these interfaces are identified as the set of method names. These sets are identified by representing the identified component as a *call graph*. The *call graph* is a control flow graph that represents the calling relationships between the methods belonging to different elements of a component. After representing the component in the form of a call graph, its interfaces are identified by finding the independent chains of call. The starting method in each of such identified chains constitutes the provides interface set. The method names that belong to other components in the system and are used by a component constitutes its requires interface. Figure 3 diagrammatically shows the call graph for a hypothetical component having a total of say e.g. nine methods belong to various software elements present in a component. In the considered example, the independent chains of method calls are also indicated by the dotted line rectangles. Here, the provides interface is the set containing methods M1, M1, M5, M8 which are the starting point of call chains (indicated by solid lines). Similarly, the method M10 belonging to component-2 constitutes the requires interface for the considered example.

3.4 Modeling Component as Java Bean

Components are the piece of software that can be independently deployable. However, the components that are obtained after applying PSO based clustering are not directly reusable in nature. To make them directly reusable, it is necessary to have a description of its properties that abstract its definition from the underlying implementation. There are several component models available that provides an abstract description such as *Java Beans, Enterprise Java Beans, Microsoft COM,* and *C*ORBA. In this paper, we provide a model that maps a component (identified as an obtained cluster in PSO based optimization) as a Java Beans standard. The Sun Microsystems originally develops the Java Beans component model in the year 1996. According to its official definition, a *Java Bean is a reusable software component that can be manipulated visually in a builder tool*. Figure 4 shows a general description of the common characteristics of a Java class and a Java Bean. The key differences between a Java Class and a Java Bean/ Component are as follows:

1. A Java Bean is always a *p*ublic class so that these can be easily instantiated by the third party application system or builder tools.
2. A Java Bean must have a no-argument constructor.
3. A Java Bean can have getter and setter methods that allow its customization by permitting the changes in the properties of a bean.
4. A Java Bean state must be serializable so that its state is persistent.

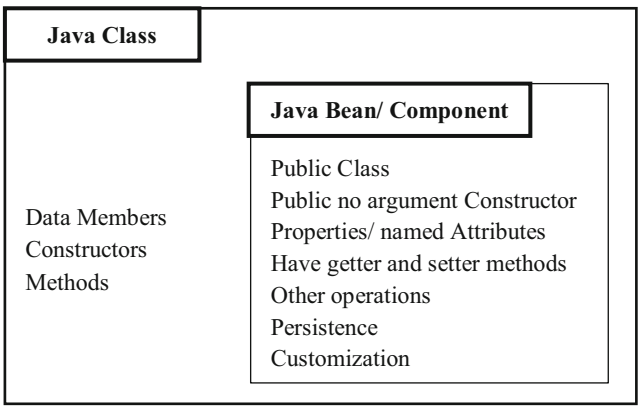

Fig. 4 General characteristics and difference between Java Class and Java Bean

Further, the other unifying features of a bean are (1) *I*ntrospection- so tools can analyze a bean, (2) *C*ustomization- so tools can alter a bean, (3) *p*ersistence. Figure 5 shows the proposed mapping that converts an obtained cluster of PSO algorithm to reusable Java Bean specification. Here, different software elements (classes) belonging to the cluster are modified in two steps: first, corresponding getter and setter methods are added, and secondly, each of these classes is implemented with the java.io.Serializable interface. The important Java Beans features and how they are modeled in the proposed approach are discussed in detail in below subsections:-

3.4.1 Introspection

The Java Bean API architecture provides support for introspection by defining classes that provide global specification about the underlying bean such as bean name, set of property names, set of methods. The introspection support helps the third party tool to gather information about the component while composing it along with other components. In the proposed modeling approach, the introspection support is provided by creating a new class that extends *java.beans.SimpleBeanInfo* class of Java Beans API that ultimately inherit from *java.beans.BeanInfo* interface. This class contains information about the component that we want to make visible to the third-party tools.

3.4.2 Customization

This property of a Java Bean helps the user to modify the appearance (only for GUI component) and the behavior of the bean at design time in order to meet the specific user need. In the proposed modeling approach, the customization support is provided by adding getter and setter methods that allow the third party tools to visualize the current property value and allow them to modify. Such methods are added to the original software element's source-code. This addition does not change the underlying original functionality of the element in any way. The added getter and setter methods follow a specific design pattern. For example, for a specific bean property named *property* that is of the type named *SomeType*, its corresponding getter and setter methods have the following signature:

SomeType get Property() *//getter method*
SomeType set Property(**SomeType value**) *//setter method*

3.4.3 Persistence

The persistence property of a bean allows it to store its properties, fields, and state information and retrieve it from storage. Persistence in the Java Bean is provided by the mechanism called *serialization*. Serialization helps convert the corresponding Bean object to a data stream and write it to the storage. In the proposed modeling approach, the persistence is supported by implementing different software elements with the *java.io.Serializable* interface.

4 Experimental Planning

This section of the paper provides details of the experimental setup conducted in order to validate the proposed approach. The proposed approach in this paper targets identifying components from existing legacy software system and ultimately model them as Java Beans standard.

