STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

A PARTITIONAL CLUSTERING ALGORITHM FOR IMPROVING THE STRUCTURE OF OBJECT-ORIENTED SOFTWARE SYSTEMS

ISTVAN GERGELY CZIBULA AND GABRIELA CZIBULA

Abstract. In this paper we are focusing on the problem of program restructuring, an important process in software evolution. We aim at introducing a partitional clustering algorithm that can be used for improving software systems design. The proposed algorithm improve several clustering algorithms previously developed in order to recondition the class structure of a software system. We experimentally validate our approach and we provide a comparison with existing similar approaches.

1. Introduction

The software systems, during their life cycle, are faced with new requirements. These new requirements imply updates in the software systems structure, that have to be done quickly, due to tight schedules which appear in real life software development process. That is why continuous restructuring of the code is needed, otherwise the system becomes difficult to understand and change, and therefore it is often costly to maintain. Without continuous restructurings of the code, the structure of the system becomes deteriorated. Thus, *program restructuring* is an important process in software evolution.

A continuous improvement of the software systems structure can be made using *refactoring*, that assures a clean and easy to maintain software structure.

We have previously introduced in [6] a clustering approach for identifying refactorings in order to improve the structure of software systems. For this purpose, a clustering algorithm named *kRED* was introduced. To our knowledge, there is no approach in the literature that uses clustering in order to improve the class structure of a software system, excepting the approach introduced in

Received by the editors: November 10, 2008.

2000 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement –*Restructuring, reverse engineering, and reengineering*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

Key words and phrases. Software design, Refactoring, Clustering.

[6]. The existing clustering approaches handle methods decomposition [26] or system decomposition into subsystems [13].

We have improved the approach from [6] by developing several clustering algorithms that can be used to identify the refactorings needed in order to recondition the class structure of an object-oriented software system [3–5, 20, 21].

The aim of this paper is to introduce a partitional clustering algorithm which takes an existing software and reassembles it, in order to obtain a better design, suggesting the needed refactorings. The clustering algorithm proposed in this paper improves all the algorithms that we have already developed.

The rest of the paper is structured as follows. Section 2 presents the main aspects related to the clustering approach (*CARD*) for determining refactorings [6] that we intend to improve in this paper. A new partitional clustering algorithm for determining refactorings is introduced in Section 3. Section 4 presents experimental evaluations of the proposed approach: the open source case study JHotDraw [10] and a real software system. Some conclusions and further work are given in Section 6.

2. Background

We have previously introduced in [6] a clustering approach (*CARD*) in order to find adequate refactorings to improve the structure of software systems. *CARD* approach consists of the following steps:

(1) The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system.
(2) The set of entities extracted at the previous step are re-grouped in clusters (classes) using a clustering algorithm (*PARED* in our approach). The goal of this step is to obtain an improved structure of the existing software system.
(3) The newly obtained software structure is compared with the original software structure in order to provide a list of refactorings which transform the original structure into an improved one.

3. A Partitional Clustering Algorithm for Refactorings Determination (PARED)

In this section we introduce a new partitional clustering algorithm (*PARED*) (Partitional Clustering Algorithm for Refactorings Determination). *PARED* algorithm can be used in the **Grouping** step of *CARD* in order to identify a

partition of a software system S that corresponds to an improved structure of it.

In our clustering approach, the objects to be clustered are the entities from the software system S, i.e., $\mathcal{O} = \{s_1, s_2, \ldots, s_n\}$. Our focus is to group similar entities from S in order to obtain high cohesive groups (clusters).

