KNOWLEDGE ENGINEERING: PRINCIPLES AND TECHNIQUES
Proceedings of the International Conference on Knowledge Engineering,
Principles and Techniques, KEPT2007
Cluj-Napoca (Romania), June 6–8, 2007, pp. 316–323

A HIERARCHICAL CLUSTERING ALGORITHM FOR SOFTWARE DESIGN IMPROVEMENT

ISTVAN GERGELY CZIBULA(1) AND GABRIELA ŞERBAN(2)

Abstract. *Refactoring* is a process that helps to maintain the internal software quality, during the whole software lifecycle. The aim of this paper is to present a new hierarchical clustering algorithm that can be used for improving software systems design. *Clustering* is used in order to recondition the class structure of a software system. The proposed approach can be useful for assisting software engineers in their daily works of refactoring software systems. We evaluate our approach using the open source case study JHotDraw ([13]), providing a comparison with previous approaches.

1. Introduction

The structure of a software system has a major impact on the maintanability of the system. That is why continuous restructurings of the code are needed, otherwise the system becomes difficult to understand and change, and therefore it is often costly to maintain.

In order to keep the software structure clean and easy to maintain, most modern software development methodologies (extreme programming and other agile methodologies) use refactoring to continuously improve the system structure.

In [7], Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs". Refactoring is viewed as a way to improve the design of the code after it has been written. Software developers have to identify parts of code having a negative impact on the system's maintainability, and apply appropriate refactorings in order to remove the so called "bad-smells" ([11]).

In this paper we propose a new hierarchical clustering algorithm that would help developers to identify the appropriate refactorings. Our approach takes an existing software and reassembles it using hierarchical clustering, in order to obtain a better

2000 *Mathematics Subject Classification.* 68N99, 62H30.
Key words and phrases. Software Engineering, Refactoring, Hierarchical Clustering.

design, suggesting the needed refactorings. Applying the proposed refactorings remains the decision of the software engineer.

The rest of the paper is structured as follows. Section 2 presents the main aspect related to the problem of *clustering*. The clustering approach (*CARD*) for determining refactorings, that we have previously introduced in [1], is presented in Section 3. A new hierarchical clustering algorithm for identifying refactorings is introduced in Section 4. Section 5 provides an experimental evaluation of our approach. A comparison of the proposed approach with other similar approaches is given in Section 6. Conclusions and further work are given in Section 7.

2. Clustering

Unsupervised classification, or clustering, as it is more often referred as, is a data mining activity that aims to differentiate groups (classes or clusters) inside a given set of objects ([6]), being considered the most important *unsupervised learning* problem. The resulting subsets or groups, distinct and non-empty, are to be built so that the objects within each cluster are more closely related to one another than objects assigned to different clusters. Central to the clustering process is the notion of degree of similarity (or dissimilarity) between the objects.

Let $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ be the set of objects to be clustered. The measure used for discriminating objects can be any *metric* or *semi-metric* function $d : \mathcal{O} \times \mathcal{O} \longrightarrow \Re$. The distance expresses the dissimilarity between objects.

In this paper we are focusing only on *hierarchical clustering*, that is why an overview of the hierarchical clustering methods is presented. Hierarchical clustering methods represent a major class of clustering techniques. There are two styles of hierarchical clustering algorithms. Given a set of n objects, the agglomerative (bottom-up) methods begin with n singletons (sets with one element), merging them until a single cluster is obtained. At each step, the most similar two clusters are chosen for merging. The divisive (top-down) methods start from one cluster containing all n objects and split it until n clusters are obtained.

The agglomerative clustering algorithms that were proposed in the literature differ in the way the two most similar clusters are calculated and the linkage-metric used (single, complete or average) ([10]).

3. Background. Refactorings Determination using Clustering

In this section we describe the clustering approach (*CARD*) introduced in [1] in order to find adequate refactorings to improve the structure of software systems. Our aim is, that based on the approach from [1], to introduce a new hierarchical clustering algorithm, that is why a brief description of *CARD* is given below.