4.1 Data Collection

To evaluate the feasibility of the proposed approach, experimentation is conducted on 7 real-world open-source

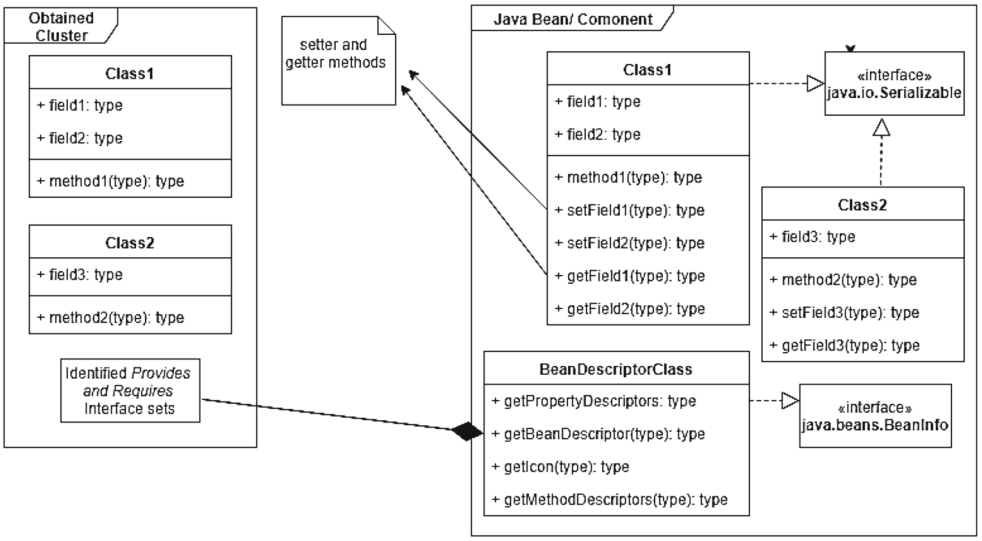

Fig. 5 Proposed mapping of a cluster to reusable Java Bean/Component

Table 1 Description of different Software Systems considered for experimentation

S. No.	Software system name	Version	# of software elements	Code size (KLOC)	Description
1	CheckStyle	6.5.0	897	63	A tool that helps adhere to Java Coding Standard.
2	JFreeChart	1.0.19	629	98	A Java framework for creating interactive and non-interactive charts.
3	Log4j	1.2.17	220	21	A Java based logging utility.
4	Ant	1.9.4	1233	135	Java library that helps build software.
5	android.view	Level 14	491	55	Android API that manages user interfaces.
6	android.app	Level 14	361	46	Android API for managing applications.
7	java.util	0.0.11	846	75	Java Utility API

Softwares. Table 1 shows various details regarding considered software such as *Version, Size, the total number of elements belonging to it, and a brief description*. These open-source software are considered as they are universally adopted among the research community and also they vary in terms of size from small to large systems. Further, the selected systems also belong to different domains. The software systems, namely *CheckStyle, JFreeChart, Log4j, and Ant* are selected from the standard *Qualitas corpus* (Tempero et al. 2010). Similarly, the software systems, namely *android.view, android.app, and java.util* are the standard API's from the *Android and the Java API* package system.

4.2 Formulated Research Questions

The approach of this paper is evaluated on the selected systems as identified in the above subsection. The experimental evaluation is carried out by following different steps. Firstly, each of the software systems is analyzed in order to identify two types of dependency relations viz FUPR and SR. This analysis is carried out using a manual tool designed by the authors specifically for this purpose. This tool returns the vector representations for each of these relations corresponding to each of the elements belonging to the system. In the second step, different software elements are clustered using the metaheuristic search-based PSO algorithm. The initial configuration and default values used for running the PSO algorithm is depicted in Table 2. Here, the initial position of different particles is randomly selected along with constraints related to pairing different software elements. The association of a software element with a cluster is randomly initialized such that the cluster size of different clusters is the same as the pairing size (2, 3 or 4 pairs) except the one. The PSO algorithm is implemented using the MOEA Framework.

In the third step, interfaces of different components obtained as different clusters in the previous step are identified. This identification is manually carried out based on the call graph of the underlying component. These interfaces are obtained as sets containing method names. Finally, the obtained components are modeled as a reusable component as per the Java Bean specification.

Further, the evaluation of the obtained Java Bean components is done by answering the following research questions that are formulated especially for this purpose:

RQ1. **Are the identified components reusable?** The proposed technique of this paper considers identifying components from underlying source-code, so we formulated this research question. This research question aims at evaluating the reusability of the identified components. Here, the evaluation is based on the fact that whether or not the underlying component contains related software elements grouped together.

RQ2. **Are the identified interfaces of the component useful and follow the component encapsulation principle?** As the proposed approach identifies different interfaces (provides & requires) of the component. Therefore, this research question is defined to evaluate the effectiveness of the interfaces. This evaluation helps determine the effectiveness of the component while composing it with other components.

Table 2 Parameter values for PSO algorithm

S.N.	PSO parameter used	Value used
1	Swarm Size	1.5 * N Where *N* represents the size of the software system.
2	Iterations	10,000
3	Initial Position	Random with Constraint on Pairing
4	Initial Velocity	Random
5	Self-Confidence (C1)	Random (1.5, 2.5)
6	Swarm-Confidence (C2)	Random (1.5, 2.5)
7	Inertia Weight (W)	Random (0.1, 0.5)

RQ3. **Is the proposed modeling semantically correct?** As the proposed approach also transforms the identified components as a Java Beans specification. This transformation results in the addition of new code to the underlying structure. Therefore, this research question is formulated to test whether or not the transformed component is semantically correct.

RQ4. **Is the proposed dependency measure (based on FUP and Semantic relations) capable of correctly identifying individual Java Bean components from a pool of multiple clusters of Beans?** The proposed approach targets grouping functionally related elements of a software system as a component and model it as a reusable Java Bean. Therefore, this research question is defined in order to test whether or not the used dependency scheme (one making combined use of FUP and semantic relations) is capable of separating components from a pool containing a mixture of various independent reusable components.

4.3 Evaluation Method

To answer RQ1, we need to determine the functional relatedness among different software elements (classes/interfaces) which are associated with the component. This relatedness is considered effective if the contents of a component are highly dependent on each other and loosely dependent on the elements belonging to other components. This can be measured as a ratio between intra component dependency and inter-component dependency. Mitchell et al. proposed a modularization metric called *TurboMQ* (Mitchell 2003). This metric provides the overall quality of the component. This measured quality is an optimum indicator between inter- and the intra- relations existing for a given component. This metric has been widely used in the literature to measure the obtained modularization quality (Rathee and Chhabra 2018a, 2019b). Hence, this metric is selected in this paper to evaluate the formulated research question RQ1.