We will adapt the generic cohesion measure introduced in [22] that is connected with the theory of similarity and dissimilarity. In our view, this cohesion measure is the most appropriate to our goal. We will consider the dissimilarity degree between any two entities from the software system S. Consequently, we will consider the distance $d(s_i, s_j)$ between two entities s_i and s_j as expressed in Equation (1).

$$d(s_i, s_j) = \begin{cases} 1 - \frac{|p(s_i) \cap p(s_j)|}{|p(s_i) \cup p(s_j)|} & if \quad p(s_i) \cap p(s_j) \neq \emptyset \\ \infty & otherwise \end{cases}, \tag{1}$$

where, for a given entity $e \in S$, $p(e)$ defines a set of relevant properties of e, expressed as follows. If $e \in Attr(S)$ (e is an attribute) then $p(e)$ consists of: the attribute itself, the application class where the attribute is defined, and all the methods from $Meth(S)$ that access e. If $e \in Meth(S)$ (e is a method) then $p(e)$ consists of: the method itself, the application class where the method is defined, all the attributes from $Attr(S)$ accessed by the method, all the methods from S used by e, and all methods from S that overwrite method e. If $e \in Class(S)$ (e is an application class) then $p(e)$ consists of: the application class itself, all the attributes and the methods defined in the class, all interfaces implemented by class e and all classes extended by class e.

Our distance, as defined in Equation (1), highlights the concept of cohesion, i.e., entities with low distances are cohesive, whereas entities with higher distances are less cohesive.

Based on the definition of distance d (Equation (1)) it can be easily proved that d is a semi-metric function, so a *k-medoids* based approach can be applied.

In order to avoid the two main disadvantages of the traditional *k-medoids* algorithm, *PARED* algorithm uses a heuristic for choosing the number of medoids (clusters) and the initial medoids. This heuristic is particular to our problem and it will provide a good enough choice of the initial medoids.

After selecting the initial medoids, *PARED* behaves like the classical *k-medoids* algorithm.

The main idea of *PARED*'s heuristic for choosing the initial medoids and the number p of clusters (medoids) is the following:

(i) The initial number p of clusters is n (the number of entities from the software system) and the intial number nr of medoids is 0.

(ii) The entity chosen as the first medoid is the most "distant" entity from the set of all entities (the entity that maximizes the sum of distances from all other entities). The number nr of medoids becomes 1.

(iii) In order to choose the next medoid we reason as follows. For each remaining entity (that was not chosen as medoid), we compute the minimum distance ($dmin$) from the entity and the already chosen medoids. The next medoid is chosen as the entity e that maximizes $dmin$ and this distance is greater than a positive given threshold ($distMin$), and nr is increased. If such an entity does not exist, it means that e is very close to all the medoids and should not be chosen as a new medoid (from the software system structure point of view this means that e should belong to the same application class with an existing medoid). In this case, the number p of medoids will be decreased.

(iv) The step (iii) will be repeatedly performed, until the number nr of chosen medoids is equal to p.

We have to notice that step (iii) described above assures, from the software system design point of view, that near entities (with respect to the given threshold $distMin$) will be merged in a single application class (cluster), instead of being distributed in different application classes.

We mention that at steps (ii) and (iii) the choice could be a non-deterministic one. In the current version of *PARED* algorithm, if such a non-deterministic case exists, the first selection is made. Future improvements of *PARED* algorithm will deal with these kind of situations.

The main idea of the *PARED* algorithm that we apply in order to group entities from a software system is the following:

(i) The initial number p of clusters and the initial medoids are determined by the heuristic described above.

(ii) The clusters are recalculated, i.e., each object is assigned to the closest medoid.

(iii) Recalculate the medoid i of each cluster k based on the following idea: if h is an object from k such that $\sum_{j \in k}(d(j,h) - d(j,i))$ is negative, then h becomes the new medoid of cluster k.

(iv) Steps (ii)-(iii) are repeatedly performed until there is no change in the partition $\mathcal{K}$.

We mention that *PARED* algorithm provides a partition of a software system S, partition that represents a new structure of the software system. Regarding to *PARED* algorithm, we have to notice the following:

- If, at a given moment, a cluster becomes empty, this means that the number of clusters will be decreased.

- Because the initial medoids are selected based on the heuristic described above, the dependence of the algorithm on the initial medoids is eliminated.
- We have chosen the value 1 for the threshold $distMin$, because distances greater than 1 are obtained only for unrelated entities (Equation (1)).

The main refactorings identified by *PARED* algorithm are *Move Method*, *Move Attribute*, *Inline Class*, *Extract Class* [9]. We have currently implemented the above enumerated refactorings, but *PARED* algorithm can also identify other refactorings, like: *Pull Up Attribute*, *Pull Down Attribute*, *Pull Up Method*, *Pull Down Method*, *Collapse Class Hierarchy*. Future improvements will deal with these situations, also.