In [1], a software system S is viewed as a set $S = \{s_1, s_2, ..., s_n\}$, where $s_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class. *CARD* consists of three steps:

- **Data collection.** The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existent relationships between them.
- **Grouping.** The set of entities extracted at the previous step are re-grouped in clusters using a partitioning algorithm (*HARED* algorithm, in our approach). The goal of this step is to obtain an improved structure of the existing software system.
- **Refactorings extraction.** The newly obtained software structure is compared with the original structure in order to provide a list of refactorings which transform the original structure into an improved one.

A more detailed description of *CARD* is given in [1]. At the **Grouping** step of *CARD*, the software system S has to be re-grouped. This re-grouping is represented as a ***partition*** of S, $\mathcal{K} = \{K_1, K_2, ..., K_v\}$. In the following, we will refer to K_i as the i-th *cluster* of $\mathcal{K}$, and to an element s_i from S as an *entity*. A cluster K_i from the partition $\mathcal{K}$ represents an application class in the new structure of the software system.

4. A new Hierarchical Clustering Algorithm for Refactorings Determination - HARED

Based on the clustering approach *CARD* described in Section 3, we present in this section a new hierarchical clustering algorithm for refactoring determination (*HARED - Hierarchical Algorithm for Refactorings Determination*). This algorithm can be used in the **Grouping** step of *CARD*, in order to find an improved structure of the software system S.

HARED is based on the idea of hierarchical agglomerative clustering, but uses an heuristic for merging two clusters. We use *average link* as linkage metric, because we have obtained better results with this metric.

The heuristic used in *HARED* is that, at a given step, the most two similar clusters (the pair of clusters that have the smallest distance between them) are merged only if the distance between them is less or equal to a given threshold, $distMin$. This means that the entities from the two clusters are close enough in order to be placed in the same cluster (application class). This heuristic is particular to our approach and it will provide a good enough choice for merging two application classes.

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 [8] 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 $e \in S$, $p(e)$ represents a set of relevant properties of e, defined as:

- If e is an attribute, then $p(e)$ consists of: the attribute itself, the application class where the attribute is defined, and all methods from S that access the attribute.
- If e is a method, then $p(e)$ consists of: the method itself, the application class where the method is defined, and all attributes from S accessed by the method.
- If e is a class, then $p(e)$ consists of: the application class itself, and all attributes and methods defined in the class.

Based on the definition of distance d given in Equation (1) it can be easily proved that d is a semi-metric function. We will consider the distance $dist(k, k')$ between two clusters $k \in \mathcal{K}$ and $k' \in \mathcal{K}$ as given in Equation (2).

$$dist(k, k') = \frac{1}{|k| \cdot |k'|} \cdot \sum_{e \in k, e' \in k'} d(e, e') \tag{2}$$

The main steps of *HARED* algorithm are:

- Each entity from the software system is put in its own cluster (singleton).
- The following steps are repeated until the partition of methods remains unchanged (no more clusters can be selected for merging):
  - select the two most similar clusters from the current partition, i.e, the pair of clusters that minimize the distance from Equation (2). Let us denote by *dmin* the distance between the most similar clusters K_i and K_j;
  - if $dmin \leq distMin$ (the given threshold), then clusters K_i and K_j will be merged, otherwise the partition remains unchanged.

We give next *HARED* algorithm.

`Algorithm HARED is`
Input: `-` the software system $\mathcal{S} = \{s_1, \ldots, s_n\}, n \geq 2$,
`-` the semi-metric d between entities,
`-` $distMin > 0$ the threshold for merging the clusters.
Output: `-` the partition $\mathcal{K} = \{K_1, K_2, ..., K_p\}$, the new structure of $\mathcal{S}$.
Begin
`For i ← 1 to n do`

```
    Ki ← {si} //each entity is put in its own cluster
  endfor
  K ← {K1, ..., Kn} //the initial partition
 change ← true
 While change do //while K changes
    dmin ← dist(K1, K2) //the minimum distance between clusters
    For i* ← 1 to n-1 do //the most similar clusters are chosen
      For j* ← i* + 1 to n do
        d ← dist(Ki*, Kj*)
        If d < dmin then
           dmin ← d; i ← i*; j ← j*
        endif
      endfor
    endfor
    If dmin ≤ distMin then
      Knew ← Ki ∪ Kj; K ← (K \ {Ki, Kj}) ∪ {Knew}
     else
      change ← false //the partition remains unchanged
    endif
  endwhile
End.
```