To answer research question RQ2, the most important aspect is to determine how the object-oriented dependencies are handled in the interface-based reusable components identified from the object-oriented software system. In this paper, it is determined by improving the concept of a well-known metric called *Abstractness* as proposed in Martin (2003). Martin et al. defined the abstractness metric as the ratio between abstract types and the total number of types in a package. This metric helps us measure the rigidity of a component. If a component is having higher abstractness value, then it possesses lower rigidity (i.e. high flexibility of use) and vice-versa. An abstracted component is easier to extend and change. From the interface-based component point of view, the abstractness metric is redefined as follows in equation (5) and it is modified as per the author's recommendation in Hamza et al. (2013).

$$Abstractness(A) = \frac{T_{Abstract}}{T_{Abstract} + T_{Concrete}} \tag{5}$$

Here, $T_{Abstract}$ is the total number of object-oriented dependencies relations through the provides interface defined by the component. $T_{Concrete}$ is the total number of dependencies relations that are not made through the provides interface of the identified component. The value of this metric varies in the range [0...1]. Here, a value of zero indicates a completely concrete component (with no dependency through provides interface) and a value of one indicates a perfect abstract interface. To answer the research question RQ3, we constituted a team that manually checks the semantics of the transformed Java Bean component. The team consists of 15 developers who are expert in developing software using Java. Table 3 provides detail about the constituted team. The team is randomly chosen and is divided into three groups, namely research scholars, software developers, and postgraduate students. Moreover, the team is informed about the transformation steps involved in our proposed approach. The obtained Java Bean components are presented to these groups before and after the transformation. They are asked to test the functionality of these components. To make the experiment fair and unbiased the team members of different groups are kept unaware of the decisions of the persons belonging to other groups.

5 Results & Interpretations

This section of the paper gives detailed results obtained after performing the formulated experiment and analyze them. The answer to the formulated different research question is also provided. Finally, the proposed approach is evaluated with two rival approaches namely (Shatnawi et al. 2017; Rathee and Chhabra 2018a) and its comparison results are presented.

Table 3 Details of the testing team involved in the experiment

Group ID	Group domain	# Size	Experience in development
1	Research Scholar	5	5-7 Years
2	Software Developers	5	8-10 Years
3	Postgraduate Students	5	2-4 Years

Table 4 Identified reusable software components statistics

S.No.	Software name	TNC	ACS	APIS	Average afferent coupling
1	CheckStyle	262	3.42	9.12	5.87
2	JFreeChart	158	3.98	7.67	6.21
3	Log4j	60	3.65	5.32	3.67
4	Ant	292	4.23	22.43	8.23
5	android.view	132	3.73	13.65	4.54
6	android.app	95	3.80	10.78	4.12
7	java.util	287	2.95	20.33	10.23

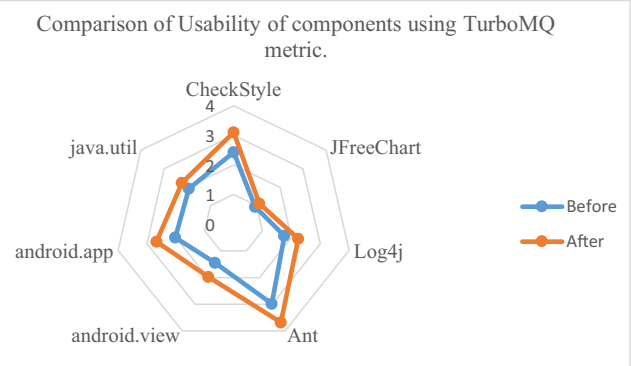

Fig. 6 Comparison of usability of different components measured in terms of TurboMQ metric

5.1 Identified Reusable Software Components

Table 4 presents the result of the clustering algorithm for different software system under study. It specifies the total count of the identified reusable components (TNC) as different clusters, the average size of the component (ACS) in terms of total software elements belonging to it, the average size of provides interface (APIS) measured in terms of the total number of methods in it. Finally, the last column in the table specifies the average value of the *Afferent Coupling*. The Afferent coupling denotes the incoming dependencies to a component and denotes how many software elements belonging to other components depends on the given component (Alshara et al. 2015). The TNC varies from 60 to 292. Moreover, as we are grouping elements in the pair of 2, 3, and 4, therefore, the ACS varies from 2.95 to 4.23. The APIS of a component varies from 5.32 to 20.33. Finally, the afferent coupling varies from 3.67 to 10.23.

5.2 Answers to Different Research Questions

RQ1. Are the identified components reusable? The reusability factor for the identified reusable components is measured in terms of the *TurboMQ* metric value. Table 5 gives the metric values obtained for different studied target software. In the table, the TurboMQ metric value is compared in two scenarios namely (1) *Before* and (2) *After,* implementing the proposed approach. The Before scenario refers to the architecture of the underlying system prior to the implementation of the proposed approach. It refers to the exact package structure of the system. Here, it is to be noted that the original package and sub-packages are considered as independent components in the original architecture. This is assumed because there does not exist a list in the literature that specifies all the components. Hence, we considered the original architecture for the evaluation purpose. The After scenario similarly refers to the architecture of the system after the proposed approach has been applied. It refers to the mutated structure of the system after clustering different elements into components. For both the scenarios the metric value is recorded. The last column in the table specifies the percentage change in the metric value for two different system architecture cases (Before and After). The values in this column vary from minimum 16.15 to maximum of 35.86. Here, higher values after clustering (component identification) indicate that now the clusters, hence components, are more balanced in terms of cohesion and coupling parameters.

