Hierarchical Clustering for Software Systems Restructuring

ISTVÁN GERGELY CZIBULA[1]
GABRIELA ŞERBAN[1]

Babeş-Bolyai University
Department of Computer Science
1, M. Kogălniceanu Street, Cluj - Napoca
RO-400085, Romania
[1](istvanc,gabis)@cs.ubbcluj.ro

Abstract. Improving the quality of software systems design is the most important issue during the evolution of object oriented software systems. In this paper we are focusing on the problem of determining refactorings that can be used in order to improve the design of object oriented software systems. *Refactoring* ([6]) is a major issue to improve internal software quality. This paper aims at presenting a new hierarchical agglomerative clustering algorithm, *HARS* (Hierarchical agglomerative clustering algorithm for restructuring software systems), that identifies the refactorings needed in order to restructure a software system. *Clustering*([10]) is used in order to recondition the class structure of the 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 ([7]), emphasizing its advantages in comparison with existing approaches.

Keywords: Software engineering, system design, refactoring, hierarchical clustering.

(Received May 05, 2007 / Accepted July 13, 2007)

1 Introduction

Non-trivial software systems usually evolve over time and have many releases. These new releases resolve new requirements or are due to technological improvements. Improving the quality of software systems design is the most important issue during the evolution of object-oriented software systems.

The software maintenance cost increases with the complexity of the software systems. Without continuous restructurings of the code, the structure of the system becomes deteriorated. Thus, *program restructuring* is an important process in software evolution.

Refactoring is one major issue to increase internal software quality. It is used by most modern development methodologies (extreme programming and other agile methodologies), as a solution to keep the software structure clean and easy to maintain. Refactoring becomes an integral part of the software development cycle: developers alternate between adding new tests and functionalities and refactoring the code to improve its internal consistency and clarity.

In [6], 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 to apply appropriate refactorings in order to remove the so called "bad-smells" ([3]). All existing Integrated Development Environments offer support for automatic application of various refactorings.

Our approach takes an existing software system and reassembles it using a hierarchical agglomerative clus-

tering algorithm, in order to obtain a better design. The proposed approach would help developers to identify the appropriate refactorings. Applying the proposed refactorings remains the decision of the software engineer.

Related Work

There are various approaches in the literature in the field of *refactoring*. In [13], a search based approach for refactoring software systems structure is proposed. The authors use an evolutionary algorithm for identifying refactorings that improve the system structure.

An approach for restructuring programs written in Java starting from a catalog of bad smells is introduced in [5]. Based on some elementary metrics, the approach in [16] aids the user in deciding what kind of refactoring should be applied. The paper [15] describes a software vizualization tool which offers support to the developers in judging which refactoring to apply.

Clustering techniques have already been applied for program restructuring. In [17], a clustering based approach for program restructuring at the functional level is presented. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [11] presents a quantitative approach based on clustering techniques for software architecture restructuring and reengineering as well for software architecture recovery. It focuses on system decomposition into subsystems.

A clustering approach for identifying refactorings in order to improve the structure of software systems is developed in [4]. 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 [4]. The existing clustering approaches handle methods decomposition ([17]) or system decomposition into subsystems ([11]).

The main contributions of this paper are:

- To propose, based on the approach developed in [4], a novel *agglomerative hierarchical* clustering algorithm for identifying refactorings in order to improve the structure of software systems. The proposed approach can be useful for assisting software engineers in their daily work of restructuring software systems and it improves the approach from [4].

- To evaluate the obtained results on an open source case study ([7]) illustrating the advantages of the proposed approach in comparison with existing approaches.

The rest of the paper is structured as follows. Section 2 presents the main aspects related to the problem of *clustering* and particularly to the problem of *hierarchical clustering*. The approach (*CARD*) for determining refactorings using a clustering technique, previously introduced in [4], is presented in Section 3. A new *hierarchical agglomerative* clustering algorithm for restructuring software systems, *HARS*, is introduced in Section 4. Section 5 provides an experimental evaluation of our approach using the open source case study JHotDraw ([7]). Some conclusions and further work are given in Section 7.

