Software Remodularization by Estimating Structural and Conceptual Relations Among Classes and Using Hierarchical Clustering

Amit Rathee(✉) and Jitender Kumar Chhabra

Department of Computer Engineering, National Institute of Technology, Kurukshetra 136119, India
amitl983_rathee@rediffmail.com,
jitenderchhabra@gmail.com

Abstract. In this paper, we have presented a technique of software remodularization by estimating conceptual similarity among software elements (Classes). The proposed technique makes use of both structural and semantic coupling measurements together to get much more accurate coupling measures. In particular, the proposed approach makes use of lexical information extracted from six main parts of the source code of a class, namely comments, class names, attribute names, method signatures, parameter names and method source code statements zone. Simultaneously, it also makes use of counting of other class's member functions used by a given class as a structural coupling measure among classes. Structural coupling among software elements (classes) are measured using information-flow based coupling metric (ICP) and conceptual coupling is measured by tokenizing source code and calculating Cosine Similarity. Clustering is performed by performing Hierarchical Agglomerate Clustering (HAC). The proposed technique is tested on three standard open source Java software's. The obtained results encourage remodularization by showing higher accuracy against the corresponding software gold standard.

1 Introduction

In object-oriented systems, classes act as a primary composing element, which group the associated data and the functionality together. Also, when a software is developed, these classes are grouped together into various modules called packages. These modules are expected to possess higher cohesion and low coupling properties. But this structure starts deteriorating with time because of continuous maintenance activities performed on it for changing requirements. During maintenance new classes are added and older ones may be modified or deleted. Hence, resulting in architectural erosion. This may also happen due to not following the object-oriented guidelines properly because of the pressure to deliver the software early in the market. All these activities degrades class cohesion and coupling. So, to reduce maintenance activity, a new and efficient approach is needed, which can detect this degradation and ultimately, restructure the software system to minimize the maintenance.

In this paper, dependency among classes are measured and highly dependent/coupled classes are identified. Such classes need remodularization to maintain

D. Singh et al. (Eds.): ICAICR 2017, CCIS 712, pp. 94–106, 2017.
DOI: 10.1007/978-981-10-5780-9_9

structural quality. In this paper, we propose a new approach to measure coupling among classes that is used further for restructuring of the software architecture by performing hierarchical agglomerate clustering (HAC). The proposed approach focuses on reducing coupling among classes by doing restructuring. In proposed approach, coupling among classes is based on both structural and conceptual coupling analysis. The conceptual coupling is analysed based on semantic relations among classes. Two classes possess higher structural coupling if they are using each other's functionality by calling each other's member functions and accessing member variables. So, the structural coupling is measured by counting the in-flow and out-flow of information, called Information-Flow-based Coupling (ICP), among classes of the given software. To measure semantic coupling and hence conceptual dependency, we first extract tokens from six main zones of a class namely: comments, class names, attribute names, method signatures, parameter names and method source code statements zone and then performed Latent Semantic Indexing (LSI) to reduce dimensionality. The extracted tokens from these zones are further normalized using Porter Stemmer's algorithm [31] and TF-IDF value of each token is measured. Then, cosine similarity among classes of the software is computed as semantic dependency. Finally, we combine both measurements to get the final coupling among classes. Since, the extracted tokens strongly represent the corresponding domain of the software, so, they reflect the conceptual coupling among classes.

The rest of this paper is organized as follows: Sect. 2 gives the literature survey, Sect. 3 gives the details of the proposed work, Sect. 4 shows the experiment and results obtained and finally Sect. 5 gives conclusions and future work remarks.

2 Literature Survey

Lots of ideas have already been proposed in literature aiming at supporting software system restructuring and remodularization in the world of software engineering. Since the 80's, many authors have researched and proposed techniques in term of metrics, frameworks, etc. to increase overall quality of systems.