Table 5 TurboMQ metric values for different software systems

S.No.	Software name	TurboMQ metric value		% Change
		Before component identification	After component identification	
1	CheckStyle	2.43	3.10	27.57
2	JFreeChart	0.94	1.12	18.02
3	Log4j	1.76	2.24	27.27
4	Ant	2.98	3.68	23.48
5	android.view	1.45	1.97	35.86
6	android.app	2.02	2.67	32.17
7	java.util	1.92	2.23	16.15

Table 6 Abstractness metric score for different software systems

S.No.	Software name	Abstractness metric value		Improvement factor
		Before component identification	After component identification	
1	CheckStyle	0.12	0.88	7.33
2	JFreeChart	0.22	0.90	4.09
3	Log4j	0.19	0.84	4.42
4	Ant	0.18	0.86	4.78
5	android.view	0.16	0.92	5.75
6	android.app	0.28	0.83	2.96
7	java.util	0.34	0.92	2.71

The plot in Fig. 6 shows the comparison between the qualities of software in two situations measured using the TurboMQ metric. The two situations are namely the original software (Before) and the system after identification of the underlying component (After). From the plot, it is clearly visible that the system, which is obtained after applying the proposed approach, has higher quality as compared to the original system. The improved quality is an indicator of the fact that the proposed approach is beneficial for project managers. This is because of improved information about the structural and semantic features used in the proposed approach. These details are dynamic and all such information can be stored with timestamps in an information system, which can be utilized by project managers during the maintenance of this software as well as the development of new similar software.

RQ2. Are the identified interfaces of the component useful and follow the component encapsulation principle? The identified provides interface of a component plays an important role in its composition with other components. Table 6 depicts the *Abstractness* metric value for different studied systems before and after transformation. The last column of the table also specifies the improvement factor that denotes the times by which the abstractness is increased in the underlying system after applying the proposed approach. The values in this column vary between 2.71 and 5.76. On average, the abstractness values change from 0.21 to 0.88 with an improvement factor of 4.19. This increase in the abstractness of the component is a good sign of its underlying composition quality as the dependency relations among different components is handled through the provides interface. Further, the obtained abstractness values are closer to 1 but not 1, it means still there are some kind of dependencies among components that are not handled through the identified provides interface of the component. On manual inspection, it is determined that there are some dependency relations such as events and exception conditions that are not captured in the two relations namely FUPR and SR.

Figure 7 shows the stacked plot of abstractness metric score for various considered software. From the plot, it is clearly visible that the software system after component identification is having much higher abstractness metric score. This indicates that the abstractness of the underlying system increased remarkably once our proposed approach is applied to it. From a component point of view, the abstractness property is mandatory because it indicates that all interaction (while integrating different components together) occurs only through the interface of the component.

RQ3. Is the proposed modeling semantically correct? The semantic/ functional consistency of a transformed system is checked manually by an expert team. The expert team is presented with three software systems of intermediate

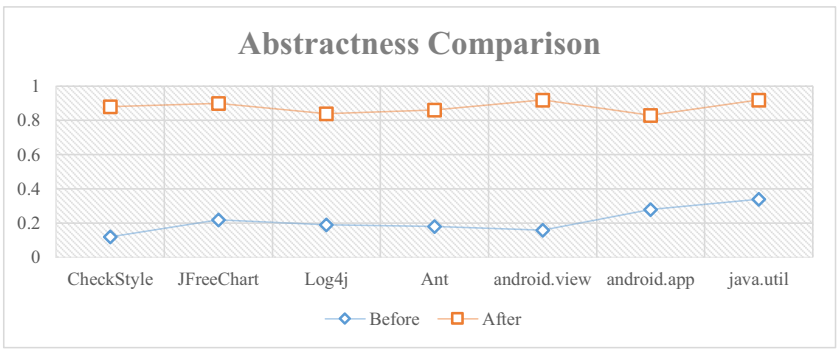

Fig. 7 Comparison of Abstractness of different software systems

Table 7 Statistics of the manual checking by expert team

S.No.	Software name	G1 Group		G2 Group		G3 Group		Average score
		U	FC	U	FC	U	FC	
1	Log4j	2	2	1	1	1	1	1.33
2	android.view	2	1	1	2	3	2	1.83
3	android.app	2	2	1	1	2	2	1.67
Average semantic correctness score								1.61

size, namely *Log4j, android.view,* and *android.app*. These systems are selected in order to make the manual analysis feasible. These systems are randomly distributed among different groups. Two versions of these systems are presented to the team, namely the original unchanged software along with its underlying source code and its corresponding transformed source-code. Each of the group checks these two versions of the system and checks the semantic correctness based on two parameters viz Understandability (U) and Functional Correctness (FC). Based on their experimentation, each of the group assigns a score in the range of [1… 5] to these evaluation parameters with the 1 assigned as the Best and 5 as the worst score. Here, the understandability score for software is dependent on the ease with which various group members cope up and understand the modification to the underlying source code. Similarly, the functional correctness measures the difference in functionalities of the system as observed by the team members in the original and transformed system. Table 7 mentions the results of the manual examination by the expert team.

Based on the statistics presented in Table 7, it can be concluded that the average semantic correctness score obtained from the expert team is 1.61. This high score confirms the fact that the proposed transformation model actually preserves the semantics of the software system.

Comparison with the Rival Approaches Here, we present a comparison result of our proposed approach with a closer rival approach. As to the best of the author's knowledge, there are no direct related works that identify the set of reusable components and together model these as Java Beans. Therefore, this paper considers evaluating the proposed approach in two steps:-

1. First, this paper analyzes the proposed approach against the Shatnawi et al. approach in Shatnawi et al. (2017). The considered rival approach converts an object-oriented API into its corresponding component API that possesses layered architecture. Their approach is based on FUP mining (Our approach is also based on FUP information mined at the element level).
2. Secondly, we compared the proposed approach against the approach proposed by Rathee et al. in (2018a). This considered rival approach aims at doing remodularization using a newly proposed dependency scheme and the multi-objective NSGA-III algorithm. This comparison helps us evaluate our used dependency criteria (based on FUP and semantic relations) in the proposed approach.