2 Hierarchical Clustering

Unsupervised classification, or *clustering*, as it is more often referred as, is considered the most important *unsupervised leraning* problem. It is a data mining activity that aims to differentiate groups (classes or clusters) inside a given set of objects ([8]). 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 between two objects expresses the dissimilarity between them.

A large collection of clustering algorithms is available in the literature. [8], [9] and [10] contain comprehensive overviews of the existing techniques. Most clustering algorithms are based on two popular techniques known as *partitional* and *hierarchical* clustering.

In this paper we are focusing only on *hierarchical* clustering, that is why, in the following, an overview of the hierarchical clustering methods is presented.

Hierarchical clustering methods represent a major class of clustering techniques ([10]). There are two types of hierarchical clustering algorithms: *agglomerative* and *divisive*. 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).

Single link algorithms merge the clusters whose distance between their closest objects is the smallest. Complete link algorithms, on the other hand, merge the clusters whose distance between their most distant objects is the smallest ([10]). In general, complete link algorithms generate compact clusters while single link algorithms generate elongated clusters. Thus, complete link algorithms are generally more useful than single link algorithms.

Average link algorithms merge the clusters whose average distance (the average of distances between the objects from the clusters) is the smallest. So, average link clustering is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects ([12]).

3 Refactorings Determination using a Clustering Approach

In this section we briefly describe the clustering approach (*CARD*) that was previously introduced in [4] in order to find adequate refactorings that would improve the structure of software systems.

CARD approach 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 existing relationships between them.
- **Grouping** - The set of entities extracted at the previous step are re-grouped in clusters using a clustering algorithm (*HARS* 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 software structure in order to provide a list of refactorings which transform the original structure into an improved one.

3.1 Theoretical model

A theoretical model on which *CARD* approach is based on is introduced in [4]. Because we base our approach on this model, in the following we will briefly describe it.

Let $S = \{s_1, s_2, ..., s_n\}$ be a software system, where $s_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class.

We will consider that:

- $Class(S) = \{C_1, C_2, \ldots, C_l\}$, $Class(S) \subset S$, is the set of applications classes in the initial structure of the software system S.
- Each application class C_i ($1 \leq i \leq l$) is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, \ldots, a_{ir_i}\},\ 1 \leq p_i \leq n,\ 1 \leq r_i \leq n$, where m_{ij} ($\forall j,\ 1 \leq j \leq p_i$) are methods and a_{ik} ($\forall k,\ 1 \leq k \leq r_i$) are attributes from C_i.
- $Meth(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system S.
- $Attr(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system S.

Based on the above notations, the software system S is defined as in Equation (1):

$$S = Class(S) \bigcup Meth(S) \bigcup Attr(S). \quad (1)$$

As described above, at the **Grouping** step of *CARD*, the software system S has to be re-grouped. This re-grouping is represented in [4] as a ***partition*** of S.

Definition 1 ([4]) ***Partition of a software system*** S**.** *The set* $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ *is called a* ***partition*** *of the software system* $S = \{s_1, s_2, \ldots, s_n\}$ *iff*

- $1 \leq v \leq n$*;*
- $K_i \subseteq S, K_i \neq \emptyset, \forall i,\ 1 \leq i \leq v$*;*
- $S = \bigcup_{i=1}^{v} K_i$ *and* $K_i \cap K_j = \emptyset,\ \forall i, j, 1 \leq i, j \leq v, i \neq j$*.*

In the following, we will refer K_i as the i-th *cluster* of $\mathcal{K}$, $\mathcal{K}$ as a *set of clusters* and 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 Hierarchical Agglomerative Clustering Algorithm for Restructuring Software Systems (HARS)

In this section we propose a hierarchical agglomerative clustering algorithm, *HARS*. It aims at identifying a **partition** of a software system S that corresponds to an improved structure of it. *HARS* can be used in the *Grouping* step of *CARD* in order to re-group entities from the software system.