Many graph based techniques had been proposed by Cimitile and Visaggio [1], Shaw et al. [15], and Antoniol et al. [3] which aims at identifying strongly connected subgraphs in the Call Graph representation of the system under study. van Deursen and Kuipers [5] and Tonella [4] also proposed the re-modularization of software system by using concept analysis technique, which extracts the group of objects that share some common attributes among them. Mancoridis et al. [16] proposed algorithms to automatically recover the underlying software architecture from source code. They have used a search-based clustering approach to recover the software modular structure. Mitchell and Mancoridis [6] used the same approach and developed Bunch tool for automatic system decomposition into a more cohesive modular structure. Harman et al. [7] and Seng et al. [8] have also proposed system decomposition techniques which makes use of search-based approaches and various soft computing techniques. Abdeen et al. [9] have proposed and implemented a heuristic search-based technique for optimizing inter-package coupling. Abdellatief et al. [17] provided various component-based system dependency metrics which are based on graph methods. Qiu et al. [18]

also proposed methods to measure software similarity based on the structural properties derived from UML diagrams. Savic et al. [19] proposed a SNEIPL tool for the automatic extraction of dependencies using software network and enriched Concrete Syntax Tree (eCST) representation of the source code.

Component classification and similarity measurement is also done based only on lexical information. Srinivas et al. [20] gives a similarity metric based on the lexical properties and representation of the component in vector form. Another set of conceptual coupling matrices is proposed by Poshyvank based on the semantic information present in identifiers and comments from the source code. Kagdi et al. [30] proposed another set of metrics for measuring the conceptual similarity between the method-method, method-class and class-class.

The combined use of both conceptual and structural measures have also been studied in the literature by many researchers [2, 10–13, 21–23, 27–29]. Bavota et al. [13, 14] used graph theory to identify various extract package refactoring opportunities. These refactoring operations tries to remove promiscuous package problem in software system, i.e. a package is assigned more responsibilities that should have been divided in separate packages. Maletic and Marcus [10] have proposed an approach which makes use of both semantic and structural data to propose refactoring decisions that are helpful in reducing maintenance efforts and other reengineering activities of software systems. Kuhn et al. [11] extended the work by Maletic and Marcus by giving a visualization support for the clusters and their semantic relationships. Adritsos and Tzerpos [24] presented LIMBO, a hierarchical algorithm for software clustering. The clustering algorithm proposed counts both structural and non-structural attributes to reduce the maintenance activity of a software system by decomposing it into clusters. Corazza et al. [21–23] presented a clustering based approach to divide OO system into subsystems, by considering only lexical information extracted from the source code and then using a partitioning algorithm to build more cohesive subsystems. Belle et al. [25] proposed an approach based on both linguistic and structural data to recover the underlying architecture of OO Systems. Measuring structural and conceptual coupling based on the structural and conceptual information and using it to remodularize software system using HAC clustering is the main characteristics of our approach. The combined approach to estimate similarity measurement and remodularization using structural and semantic coupling in literature have also been proposed similar to our approach, but, no one in literature have proposed a combined approach which aims at dividing the source code in different zones to measure semantic coupling and simultaneously used in-flow and out-flow information counting to measure structural coupling together. The approach of dividing the source code into various zones and extracting tokens for similarity measurement is more accurate because most of the conceptual information's introduced by programmer are covered by these zones. The in-flow & out-flow measurement of information also gives a direct measurement of actual structural coupling present in source code.

3 Proposed Approach

The proposed approach focus on the measurement of conceptual coupling between different Java classes belonging to a software system and then use that information in performing remodularization. The proposed approach is depicted in Fig. 1. It is based on measuring both Structural and Semantic Coupling among classes. As today's software are built by following proper standards and guidelines by programmers. So, lots of valuable semantic information is already present in the source code in the form of member-variable names, class-names, function-names, parameter-names etc. and they are strongly related with the concept of the class i.e. the idea implemented by the class. This idea motivates us to extract those words from source code in the form of tokens and use them for dependency calculation. Similarly, classes are also structurally coupled due to the usage of each other's member function. This motivates us to measure Structural Coupling among classes also. Further, these two dependencies information, Semantic & Structural Coupling, are combined together after normalizing Structural Coupling and used for clustering using a hierarchical agglomerative clustering algorithm.