In this paragraph, we discuss the comparison results with the approach proposed by Shatnawi et al. in (2017). Table 7 shows the comparison results. Here, the proposed approach of this paper is applied to the considered systems and the identified components are compared with the results of the rival approach. From the results, it is clearly understandable that the approach of this paper is capable of identifying more coarse-grained components (Table 8).

Further, more differences between these two approaches are presented as follows:

1. Our proposed approach identifies, provides interface of a component as a set of method names that designate the beginning of the independent call chains in call-graph of the component. However, the rival approach identifies the interface of a component as the set of all the methods present in the classes belonging to its provides interfaces. From the component point of view, all the methods of a class are not directly accessed outside and hence do not constitute the optimal provides interface.
2. The rival approach does not transform the identified components to any standards of the component standards. This limits the usability of their proposed approach as their identified components cannot be directly composed using a third party software builder

Table 8 Result showing comparison with the considered rival approach in Shatnawi et al. (2017)

S.No.	Software name	# of components identified	
		Our proposed approach	Rival approach
1	java.util	287	147
2	android.view	132	43
3	android.app	95	55

Table 9 Software system quality comparison

S.No.	Software name	TurboMQ metric value		% Change
		Rival approach proposed in Rathee and Chhabra (2018a)	Proposed approach of this paper	
1	CheckStyle	2.73	3.10	13.55
2	JFreeChart	1.04	1.12	7.69
3	Log4j	1.96	2.24	14.29
4	Ant	3.28	3.68	12.20
5	android.view	1.65	1.97	18.18
6	android.app	2.32	2.67	15.09
7	java.util	2.08	2.23	7.21

tool. On the other hand, our proposed approach transforms the identified components into a reusable Java Bean standard, making them directly usable to the third party component builder tools.

3. The component mining process in the considered rival approach is highly relying on the choice of third-party client applications. This initial choice can highly deviate the obtained results. However, our proposed approach is free of such limitations.

In this paragraph of the paper, we further discuss the comparison results with another rival approach as proposed in Rathee and Chhabra (2018a). Here, first, we identified three types of dependencies viz Structural, Conceptual, and Evolutionary that exist between different software elements associated with different software systems as considered in Table 1. Secondly, we performed remodularization using the multi-objective NSGA-III algorithm having the same configuration as used by Rathee and Chhabra (2018a). Thirdly, the proposed approach in this paper is applied to the systems considered in Table 1. Finally, both the approaches are again compared using the TurboMQ quality metric. Table 9 presents the comparison results.

Figure 8 shows the plot of the overall improvement in software quality (measured in terms of TurboMQ score) for the two approaches. From the plot in Fig. 8 and the results shown in the Table 9, it is can be clearly observed that the approach of this paper significantly outperforms the rival approach by 7.21% - 18.18% in terms of the TurboMQ quality metric used. The used metric indicates an optimum balance between cohesion and coupling quality parameters of the underlying systems. Therefore, it is clear that the used dependency criteria in this paper (based on FUP and semantic relations) is capable of reflecting accurate dependency among different software elements. This accuracy is important from the reusable component identification point of view. Because this dependency is the basis of the clustering that finally identifies the set of components by analyzing the underlying source-code of the system.

RQ4. Is the proposed dependency measure (based on FUP and Semantic relations) capable of correctly identifying individual Java Bean components from a pool of multiple clusters of Beans? In order to check the feasibility of the used dependency scheme in the proposed approach, we considered a pool of ready-made Java Beans designed by third-party developers. The considered pool consists of seven Java Beans. These considered systems are readily available to the researchers and are directly usable during the software

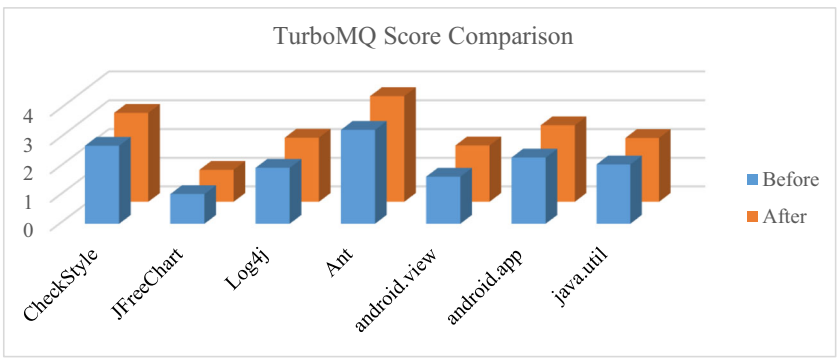

Fig. 8 Comparison of the effect of dependency criteria in terms of software quality

Table 10 Details of the considered Java Bean components

S.No.	Java Bean component name	Size (# Classes)	Description
1.	Alarm Clock Bean	5	A non-graphical bean that fires off an event after a specified delay.
2.	Arrow Bean	4	A graphical left-arrow/right-arrow bean.
3.	Font Chooser Bean	2	A graphical font-chooser bean. This bean makes use of the *NumberFieldBean* bean.
4.	Font Dialog Bean	4	A graphical font-chooser bean that pops up the font selector in a separate dialog. This bean makes use of the *FontSelectorBean* bean.
5.	Font Selector Bean	2	A graphical font-chooser bean that displays the current font and provides OK/Cancel buttons. This bean makes use of the *FontChooserBean* bean.
6.	Number Field Bean	2	A graphical numeric TextField bean with roll buttons. This bean makes use of the *ArrowBean* bean.
7.	Progress Bar Bean	2	A graphical progress-display bean.
Average		3	

development by integrating them with the software system under development.[1] Table 10 shows the details about these software systems. Here, it is to be noted that the average size of the component is three and it confirms to our considered clustering approach that focused on grouping 2, 3, and/or 4 software elements together.