In our approach we have chosen the value 1 for the threshold $distMin$, because distances greater than 1 are obtained only for unrelated entities (Equation (1)).

4.1. **Refactorings Extraction.** In this section we briefly discuss about the refactorings that *HARED* algorithm is able to identify.

Let us consider that S is the analyzed software system, and that $\mathcal{K} = \{K_1, K_2, \ldots, K_p\}$ is the partition provided by *HARED*, i.e., the new structure of S. The main refactorings identified by *HARED* algorithm are given below.

***Move Method* ([7]) refactoring**. It moves a method m of a class C to another class C' that uses the method most. The bad smell motivating this refactoring is that a method uses or is used by more features of another class than the class in which it is defined ([5]). This refactoring is identified by *HARED* by moving the method m in the cluster K_t corresponding to the class C'.

***Move Attribute* ([7]) refactoring**. It moves an attribute a of a class C to another class C' that uses the attribute most. The bad smell motivating this refactoring is that an attribute is used by another class more than the class in which it is defined ([5]). This refactoring is identified by *HARED* algorithm by moving the attribute a in the cluster K_t corresponding to the class C'.

***Inline Class* ([7]) refactoring**. It moves all members of a class C into another class C' and deletes the old class. The bad smell motivating this refactoring is

that a class is not doing very much ([5]). This refactoring is identified by *HARED* algorithm by decreasing the number of elements in the partition $\mathcal{K}$. Consequently, the number of application classes in the new structure of S is decreased, and classes C and C' with their corresponding entities (methods and attributes) will be merged in the same cluster K_t.

***Extract Class* ([7]) refactoring**. Creates a new class C and move some cohesive attributes and methods into the new class. The bad smell motivating this refactoring is that one class offers too much functionality that should be provided by at least two classes ([5]). This refactoring is identified by *HARED* algorithm by increasing the number of elements in the partition $\mathcal{K}$. Consequently, a new cluster appears, corresponding to a new application class in the new structure of S.

5. Experimental Evaluation

In order to validate our clustering approach, we consider as case study the open source software JHotDraw, version 5.1 ([13]). 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.

At the *Data collection* step of *CARD*, in order to extract from the system the input data for *HARED* algorithm, we use ASM 3.0 ([3]). ASM is a Java bytecode manipulation framework. We use this framework in order to extract the structure of the system (attributes, methods, classes and relationships between entities).

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 *HARED* algorithm introduced in Section 4 on JHotDraw, i.e., how accurate are the results obtained after applying *HARED* algorithm in comparison to the current design of JHotDraw. As JHotDraw has a good class structure, the *Grouping* step of *CARD* should generate a nearly identical class structure. After applying *HARED* we have obtained the following results:

(i) The algorithm obtains a new class after the re-grouping step, meaning that an *Extract Class* refactoring is suggested. The methods which are placed in the new class are: **PertFigure.handles**, **GroupFigure.handles**, **TextFigure.handles**, **StandardDrawing.handles**.

(ii) There are two misplaced attributes, **ColorEntry.fColor** and **ColorEntry.fName** which are placed in **ColorMap** class. This means that two *Move Attribute* refactorings are suggested.

(iii) There are four misplaced methods, **UngroupCommand.execute**, **FigureTransferCommand.insertFigures**, **SendToBackCommand.execute**, and **BringToFrontCommand.execute** which are placed in **StandardDrawing** class.

In our view, the refactorings identified at (i) and (ii) can be justified.