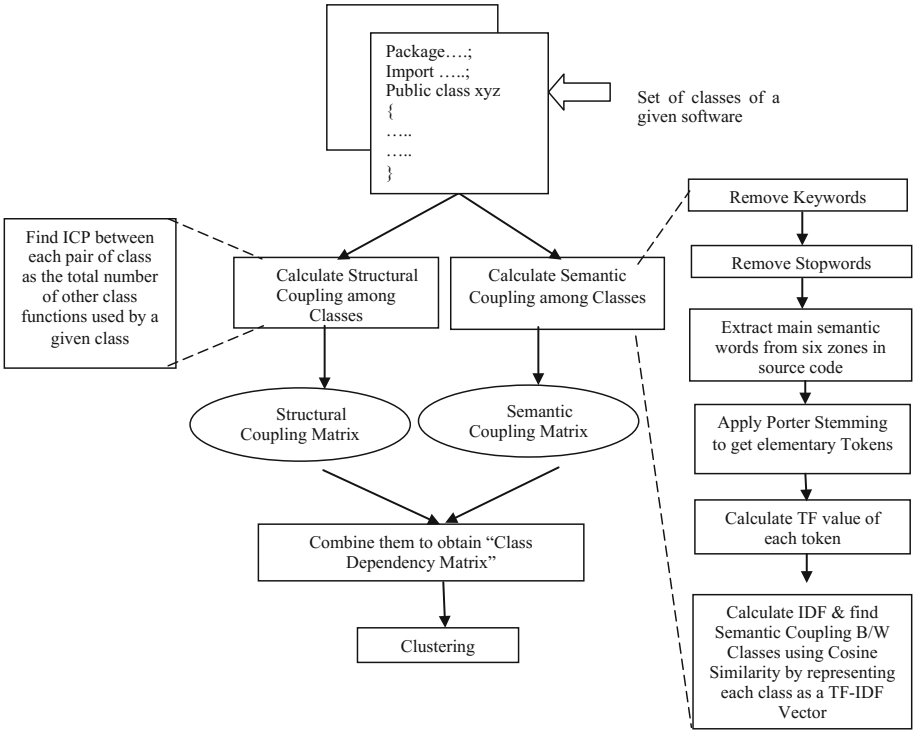

Fig. 1. Proposed methodology to measure conceptual coupling among classes

The source code can be considered as a systematic collection of lexical information, which are present in various parts. We have found that most of these information's are mostly present in following six parts/zones of source code:

1. Comments Zone.
2. Class Name Zone.
3. Attribute Names Zone.
4. Method Signatures Zone.
5. Parameter Names Zone.
6. Method Source Code Statements Zone.

The Comments Zone includes two sections: Javadoc & Comments provided by the programmers. This zone provides documentation related to class and used mainly for maintenance activities. The Class Name Zone includes class definition section present in source code. The Attribute Name Zone includes all member variables defined in a class definition. The Method Signature Zone includes all class member function's prototype statements. It also includes user defined data types used as parameters. The Parameter Names Zone section includes all the variables used in a class member function's definition as parameter names. And finally the Method Source Code Statements Zone consists of all the statements present inside the class member function's definition body. The proposed work is divided into following subsections:

3.1 Semantic Similarity

In the first step, we find the similarity among all classes of a software by utilizing lexical information extracted from different zones of classes and the cosine similarity technique in which each class Ci is represented in the form of lexical vector Vi consisting of token's TF-IDF value and then similarity among two classes Ci and Cj is calculated through corresponding lexical vectors Vi and Vj respectively by using the following formula:

$$\text{CoSim}\ (\text{C}_\text{i}, \text{C}_\text{j}) = \frac{V_i \cdot V_j}{\|V_i\| * \|V_j\|} \tag{1}$$

Finally, we get a square matrix called **Semantic Coupling Matrix (SeCM)** of size n where n is the total number of classes in a software. The SeCM is a symmetric matrix because semantic coupling among any two class C_i and C_j is unidirectional in nature. The overall process is represented as an algorithm consisting of the following steps:

// Finding Semantic Coupling Matrix

1. Remove all keywords.
2. Remove Stopwords.
3. Extract tokens.
4. Normalize tokens by applying Porter Stemmer Algorithm.
5. Divide tokens into various zones.
6. Calculate the frequency of each token.