The considered Java Beans (Table 10) are first randomly mixed together (by considering all of the underlying contents of a component). In the second step, again the individual components are obtained by applying our proposed approach. Finally, the obtained clustering is compared with the actual association (the original belongingness of the software element with a Java Bean) of different software elements with different considered Java Beans. Here, the comparison is evaluated using the leading information retrieval metrics, namely *Precision (P), Recall (R),* and *F-Measure* (Baeza-Yates and Ribeiro-Neto 1999). The computation of these different metrics is based on the occurrence of true positives, false positives and false negatives during the comparison of obtained clustering results and the actual Java Bean structure. Table 11 shows the results of the experimentation in terms of precision, recall, and f-measure score. The experimental results show that the average obtained precision, recall, and the f-measure score is >= 93%. This higher obtained average score indicates the capability of our proposed approach in correctly detecting the dependency relations present in a software system.

[1] https://www.javaworld.com/article/2077032/building-a-bevy-of-beans–create-reusable-javabeans-components.html

Further, as part of the investigation, we investigated the accuracy of the proposed approach in identifying the provides interfaces of the component. The provides interface of a component is identified using the proposed approach of this paper. The provides interface is identified as a set of method names. Later, the identified interface is matched by comparing it with the methods identified by the experts who developed the considered Java Beans as mentioned in Table 10. Figure 9 shows the generated call-graph for the *Alarm Bean* using the proposed approach. This figure shows different methods as circle and call chain using the arrows for the *AlarmBean.java* software element belonging to the *Alarm Bean*. From this graph the provides interface identified consist of two method names, namely *start ()* and *stop ()*. These methods confirm to the provides interface as decided by the Bean developers. These are

Table 11 Precision, recall, and F-measure values for different Java Beans

S.No.	Java Bean name	Precision	Recall	F-Measure
1.	Alarm Clock Bean	.96	.98	.97
2.	Arrow Bean	.92	.91	.92
3.	Font Chooser Bean	.88	.92	.90
4.	Font Dialog Bean	.94	.95	.95
5.	Font Selector Bean	.92	.95	.93
6.	Number Field Bean	.97	.96	.97
7.	Progress Bar Bean	.95	.94	.95
Average		.93	.94	.94

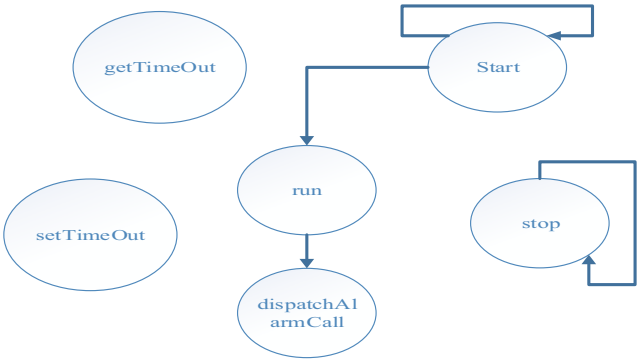

Fig. 9 Generated Call Graph for *Alarm Bean*

depicted in the *AlarmBeanBeanInfo.java* file. This confirms the feasibility of our proposed approach in identifying the interfaces of the component.

6 Threats to Validity

The findings and conclusions in this paper are subject to the following threats:

1. As the proposed approach is based on clustering of underlying software elements. This cluster identification is exponentially related to the size of the system under study and hence is an NP-Hard problem. To cope up with this we are using metaheuristic search-based algorithm in this paper. This choice may affect the results as any metaheuristic algorithm generally returns near-optimal solutions. This constitutes the internal threats to the validity of the proposed approach.
2. The validity of the semantics of the transformed software system is directly dependent on the choice of the expert team. This constitutes the external threat to the validity of the proposed approach. However, in this paper utmost care is taken while deciding team members in order to minimize the identified threat.

7 Conclusion & Future Work

In this paper, we propose an accurate and efficient reusable component identification approach from the underlying singular object-oriented legacy software's source-code. The components from the source-code of an OO software are obtained as different clusters containing different software systems of the system. The clusters contain elements in the group of 2/ 3/ or 4. The optimized clustering is performed using a modified discrete PSO algorithm. The provides interface of a component is identified by obtaining the call-graph of the component and identifying the independent call chains in this graph. The requires interface of a component is the set of methods of other components provides interfaces which are used by this component. After identifying the interfaces of the component, the obtained component is transformed into the reusable *Java Bean* component model standard specifications. The component thus obtained can directly be integrated with other components in an application being developed. The proposed approach is evaluated on seven open-source object-oriented Java software systems. All these systems are evaluated based on three criteria, namely (1) TurboMQ modularization metric, (2) the Martin's Abstractness metric, and (3) Human expertise for judging the semantics of the transformed system.

This paper offers an innovative reusable software component identification and transformation approach. The theoretical contributions of the proposed approach include the proposal of a new combined dependency scheme, a modified discrete PSO clustering algorithm, which can be useful for the research community to solve similar other information based engineering problems. The proposed approach has its own practical applications in the IT industry to develop automated tools for extracting reusable components, transforming them into Java Beans and developing an in-house repository for the IT-company. The possible future directions include the following investigations:

1. Investigation of the dependencies of the system such that the currently missing exception and event-handling dependencies are also captured in the provides interface of the component.
2. Creation of a visualization framework that helps the developers and researchers in the easy understanding of the proposed component identification approach.
3. Extension of the proposed approach to make it adaptable for other object-oriented software languages such as C++, Objective-C, and Ruby, etc.
4. Evaluation of the proposed approach in the component-based architecture recovery process.