- All the methods enumerated at (i) provide similar functionality ([13]), so, in our view, these methods can be extracted in a new class in order to avoid duplicated code, applying *Extract Class* refactoring.
- **ColorMap** and **ColorEntry** are two classes defined in the same source file. **ColorMap** is an utility class which manages the default colors used in the application. **ColorEntry** is a simple class used only by **ColorMap**, that is why, in our view, **fColor** and **fName** attributes can be placed in either of the two classes.

6. Related Work

There are various approaches in the literature in the field of *refactoring*. The only approach on the topic studied in this paper, that partially gives the results obtained on a relevant case study (like JHotDraw) is [2]. The authors use an evolutionary algorithm in order to obtain a list of refactorings using JHotDraw.

The advantages of *HARED* algorithm in comparison with the approach presented in [2] are illustrated bellow:

- In the technique from [2] there are **10** misplaced methods, while in our approach there are only **4** misplaced methods.
- Our technique is deterministic, in comparison with the approach from [2]. The evolutionary algorithm from [2] is executed **10** times, in order to judge how stable are the results.
- The overall running time for the technique from [2] is about **300** minutes (30 minutes for one run), while *HARED* algorithm provide the results in about **3.5** minutes (the execution was made on similar computers).
- As the results are provided in a reasonable time, our approach can be used by developers in their daily work for improving software systems.

We cannot make a complete comparison with other refactoring approaches, because, for most of them, the obtained results for relevant case studies are not available. Most approaches (like [4], [12]) give only short examples indicating the obtained refactorings. Other techniques address particular refactorings: the one in [4] focuses on automated support only for identifying ill-structured or low cohesive functions and the technique in [12] focuses on system decomposition into subsystems.

7. Conclusions and Future Work

We have presented in this paper, based on the approach from [1], a new hierarchical clustering algorithm (*HARED*) that can be used for improving systems design. We have demonstrated the potential of our approach by applying it to the open source case study JHotDraw and we have also presented the advantages of our approach in comparison with existing approaches.

Further work can be done in the following directions:

- To apply *HARED* for other relevant case studies.
- To use other approaches for clustering, such as search based clustering ([9]), or genetic clustering.
- To develop a tool (as a plugin for Eclipse) that is based on the approach presented in this paper.

References

[1] Czibula, I.G., Serban, G.: Improving Systems Design Using a Clustering Approach. International Journal of Computer Science and Network Security, VOL.6, No.12 (2006) 40–49
[2] Seng, O., Stammel, J., Burkhart, D.: Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. Proceedings of GECCO'06 (2006) 1909–1916
[3] http://asm.objectweb.org/ (2006)
[4] Xu, X., Lung, C.H., Zaman, M., Srinivasan, A.: Program Restructuring Through Clustering Technique. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), USA (2004) 75–84
[5] Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering. IEEE Computer Society Press (2001) 30-38
[6] Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers (2001)
[7] Fowler, M.: Improving the design of existing code. Addison-Wesley, New-York (1999)
[8] Simon, F., Loffler, S., Lewerentz, C.: Distance based cohesion measuring. In Proceedings of the 2nd European Software Measurement Conference (FESMA) 99, Technologisch Instituut Amsterdam (1999)
[9] Doval, D., Mancoridis, S., Mitchell, B.S.: Automatic clustering of software systems using a genetic algorithm. IEEE Proceedings of the 1999 Int. Conf. on Software Tools and Engineering Practice STEP'99 (1999)
[10] Jain, A., Murty, M.N., Flynn, P.: Data clustering: A review. ACM Computing Surveys **31** (1999) 264–323
[11] McCormick, H., Malveau, R.: Antipatterns: Refactoring Software, Architectures, and Projects in Crises. John Wiley and Sons (1998)
[12] Lung, C.H.: Software Architecture Recovery and Restructuring through Clustering Techniques. ISAW3, Orlando, SUA (1998) 101–104
[13] JHotDraw Project: http://sourceforge.net/projects/jhotdraw (1997)

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

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