7. Represent each class as a vector in TF-IDF form.
8. Calculate Cosine Similarity between each pair of class to obtain **Semantic Coupling Matrix (SeCM).**

The working of the above algorithm is explained with the help of two java classes as shown in Fig. 2. The keyword removal step consists of filtering out all standard words of java like int, public, class and predefined class names such as serializable, hashtable etc. As the Javadoc and comments consist of English language sentences, so it is necessary to remove all common words called stopwords to get an accurate similarity measure. In our example, the stopwords are a, to, of and in. After this step, the next step is to extract all major words as lexical information and apply java's Camel NameCase Style division to get more rooted elements. E.g. schoolYear is divided into two tokens: school and year respectively. In the next step, the porter stemming algorithm is applied to normalize ambiguity due to plurals, verbs etc. e.g. handle, handling and handler will represent the same token rather than three separate tokens. This gives better control over semantic similarity measurement.

```
/* A public class to handle SemesterWise details of a student
*/
    public class Semester implements Serializable {
       // a list of member variables
       private int schoolYear;
       private int semNum;
       private String semCode;
       private Hashtable studEnrolled = new
      Hashtable();
       private Hashtable subjOpen = new Hashtable();
       // a list of public member functions
       public Semester() { …..}
       public Semester(int sy, int sem) {…….}
       public void enrollStudent (Student thisStudent) {…..}
       public int getSchoolYear() {…..}
       public String getSemCode() {…..}
       public int getSem() {…..}
       public void openSubject(Subject thisSubject) {…..}
       public void setSem(int newSem) {…..}
    }
```

Semester.java

```
/* A class to handle Students personal details in a university */
public class PersonRecord implements Serializable {
   // a list of member variables
   private String firstName;
   private String lastName;
   private String middleName;
   private String middleInitial;
   private static int count;
   // a list of member functions
   public Person(){int a;……}
   public Person(String fn, String ln, String mn){…..}
   public String getFirstName(){……}
   public String getLastName(){……}
   public static int getCount(){….}
   public void setFirstName(String str){……}
   public void setLastName(String str){…..}
   public static void setCount(){…..}
}
```

Person.java

Fig. 2. Sample java files

Finally, all tokens are extracted and their frequency count is recorded separately for each class. A complete list of these tokens is as shown below in Table 1. Finally, this information is represented as vector and similarity is calculated using Eq. (1).

3.2 Structural Similarity

In the second step, we find the similarity among the classes by utilizing the structural properties. We consider Information-Flow-based Coupling (ICP) as a structural similarity measure. In this we measure total number of member functions of other classes called/ used by the given class as the structural coupling, i.e. ICP measures the total

Table 1. Token list extracted from four zones of two sample files

Zone	Semester.java		Person.java	
	Token name	Frequency	Token name	Frequency
Comments	handle	1	handle	1
	semester	1	student	1
	wise	1	person	1
	detail	1	detail	1
	student	1	university	1
	list	2	list	2
	member	2	member	2
	variable	1	variable	1
	function	1	function	1
Class name	semester	1	person	1
	serialize	1	record	1
	–	–	serialize	1
Attribute name	school	1	first	1
	year	1	name	3
	sem	2	last	1
	num	1	middle	2
	code	1	initial	1
	stud	1	count	1
	enrol	1	–	–
	subj	1	–	–
	open	1	–	–
Method signature	semester	2	person	2
	sem	3	first	2
	enrol	1	name	4
	student	2	last	2
	school	1	count	2
	year	1	–	–
	code	1	–	–
	open	1	–	–
	Subject	2	–	–
	–	–	str	2
Parameter	Sy	1	fn	1
	sem	1	ln	1
	–	–	mn	1
Source code statement	–	–	a	1

amount of inflow and outflow of information from a class. The overall process is represented in the form of an algorithm consisting of the following steps:

// Finding Structural Coupling Matrix

1. Find ICP coupling matrix between each pair of class, which measures the total no. of method invocations.
2. Represents the result in **Structural Coupling Matrix (StCM)** and normalize the entries to the range [0…1].