References

Allier, S., Sadou, S., Sahraoui, H., Fleurquin, R. (2011). From object-oriented applications to component-oriented applications via component-oriented architecture. In *2011 Ninth working IEEE/IFIP conference on software architecture* (pp. 214–223). https://doi.org/10.1109/WICSA.2011.35.

Alshara, Z., Seriai, A.D., Tibermacine, C., Bouziane, H.L., Dony, C., Shatnawi, A. (2015). Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. *SIGPLAN Not.*, *51*(3), 55–64.

Baeza-Yates, R.A., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. Boston: Addison-Wesley Longman Publishing Co., Inc.

Bavota, G., Carnevale, F., De Lucia, A., Di Penta, M., Oliveto, R. (2012). Putting the developer in-the-loop: an interactive ga for software re-modularization. In Fraser, G., & Teixeira de Souza,

J. (Eds.) *Search based software engineering* (pp. 75–89). Berlin: Springer.

Birkmeier, D., & Overhage, S. (2009). On component identification approaches – classification, state of the art, and comparison. In Lewis, G.A., Poernomo, I., Hofmeister, C. (Eds.) *Component-based software engineering* (pp. 1–18). Berlin: Springer.

Chardigny, S., Seriai, A., Oussalah, M., Tamzalit, D. (2008). Extraction of component-based architecture from object-oriented systems. In *Seventh working IEEE/IFIP conference on software architecture (WICSA 2008)* (pp. 285–288). https://doi.org/10.1109/WICSA.2008.44.

von Detten, M., Platenius, M.C., Becker, S. (2014). Reengineering component-based software systems with archimetrix. *Software &, Systems Modeling*, *13*(4), 1239–1268.

Ducasse, S., & Pollet, D. (2009). Software architecture reconstruction: a process-oriented taxonomy. *IEEE Transactions on Software Engineering*, *35*(4), 573–591. https://doi.org/10.1109/TSE.2009.19.

Emmerich, W., & Kaveh, N. (2001). Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. *SIGSOFT Softw. Eng. Notes*, *26*(5), 311–312. https://doi.org/10.1145/503271.503259.

Garriga, M., Renzis, A.D., Lizarralde, I., Flores, A., Mateos, C., Cechich, A., Zunino, A. (2018). A structural-semantic web service selection approach to improve retrievability of web services. *Information Systems Frontiers*, *20*(6), 1319–1344.

Gong, C., Chen, H., He, W., Zhang, Z. (2017). Improved multi-objective clustering algorithm using particle swarm optimization. In *Plos one*.

Hamdouni, A.E., Seriai, A., Huchard, M. (2010). Component-based architecture recovery from object oriented systems via relational concept analysis. In *Proceedings of the 7th international conference on concept lattices and their applications, Sevilla, Spain, October 19-21, 2010, pp. 259–270*.

Hamza, S., Sadou, S., Fleurquin, R. (2013). Measuring qualities for osgi component-based applications. In *2013 13Th international conference on quality software* (pp. 25–34). https://doi.org/10.1109/QSIC.2013.42.

Heisel, M., Santen, T., Souquières, J. (2002). On the specification of components - the JavaBeans example. Intern report A02-R-025 —– heisel02a. https://hal.inria.fr/inria-00107634. Rapport interne.

Ilk, N., Zhao, J.L., Goes, P., Hofmann, P. (2011). Semantic enrichment process: an approach to software component reuse in modernizing enterprise systems. *Information Systems Frontiers*, *13*(3), 359–370.

Kebir, S., Seriai, A.D., Chardigny, S., Chaoui, A. (2012). Quality-centric approach for software component identification from object-oriented code. In *2012 Joint working IEEE/IFIP conference on software architecture and european conference on software architecture* (pp. 181–190).

Kessel, M., & Atkinson, C. (2016). Ranking software components for reuse based on non-functional properties. *Information Systems Frontiers*, *18*(5), 825–853. https://doi.org/10.1007/s10796-016-9685-3.

Lorenz, D.H., & Vlissides, J. (2001). Designing components versus objects: a transformational approach. In *Proceedings of the 23rd international conference on software engineering. ICSE 2001* (pp. 253–263). https://doi.org/10.1109/ICSE.2001.919099.

Martin, R.C. (2003). *Agile software development: principles, patterns, and practices*. Upper Saddle River: Prentice Hall PTR.

Mitchell, B.S. (2003). A heuristic approach to solving the software clustering problem. In *International conference on software maintenance, 2003. ICSM 2003. Proceedings* (pp. 285–288).

Mu, L., Sugumaran, V., Wang, F. (2019). A hybrid genetic algorithm for software architecture re-modularization. *Information Systems Frontiers*. https://doi.org/10.1007/s10796-019-09906-0.

Nebro, A.J., Durillo, J.J., Coello Coello, C.A., Luna, F., Alba, E. (2008). A study of convergence speed in multi-objective metaheuristics. In Rudolph, G., Jansen, T., Beume, N., Lucas, S., Poloni, C. (Eds.) *Parallel problem solving from nature – PPSN X* (pp. 763–772). Berlin: Springer.

Praditwong, K., Harman, M., Yao, X. (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, *37*(2), 264–282.

Prajapati, A., & Chhabra, J.K. (2017). Improving modular structure of software system using structural and lexical dependency. *Information &, Software Technology*, *82*, 96–120.

Prajapati, A., & Chhabra, J.K. (2018). FP-ABC: fuzzy-pareto dominance driven artificial bee colony algorithm for many-objective software module clustering. *Computer Languages, Systems &, Structures*, *51*, 1–21.

Prajapati, A., & Chhabra, J.K. (2018). Many-objective artificial bee colony algorithm for large-scale software module clustering problem. *Soft Comput.*, *22*(19), 6341–6361.

Prajapati, A., & Chhabra, J.K. (2018). A particle swarm optimization-based heuristic for software module clustering problem. *Arabian Journal for Science and Engineering*, *43*(12), 7083–7094. https://doi.org/10.1007/s13369-017-2989-x.