4.1 *HARS* algorithm

HARS algorithm is an improvement of *kRED* algorithm introduced in [4], as it can be seen in Section 5.

HARS is an adaptation of the traditional agglomerative clustering algorithm ([10]) that stops when p clusters are reached. The number p of clusters (application classes) to be determined is obtained using a heuristic that will be presented below.

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 [14] 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 from S as expressed in Equation (2).

$$d(s_i, s_j) = \begin{cases} 1 - \frac{|p(s_i) \cap p(s_j)|}{|p(s_i) \cup p(s_j)|} & \text{if } p(s_i) \cap p(s_j) \neq \emptyset \\ \infty & otherwise \end{cases} \quad (2)$$

where, for a given entity $e \in S$, $p(e)$ defines a set of relevant properties of e, expressed as:

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

We have chosen the distance between two entities as expressed in Equation (2) because it emphasizes the idea of cohesion. As illustrated in [2], "*Cohesion refers to the degree to which module components belong together*". Our distance, as defined in Equation (2), highlights the concept of cohesion, i.e., entities with low distances are cohesive, whereas entities with higher distances are less cohesive. We are currently working on giving a theoretical validation of this statement.

Based on the definition of distance d (Equation (2)) it can be easily proved that d is a semi-metric function, so a hierarchical clustering approach can be applied.

In the following we will introduce the heuristic for choosing the number of clusters. The determined number of clusters represents the number of application classes in the restructured software system. This heuristic is particular to our problem and it will provide a good enough choice for the number of application classes in the restructured software system. In order to determine the appropriate number p of clusters, we are focusing on determining p representative entities, i.e., a representative entity for each cluster.

The main idea of *HARS*'s heuristic for choosing the representative entities and the number *p* of clusters is the following:

(i) The initial number *p* of clusters is *n* (the number of entities from the software system).

(ii) The first representative entity chosen is the most "distant" entity from the set of all entities (the entity that maximizes the sum of distances from all other entities).

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

(iv) The step (iii) will be repeatedly performed, until *p* representatives will be chosen.

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 into a single application class (cluster), instead of being distributed in different application classes (clusters).

We mention that at steps (ii) and (iii) the choice could be a non-deterministic one. In the current version of *HARS* algorithm, if such a non-deterministic case exists, the first selection is chosen. Heuristics can be used

in non-deterministic selection cases. Improvements of *HARS* algorithm will deal with these kind of situations.

After determining the number p of clusters, *HARS* algorithm behaves like the classical *hierarchical agglomerative clustering* algorithm that stops when p clusters are reached.

We will consider *complete link* as linkage metric in our hierarchical agglomerative clustering approach, because we have obtained the best results with this metric. Consequently, the distance between two clusters K_i and K_j is considered to be the largest distance between the objects from the clusters, i.e,

$$dist(K_i, K_j) = max_{s' \in K_i,\ s'' \in K_j}\{d(s^{'}, s^{''})\}. \quad (3)$$

The main steps of *HARS* algorithm are:

- Determine the number p of clusters using the heuristic presented above.
- Each entity from the software system S is put in its own cluster (singleton).
- The following steps are repeated until p clusters are reached:
  - Select the two most similar clusters K_i and K_j from the current partition, i.e, the pair of clusters that minimize the distance from Equation (3).
  - Merge the clusters K_i and K_j into a single new cluster. The number of clusters in the partition is decreased.

We give next *HARS* algorithm.