The ICP is calculated for each pair of classes and results in the generation of Structural Coupling Matrix of order nxn, where n is the total number of classes under study. The ICP between two classes C_i and C_j is calculated using following formula:

$$ICP(C_i, C_j) = \sum_{k=1}^{k=\left|call(C_i,C_j)\right|} calls(C_i, C_j) \tag{2}$$

Here, calls (C_i, C_j) denote the total number of functions $\left(m_1, m_2 \ldots\ldots, m_j\right) \varepsilon C_j$ called by class Ci. Similarly, because, StCM is not symmetric due to the directional nature of structural coupling among classes. So, it is necessary to make it symmetric. To do this, ICP (C_j, C_i) is calculated as mentioned below just to possess symmetry relation:

$$ICP(C_j, C_i) = max\left(ICP(C_i, C_j), ICP(C_j, C_i)\right) \tag{3}$$

3.3 Combining Structural and Semantic Coupling Matrix

In this step, we combine the two similarity matrix SeCM & StCM together. Here, first of all, before addition of entries, the entries of the StCM must be normalized because it contains 0 as minimum and its maximum value is unbounded. Whereas, our semantic coupling matrix's entries are in the range [0…1]. To combine the two metrics in a single similarity measure we will first normalize the structural matrix values in the range [0…1]. To do this, the below mentioned formula as used by Bavota et al. [13] is used to normalize each entry $C_{i,\,j}$ of the structural coupling matrix (StCM):

$$C_{i,j} = \frac{\left(C_{i,j} - min(StCM)\right)}{max(StCM) - min(StCM)} \tag{4}$$

Here, min (StCM) & max (StCM) are the minimum and maximum values in Structural Coupling Matrix (StCM) respectively.

After normalization, matrices StCM & SeCM are added together, resulting in final coupling matrix for the system as Class Dependency Matrix (CDM). It is a symmetric nxn matrix, where n is the total no. of classes in the system, and it represents the overall coupling (structural + semantic) among classes of a given software.

3.4 Clustering Using Hierarchical Agglomerate Clustering

The last step in software remodularization is the clustering process. It aims to recover the underlying software architecture by regrouping classes based on dependency

relations. Here, the clustering is performed using an HAC. Agglomerate hierarchical algorithms starts with each entity (class) assigned to individual cluster, which are in turn combined together until we get a single large cluster containing all the entities. Various types of hierarchical clustering algorithms have been discussed in [26]. We have used Complete Linkage Hierarchical Algorithm. In computing distance d of newly formed cluster from existing clusters with distance d1 & d2 as d = max (d1, d2). The Complete Linkage is chosen because it gives more cohesive and uniform clusters [26]. It results in a tree with each node representing a cluster and the height of the tree represents the dissimilarity among clusters. For performing software remodularization, this tree needs to be cut at appropriate height to have a group of clusters rather than one. This process is depicted in Fig. 3 below. Here, the system is divided into two clusters by cutting the tree at cut point 1. The first cluster contains E1–E3 and the second E4, E5 entities. Similarly, we can divide the system into four clusters by cutting at cut point 2.

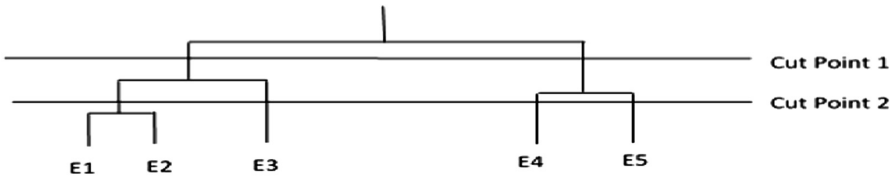

Fig. 3. A hierarchical tree divided into two clusters by cutting tree at cut point

Given a set of N entities (classes) which needs to be regrouped together, and an N*N distance (or similarity) matrix, the HAC is done by following algorithmic steps:

1. Assign each entity Ei having feature vector Ef1, Ef2,……, EfN to single cluster, so that initially there are total N clusters, each having just one item.
2. Find the minimum distinct (most similar) pair of clusters and merge them into a single cluster, so that now there are total N-1 clusters.
3. Again Calculate distances (similarities) between the newly created cluster and each of the old ones using Complete Linkage method.
4. Repeat steps 2 and 3 until all entities have been joined into a single cluster of size N.

4 Experimentation and Results

To validate the proposed methodology, we conduct our experiment on three freely available open source java software's whose details is given in Table 2. The table defines the software name, its version, total number of classes, thousand lines of code (KLOC) count and total number of packages in software. The semantic information extraction and dependency calculation is done using a tool specially designed by the authors. Whereas, the structural dependencies are calculated using Stan4J Tool.