Prajapati, A., & Chhabra, J.K. (2019). Information-theoretic remodularization of object-oriented software systems. *Information Systems Frontiers*. https://doi.org/10.1007/s10796-019-09897-y.

Prajapati, A., & Chhabra, J.K. (2019). Madhs: many-objective discrete harmony search to improve existing package design. *Computational Intelligence*, *35*(1), 98–123.

Rathee, A., & Chhabra, J.K.Singh, D., Raman, B., Luhach, A.K., Lingras, P. (Eds.) (2017). *Software remodularization by estimating structural and conceptual relations among classes and using hierarchical clustering*. Singapore: Springer.

Rathee, A., & Chhabra, J.K. (2018a). Clustering for software remodularization by using structural, conceptual and evolutionary. *Journal of Universal Computer Science*, *24*(12), 1731–1757.

Rathee, A., & Chhabra, J.K. (2018b). Improving cohesion of a software system by performing usage pattern based clustering. *Procedia Computer Science*, *125*, 740–746. The 6th International Conference on Smart Computing and Communications.

Rathee, A., & Chhabra, J.K. (2019). A multi-objective search based approach to identify reusable software components. *Journal of Computer Languages*, *52*, 26–43.

Rathee, A., & Chhabra, J.K. (2019). Reusability in multimedia softwares using structural and lexical dependencie. *Multimedia Tools and Applications*.

Seriai, A., Sadou, S., Sahraoui, H.A. (2014). *Enactment of components extracted from an object-oriented application*, (pp. 234–249). Cham: Springer.

Shan, S.M., Deng, G.S., He, Y.H. (2006). Data clustering using hybridization of clustering based on grid and density with pso. In *2006 IEEE International conference on service operations and logistics, and informatics* (pp. 868–872). https://doi.org/10.1109/SOLI.2006.328970.

Shatnawi, A., & Seriai, A.D. (2013). Mining reusable software components from object-oriented source code of a set of similar software. In *2013 IEEE 14th international conference on information reuse integration (IRI)* (pp. 193–200). https://doi.org/10.1109/IRI.2013.6642472.

Shatnawi, A., Seriai, A.D., Sahraoui, H., Alshara, Z. (2017). Reverse engineering reusable software components from object-oriented apis. *Journal of Systems and Software*, *131*, 442–460.

Shatnawi, A., Shatnawi, H., Saied, M.A., Alshara, Z., Sahraoui, H.A., Seriai, A. (2018). Identifying components from object-oriented apis based on dynamic analysis. CoRR arXiv:1803.06235.

Sierra, M.R., & Coello Coello, C.A. (2005). Improving pso-based multi-objective optimization using crowding, mutation and dominance. In Coello Coello, C.A., Hernández Aguirre, A., Zitzler, E. (Eds.) *Evolutionary multi-criterion optimization* (pp. 505–519). Berlin: Springer.

Song, M., Jung, H., Yang, Y. (2002). The analysis technique for extraction of ejb component from legacy system. In *In proc. 6th IASTED international conference on software engineering and applications* (pp. 241–244).

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J. (2010). The qualitas corpus: a curated collection of java code for empirical studies. In *2010 Asia pacific software engineering conference* (pp. 336–345). https://doi.org/10.1109/APSEC.2010.46.

Tydrichova, M. (2017). Analysis of various multi-objective optimization evolutionary algorithms for monte carlo treatment planning system. http://cds.cern.ch/record/2276909.

Washizaki, H., & Fukazawa, Y. (2005). A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, *56*(1), 99–116. New Software Composition Concepts.

Yacoub, S., Ammar, H., Mili, A. (1999). Characterizing a software component. In *Proceedings of the 2nd workshop on component-based software engineering, in conjunction with ICSE'99*.

Amit Rathee is currently a fellow researcher at National Institute of Technology, Kurukshetra- 136119, Haryana, India. He received his Master degree from Kurukshetra University, Kurukshetra, Haryana, India in Computer Science and Engineering in year 2010. He completed his graduation from Kurukshetra University, Kurukshetra in Computer Science and Engineering in year 2006. His research interest includes software engineering, human-computer interaction, soft computing, algorithm and data structure, machine learning.

Dr Jitender Kumar Chhabra, Professor, Dept of Computer Engg and HOD (VLSI and Embedded Systems), has been always topper throughout his career. He did his B Tech as well as M Tech from N.I.T. Kurukshetra as Gold Medalist (N.I.T.K. is one of the most reputed Institute in India & ranked immediately after IITs). He has more than 26 years of teaching & research experience. He is author of three books from McGraw Hill publisher including the one Schaum Series International book.

He has filed 5 patents, obtained one copyright grant, and is Principal Investigator of a Research Project funded by DRDO, Govt of India. He is Reviewer of many journals like IEEE Transactions, ACM Transactions, Elsevier, Springer, Wiley, Taylor & Francis, Inderscience etc. He has published more than 130 papers in reputed International and National Journals and conferences. He has guided many scholars for their PhD and one of his guided PhD thesis has been selected as resource Material by ACM in the area of software engineering. He has also worked in collaboration with multinational IT companies and his guided teams have performed excellently in various International competitions. He is recipient of Best Teacher Award, Sir Isaac Newton Scientific Award, Best Educator Award, All India Badminton Champion, Best Project Award, Best Presentation Award, Outstanding Educational Achievement, International Professional & Scientist, 21st Century Award for Achievement, Cambridge Blue Book Achievement, Outstanding Intellectual of 21stCentury etc. His areas of interest are Software Engineering, Software Quality, Soft Computing & Object-Oriented Systems.

Affiliations

Amit Rathee[1] · **Jitender Kumar Chhabra**[1]

Jitender Kumar Chhabra
jitenderchhabra@gmail.com

[1] Department of Computer Engineering, National Institute of Technology, Kurukshetra, 136119, India