4. Experimental Evaluation

In order to experimentally validate our clustering approach, we will consider two evaluations, which are described below.

Our first evaluation is the open source software JHotDraw, version 5.1 [10]. It is a Java GUI framework for technical and structured graphics, developed by Erich Gamma and Thomas Eggenschwiler, as a design exercise for using design patterns. It consists of **173** classes, **1375** methods and **475** attributes. The reason for choosing JHotDraw as a case study is that it is well-known as a good example for the use of design patterns and as a good design.

Our focus is to test the accuracy of our approach on JHotDraw, i.e., how accurate are the results obtained after applying *PARED* algorithm in comparison with the current design of JHotDraw. As JHotDraw has a good class structure, *PARED* algorithm should generate a nearly identical class structure.

After applying *PARED* algorithm, we have obtained a partition in which there are no misplaced methods and attributes, meaning that the class structure discovered by *PARED* is identical to the actual structure of JHotDraw.

Our second evaluation is a DICOM (*Digital Imaging and Communications in Medicine*) [8] and HL7 (*Health Level 7*) [11] compliant PACS (*Picture Archiving and Communications System*) system, facilitating medical images management, offering access to radiological images, and making the diagnosis process easier. We have applied *PARED* algorithm on one of the subsystems from this application, subsystem containing **1015** classes, **8639** methods and **4457** attributes.

After applying *PARED* algorithm, a total of 84 refactorings have been suggested: 7 *Move Attribute* refactorings, 75 *Move Method* refactorings, and

2 *Inline Class* refactoring. From the refactorings obtained by *PARED* algorithm, 55% were accepted by the developers of the considered software system.

Analyzing the obtained results, we have concluded that a large number of miss-identified refactorings are due to technical issues: the use of Java anonymous inner classes, introspection, the use of dynamic proxies. These kind of technical aspects frequently appear in projects developed in JAVA. In order to correctly deal with these aspects, we have to improve only the data collection step from our approach, without modifying *PARED* algorithm. Another cause of miss-identified refactorings is due to the fact that the *distance* (Equation (1)) used for discriminating entities in the clustering process take into account only two aspects of a good design: *low coupling* and *high cohesion*. It would be also important to consider other principles related to an improved design, like: *Single Responsibility Principle*, *Open-Closed Principle*, *Interface Segregation Principle*, *Common Closure Principle* [7], etc. Future improvements of our approach will deal with these aspects, also.

5. Related Work

In this section we present some approaches existing in the literature in the fields of *software clustering* and *refactoring*. We provide, for similar approaches, a comparison with our approach.

There is a lot of work in the literature in the field of *software clustering*.

One of the most active researches in the area of software clustering were made by Schwanke. The author addressed the problem of automatic clustering by introducing the *shared neighbors* technique [17], technique that was added to the low-coupling and high-cohesion heuristics in order to capture patterns that appear commonly in software systems. In [18], a partition of a software system is refined by identifying components that belong to the wrong subsystem, and by placing them in the correct one. The paper describes a program that attempts to reverse engineer software in order to better provide software modularity. Schwanke assumes that procedures referencing the same name must share design information on the named item, and are thus "design coupled". He uses this concept as a clustering metric to identify procedures that should be placed in the same module. Even if the approaches from [17] and [18] were not tested on large software systems, they were promising.

Mancoridis et al. introduce in [14] a collection of algorithms that facilitate the automatic recovery of the modular structure of a software system from its source code. Clustering is treated as an optimization problem and genetic algorithms are used in order to avoid the local optima problem of *hill-climbing* algorithms. The authors accomplish the software modularization process by

constructing a *module dependency graph* and by maximizing an objective function based on inter- and intra-connectivity between the software components. A clustering tool for the recovery and the maintenance of software system structures, named *Bunch*, is developed. In [15], some extensions of *Bunch* are presented, allowing user-directed clustering and incremental software structure maintenance.