Algorithm **HARS** is
Input: - the software system $\mathcal{S} = \{s_1, \ldots, s_n\}$,
- the threshold $distMin$,
- the semi-metric d between the entities.
Output: - the partition $\mathcal{K} = \{K_1, K_2, ..., K_p\}$, i.e., the new structure of $\mathcal{S}$.
Begin
//heuristically determine the number of //clusters
$p \leftarrow n$ //the initial number of clusters
//the index i_1 of the first representative //is chosen
$i_1 \leftarrow argmax_{i=1,n}\left\{\sum_{j=1, j\neq i}^{n} d(s_i, s_j)\right\}$
//nr is the number of the already chosen //clusters
$nr \leftarrow 1$
While $nr < p$ do
$D \leftarrow \{j \mid 1 \leq j \leq n,\ j \notin \{i_1, ..., i_{nr}\},$
$d = min_{l=1,nr}\{d(s_j, s_{i_l})\},\ d > distMin\}$
If $D = \emptyset$ then
//the number of clusters is decreased
$p \leftarrow p - 1$
else
//another representative is chosen
$nr \leftarrow nr + 1$
$i_{nr} \leftarrow argmax_{j \in D}\{min_{l=1,nr-1}\{d(s_j, s_{i_l})\}\}$
endif
endwhile
For i $\leftarrow$ 1 to n do
//each entity is put in its own cluster
$K_i \leftarrow \{s_i\}$
endfor
$\mathcal{K} \leftarrow \{K_1, \ldots, K_n\}$ //the initial partition
$noClus \leftarrow n$ //the initial number of clusters
While $noClus > p$ do
//the most similar clusters are chosen
$(K_i, K_j) \leftarrow argmin_{(K_{i^*}, K_{j^*})} dist(K_i^*, K_j^*)$
$K_{new} \leftarrow K_i \cup K_j$
$\mathcal{K} \leftarrow (\mathcal{K} \setminus \{K_i, K_j\}) \cup \{K_{new}\}$
$noClus \leftarrow noClus - 1$
endwhile
//$\mathcal{K}$ is the output partition
End.

In the current implementation of *HARS*, we have chosen the value 1 for the threshold $distMin$, because distances greater than 1 are obtained only for unrelated entities (Equation (2)).

We mention that *HARS* algorithm provides a partition of a software system S, partition that represents a new structure of the software system.

4.2 Refactorings identified by *HARS* algorithm

In this section we briefly discuss about the refactorings that *HARS* algorithm is able to identify. Let us assume the theoretical model from Subsection 3.1.

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 *HARS*, i.e., the new structure of S.

The main refactorings identified by *HARS* algorithm are:

1. ***Move Method*** **([6]) refactoring**.
   It moves a method m_{ij} from a class C_i into another class C_u that uses the method most; the method m_{ij} should be turned into a simple delegation, or it should be removed completely. 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 ([15]).

This refactoring is identified by *HARS* algorithm by moving the method m_{ij} into the cluster K_t corresponding to the application class C_u, i.e., $\exists t,\ 1 \leq t \leq p,\ s.t.\ C_u \in K_t,\ m_{ij} \in K_t$ and $m_{ij} \notin K_v$, where $C_i \in K_v$.

2. ***Move Attribute*** **([6]) refactoring**.
   It moves an attribute a_{ij} from a class C_i into another class C_u 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 ([15]).

   This refactoring is identified by *HARS* algorithm by moving the attribute a_{ij} into the cluster K_t corresponding to the application class C_u, i.e., $\exists t,\ 1 \leq t \leq p,\ s.t.\ C_u \in K_t,\ a_{ij} \in K_t$ and $a_{ij} \notin K_v$, where $C_i \in K_v$.

3. ***Inline Class*** **([6]) refactoring**.
   It moves all members of a class C_i into another class C_u and deletes the old class. The bad smell motivating this refactoring is that a class is not doing very much ([15]).

   This refactoring is identified by *HARS* algorithm by decreasing the number of application classes in S. Consequently, classes C_i and C_u with their corresponding entities (methods and attributes) will be merged into the same cluster K_t, i.e., $\exists t,\ 1 \leq t \leq p,\ s.t.\ C_i \in K_t,\ C_i \subset K_t,\ C_u \in K_t,\ C_u \subset K_t$.

4. ***Extract Class*** **([6]) refactoring**. It creates a new class C and moves 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 ([15]). This refactoring is identified by *HARS* algorithm by increasing the number of application classes in S. Consequently, a new cluster appears, corresponding to a new application class in the new structure of S.