Table 2. Description of software system under study

System	Version	#Classes	KLOC	#Packages
Junit	4.5	27	2.74	4
Easy mock	2.4	63	3.06	3
Servlet-API	2.3	40	2.84	4

The proposed approach has been evaluated using expert criteria approach. The collected clustering results P_C is compared against clustering obtained from expert opinions P_G as gold standards. The comparison has been performed by finding out Precision & Recall Metrics and ultimately calculating the f-measure for each software. Table 3 shows the f-measure values of each software under study. The results indicates positive measurement of cohesion based on structural and conceptual relations among entities of a software system. As two entities can be in the same cluster called Intra pair or they may be in different cluster called Inter pair. So, precision and recall can be defined as:

Table 3. Precision, recall & F-measure values

System	Precision	Recall	F-Measure
Junit	0.62	0.35	0.52
Easy mock	0.91	0.40	0.62
Servlet-API	0.67	0.64	0.66

Precision: Precision is defined as the percentage of intra pairs given by the clustering method which are also intra in the expert partition. The value of precision of the constructed package P_C and original package P_G is calculated using following formula:

$$\text{Precision } P_i = \frac{|C(P_{Ci}) \cap C(P_{Gi})|}{|C(P_{Ci})|}$$

Recall: Recall is defined as the percentage of intra pairs in the expert partition which are identified by the clustering method. The value of recall from the constructed package PC and original package PG is calculated using following formula:

$$\text{Recall } P_i = \frac{|C(P_{Ci}) \cap C(P_{Gi})|}{|C(P_{Gi})|}$$

Here, $C(P_{Ci})$ & $C(P_{Gi})$ is the total number of classes in the i^{th} constructed package P_{Ci} and original package P_{Gi}. Since the above two metrics measure two different aspects regarding a software partition, so, we further evaluated F-measure metric to make balance between two results. The F-measure value of i^{th} package is obtained from the corresponding precision and recall using following formula:

$$\text{F - measure}\,\mathrm{P_i} \;=\; 2 * \frac{PrecisionP_i * Recall\,P_i}{PrecisionP_i + Recall\,P_i}$$

5 Conclusion and Future Work

In this paper, we proposed a technique to measure the conceptual similarity among classes of an OO Software which is ultimately used for performing software remodularization using hierarchical agglomerate clustering to obtain better architecture. Based on the above experiment and results, it can be concluded that using a combination of both structural & conceptual coupling measure, gives better dependency indication rather than using them as individuals. The present work is mainly based on measuring the Cosine Similarity & ICP among classes and in future many other techniques and metrics for finding similarity can be tested and a comparison can be further done. Also, in future inter module/package dependency can be tested using the same proposed technique. The methodology can be modified and tested to suite other OO languages such as C++ etc. Furthermore, other clustering technique can be used & tested to provide appropriate grouping.

References

1. Cimitile, A., Visaggio, G.: Software salvaging and the call dominance tree. J. Syst. Softw. **28** (2), 117–127 (1995)
2. Marcus, A., Poshyvanyk, D., Ferenc, R.: Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans. Softw. Eng. **34**(2), 287–300 (2008)
3. Antoniol, G., Di Penta, M., Casazza, G., Merlo, E.: A method to re-organize legacy systems via concept analysis. In: Proceedings of 9th International Workshop on Program Comprehension, Toronto, Canada, pp. 281–292 (2001)
4. Tonella, P.: Concept analysis for module restructuring. IEEE Trans. Softw. Eng. **27**(4), 351–363 (2001)
5. van Deursen, A., Kuipers, T.: Identifying objects using cluster and concept analysis. In: Proceedings of 21st International Conference on Software Engineering, Los Angeles, California, USA, pp. 246–255 (1999)
6. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Trans. Softw. Eng. **32**(3), 193–208 (2006)
7. Harman, M., Hierons, R.M., Proctor, M.: A new representation and crossover operator for search-based optimization of software modularization. In: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA (2002)
8. Seng, O., Bauer, M., Biehl, M., Pache, G.: Search-based improvement of subsystem decompositions. In: Proceedings of the Genetic and Evolutionary Computation Conference, Washington, Columbia, USA, pp. 1045–1051 (2005)
9. Abdeen, H., Ducasse, S., Sahraoui, H.A., Alloui, I.: Automatic package coupling and cycle minimization. In: Proceedings of the 16th Working Conference on Reverse Engineering, Lille, France, pp. 103–112 (2009)