A variety of software clustering approaches have been presented in the literature. Each of these approaches looks at the software clustering problem from a different angle, by either trying to compute a measure of similarity between software objects [17]; deducing clusters from file and procedure names [1]; utilizing the connectivity between software objects [2, 12, 16]; or looking at the problem at hand as an optimization problem [14]. Another approach for software clustering was presented in [1] by Anquetil and Lethbridge. The authors use common patterns in file names as a clustering criterion. The authors' experiments produced promising results, but their approach relies on the developers' consistency with the naming of their resources.

The paper [24] also approaches the problem of software clustering, by defining a metric that can be used in evaluating the similarity of two different decompositions of a software system. The proposed metric calculates a distance between two partitions of the same set of software resources. For calculating the distance, the minimum number of operations (such as moving a resource from one cluster to another, joining two clusters etc.) one needs to perform in order to transform one partition to the other is computed. Tzerpos and Holt introduce in [25] a software clustering algorithm in order to discover clusters that follow patterns that are commonly observed in decompositions of large software systems that were prepared manually by their architects.

All of these techniques seem to be successful on a number of examples. However, not only is there no approach that is widely recognized as superior, but it is also hard to compare the effectiveness of different approaches. As presented above, the approaches in the field of *software clustering* deal with the software decomposition problem. Even if similarities exist with refactorings extraction, a comparison is hard to make due to the different granularity of the decompositions (modules vs. classes, methods, fields).

There were various approaches in the literature in the field of *refactoring*, also. But, only very limited support exists in the literature for automatic refactorings detection.

For most existing approaches, the obtained results for relevant case studies are not available. There are given only short examples indicating the obtained refactorings. That is why we have selected for comparison only two techniques mentioned below.

The paper [23] describes a software vizualization tool which offers support to the developers in judging which refactoring to apply. We have applied *PARED* algorithm on the example given in [23] an the *Move Method* refactoring suggested by the authors was obtained.

A search based approach for refactoring software systems structure is proposed in [19]. The authors use an evolutionary algorithm for identifying refactorings that improve the system structure.

The advantages of our approach in comparison with the approach presented in [19] are illustrated bellow. Our technique is deterministic, in comparison with the approach from [19]. The evolutionary algorithm from [19] is executed **10** times, in order to judge how stable are the results, while *PARED* algorithm from our approach is executed just **once**. The technique from [19] reports **11** misplaced methods, while in our approach there are **no** misplaced methods. The overall running time for the technique from [19] is about **300** minutes (30 minutes for one run), while *PARED* algorithm in our approach provide the results in about **1.2** minutes. We mention that the execution was made on similar computers. Because the results are provided in a reasonable time, our approach can be used for assisting developers in their daily work for improving software systems.

6. Conclusions and Future Work

We have presented in this paper a new partitional clustering algorithm (*PARED*) that can be used for improving software systems design. We have demonstrated the potential of our algorithm by applying it to the open source case study JHotDraw and to a real software system, and we have also presented the advantages of our approach in comparison with existing approaches. Based on the feedback provided by the developers of a real software system we have identified some potential improvements of our approach.

Further work will be done in the following directions: to use other search based approaches in order to determine refactorings that improve the design of a software system; to improve the *distance* function used in the clustering process; to apply *PARED* algorithm on other large software systems; to apply our approach in order to transform non object-oriented software into object-oriented systems.

ACKNOWLEDGEMENT

This work was supported by the research project TD No. 411/2008, sponsored by the Romanian National University Research Council (CNCSIS).

References

[1] Nicolas Anquetil and Timothy Lethbridge, *Extracting concepts from file names; a new file clustering criterion*, 20th International Conf. Software Engineering, 1998, pp. 84–93.

[2] Song C. Choi and Walt Scacchi, *Extracting and restructuring the design of large systems*, IEEE Softw. **7** (1990), no. 1, 66–71.

[3] I.G. Czibula and G. Serban, *A hierarchical clustering algorithm for software systems design improvement*, KEPT 2007: Proceedings of the first International Conference on Knowledge Engineering: Principles and Techniques, August 2007June 6, pp. 316–323.

[4] I. G. Czibula and G. Serban, *Hierarchical clustering for software systems restructuring*, INFOCOMP Journal of Computer Science, Brasil **6** (2007), no. 4, 43–51.

[5] I.G. Czibula and G. Serban, *Software systems design improvement using hierarchical clustering*, SERP'07: Proceedings of SERP'07, 2007, pp. 229–235.

[6] Istvan G. Czibula and Gabriela Serban, *Improving Systems Design Using a Clustering Approach*, International Journal of Computer Science and Network Security (IJCSNS) **6** (2006), no. 12, 40–49.

[7] Tom DeMarco, *Structured analysis and system specification* (2002), 529–560.

[8] *Digital Imaging and Communications in Medicine*. http://medical.nema.org/.

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the design of existing code*, Addison-Wesley, Reading, MA, USA, 1999.

[10] E. Gamma, *JHotDraw Project*. http://sourceforge.net/projects/jhotdraw.

[11] *Health Level 7*. www.hl7.org/.

[12] David H. Hutchens and Victor R. Basili, *System structure analysis: clustering with data bindings*, IEEE Trans. Softw. Eng. **11** (1985), no. 8, 749–757.

[13] Chung-Horng Lung, *Software architecture recovery and restructuring through clustering techniques*, Isaw '98: Proceedings of the third International Workshop on Software Architecture, 1998, pp. 101–104.

[14] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, *Using automatic clustering to produce high-level system organizations of source code*, IEEE Proceedings of the 1998 int. Workshop on Program Understanding (IWPC'98), 1998, pp. 45–52.

[15] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner, *Bunch: A clustering tool for the recovery and maintenance of software system structures*, ICSM, 1999, pp. 50–59.

[16] James M. Neighbors, *Finding reusable software components in large systems*, Working Conference on Reverse Engineering, 1996, pp. 2–10.

[17] R. W. Schwanke and M. A. Platoff, *Cross references are features*, Proceedings of the 2nd International Workshop on Software Configuration Management, 1989, pp. 86–95.

[18] Robert W. Schwanke, *An intelligent tool for re-engineering software modularity*, ICSE '91: Proceedings of the 13th International Conference on software engineering, 1991, pp. 83–92.

[19] Olaf Seng, Johannes Stammel, and David Burkhart, *Search-based determination of refactorings for improving the class structure of object-oriented systems*, GECCO '06: Proceedings of the 8th annual conference on genetic and evolutionary computation, 2006, pp. 1909–1916.

[20] G. Serban and I.G. Czibula, *A new clustering approach for systems designs improvement*, SETP-07: Proceedings of the International Conference on Software Engineering Theory and Practice, December 2007 July 9, pp. 47–54.

[21] G. Serban and I. G. Czibula, *Restructuring software systems using clustering*, ISCIS 2007: Proceedings of the 22nd International Symposium on Computer and Information Sciences, September 2007 November 7, pp. 33, IEEExplore.
[22] Frank Simon, Silvio Loffler, and Claus Lewerentz, *Distance based cohesion measuring*, Proceedings of the 2nd European Software Measurement Conference (FESMA), 1999, pp. 69–83.
[23] Frank Simon, Frank Steinbruckner, and Claus Lewerentz, *Metrics based refactoring*, CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001, pp. 30–38.
[24] Vassilios Tzerpos and Richard C. Holt, *Mojo: A distance metric for software clusterings*, Working conference on reverse engineering, 1999, pp. 187–193.
[25] Vassilios Tzerpos and Richard C. Holt, *ACDC: An algorithm for comprehension-driven clustering*, Working conference on reverse engineering, 2000, pp. 258–267.
[26] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan, *Program restructuring through clustering techniques*, SSAM '04: Proceedings of the Workshop on source code analysis and manipulation, Fourth IEEE International (SCAM'04), 2004, pp. 75–84.

Department of Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu Street, Cluj-Napoca, Romania,
E-mail address: istvanc@cs.ubbcluj.ro

Department of Computer Science, Babeş-Bolyai University 1, M. Kogălniceanu Street, Cluj-Napoca, Romania,
E-mail address: gabis@cs.ubbcluj.ro