10. Maletic, J., Marcus, A.: Supporting program comprehension using semantic and structural information. In: Proceedings of 23rd International Conference on Software Engineering. Toronto, Ontario, Canada, pp. 103–112 (2001)
11. Kuhn, A., Ducasse, S., Gîrba, T.: Semantic clustering: identifying topics in source code. Inf. Soft. Technol. **49**(3), 230–243 (2007)
12. Scanniello, G., Risi, M., Tortora, G.: Architecture recovery using latent semantic indexing and k-means: an empirical evaluation. In: Proceedings of International Conference on Software Engineering and Formal Methods, pp. 103–112 (2010)
13. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Software re-modularization based on structural and semantic metrics. In: Proceedings of International Working Conference on Reverse Engineering, pp. 195–204. IEEE Computer Society (2010)
14. Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., De Lucia, A.: Methodbook: recommending move method refactorings via relational topic models. IEEE Trans. Softw. Eng. **40**(7), 671–694 (2014)
15. Shaw, S.C., Goldstein, M., Munro, M., Burd, E.: Moral dominance relations for program comprehension. IEEE Trans. Softw. Eng. **29**(9), 851–863 (2003)
16. Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.-F., Gansner, E.R.: Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of 6th International Workshop on Program Comprehension, Ischia, Italy. IEEE CS Press (1998)
17. Abdellatief, M., Sultan, A.B.M., Ghani, A., Jabar, M.A.: Component-based software system dependency metrics based on component information flow measurements. In: Sixth International Conference on Software Engineering Advances, IARIA (2011)
18. Qiu, D.H., Li, H., Sun, J.L.: Measuring software similarity based on structure and property of class diagram. In: 6th International Conference on Advanced Computational Intelligence (ICACI). IEEE (2013)
19. Savic, M., Rakic, G., Budimac, Z., Ivanovic, M.: A language-independent approach to the extraction of dependencies between source code entities. IST **56**, 1268–1288 (2014). Elsevier
20. Srinivas, C., Radhakrishna, V., Rao, C.V.G.: Software component clustering and classification using noval similarity measure. In: 8th International Conference Interdisciplinarity in Engineering (INTER-ENG), Romania (2014)
21. Corazza, A., Di Martino, S., Scanniello, G.: A probabilistic based approach towards software system clustering. In: Proceedings of European Conference on Software Maintenance and Reengineering, pp. 89–98. IEEE Computer Society (2010)
22. Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: Investigating the use of lexical information for software system clustering. In: Proceedings of European Conference on Software Maintenance and Reengineering, pp. 35–44. IEEE Computer Society (2011)
23. Corazza, A., Martino, S., Maggio, V., Scanniello, G.: Weighing lexical information for software clustering in the context of architecture recovery. Empir. Softw. Eng. **21**, 72–103 (2016)
24. Andritsos, P., Tzerpos, V.: Information-theoretic software clustering. IEEE Trans. Softw. Eng. **31**(2), 150–165 (2005)
25. Belle, A.B., Boussaidi, G.E., Kpodjedo, S.: Combining lexical and structural information to reconstruct software layers. Inf. Softw. Technol. **74**, 1–16 (2016)
26. Maqbool, O., Babri, A.H.: Hierarchical clustering for software architecture recovery. IEEE Trans. Softw. Eng. **33**(11), 759–780 (2007)
27. Prajapati, A., Chhabra, J.K.: Improving modular structure of software system using structural and lexical dependency. Inf. Softw. Technol. **82**, 96–120 (2017). (Elsevier, SCI)
28. Parashar, A., Chhabra, J.K.: An approach for clustering class coupling metrics to mine object oriented software components. Int. Arab J. Inf. Technol. **13**(3), 239–248 (2016). (SCI)

29. Prajapati, A., Chhabra, J.K.: Preserving core components of object-oriented packages while maintaining structural quality. Procedia Comput. Sci. **46**, 833–840 (2015). (Elsevier)
30. Kagdi, H., Gethers, M., Poshyvanyk, D.: Integrating conceptual and logical couplings for change impact analysis in software. Empir. Softw. Eng. **18**, 933–969 (2013)
31. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)