We have currently implemented the above enumerated refactorings, but *HARS* 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.

5 Experimental validation

In order to validate *HARS* algorithm, we will consider two evaluations, which are described in Subsections 5.1 and 5.2. In the following, the *Data Collection* step from our approach will be briefly described.

Each of the systems evaluated in Subsections 5.1 and 5.2 are written in Java. In order to extract from the systems the data needed in the *Grouping* step of *CARD* approach (Subsection 3) we use ASM 3.0 ([1]). ASM is a Java bytecode manipulation framework. We use this framework in order to extract the structure of the systems (attributes, methods, classes and relationships between all these entities).

5.1 Code Refactoring Example

In the following we aim at illustrating how the *Move Method* refactoring is obtained after applying *HARS* algorithm. We have chosen this example in order to compare our approach with the one in [15], as this example is the only result provided by the authors.

Let us consider the Java code example shown below.

```
public class Class_A {
    public static int attributeA1;
    public static int attributeA2;
    public static void methodA1(){
      attributeA1 = 0;
      methodA2();
    }
     public static void methodA2(){
      attributeA2 = 0;
      attributeA1 = 0;
    }
    public static void methodA3(){
      attributeA2 = 0;
      attributeA1 = 0;
      methodA1();
      methodA2();
    }
}
 public class Class_B {
    private static int attributeB1;
    private static int attributeB2;
    public static void methodB1(){
      Class_A.attributeA1=0;
      Class_A.attributeA2=0;
      Class_A.methodA1();
    }
    public static void methodB2(){
      attributeB1=0;
      attributeB2=0;
    }
    public static void methodB3(){
      attributeB1=0;
      methodB1();
```

```
    methodB2();
  }
}
```

Analyzing the code presented above, it is obvious that the method **methodB1()** has to belong to **class_A**, because it uses features of **class_A** only. Thus, the refactoring *Move Method* should be applied to this method.

We have applied *HARS* algorithm, introduced in Section 4, and the *Move Method* refactoring for **methodB1()** was determined.

The two obtained clusters are:

- Cluster 1:
  {**Class_A, methodA1(), methodA2(), methodA3(), methodB1(), attributeA1, attributeA2**}.
- Cluster 2:
  {**Class_B, methodB2(), methodB3(), attributeB1, attributeB2**}.

The first cluster corresponds to application class **Class_A** and the second cluster corresponds to application class **Class_B** in the new structure of the system. Consequently, *HARS* algorithm proposes the refactoring *Move Method* **methodB1()** from **Class_B** to **Class_A**.

We mention that the refactoring proposed by our approach coincides with the one given in [15].

5.2 JHotDraw Case Study

Our second evaluation is the open source software JHotDraw, version 5.1 ([7]). 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 *HARS* algorithm on JHotDraw, i.e., how accurate are the results obtained after applying *HARS* 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. In order to capture the similarity of the two class structures (the one obtained by *HARS* algorithm and the original one) we use a measure, ACC, that was previously introduced in [4].

Let us consider the theoretical model from Subsection 3.1 and let us consider that $\mathcal{K} = \{K_1, \ldots K_p\}$ is a partition reported after applying *HARS* algorithm. We will denote by l the number of application classes from the software system S.

Definition 2 ([4]) ***Accuracy of classes recovery - ACC.*** *The accuracy of partition $\mathcal{K}$ with respect to the software system S, denoted by $ACC(S, \mathcal{K})$, is defined as:*

$$ACC(S, \mathcal{K}) = \frac{1}{l} \sum_{i=1}^{l} acc(C_i, \mathcal{K}).$$

$$acc(C_i, \mathcal{K}) = \frac{\sum_{k \in \mathcal{M}_{C_i}} \frac{|C_i \cap k|}{|C_i \cup k|}}{|\mathcal{M}_{C_i}|}$$ *(where $\mathcal{M}_{C_i} = \{K_j | 1 \leq j \leq p,\ |C_i \cap K_j| \neq 0\}$ is the set of clusters from $\mathcal{K}$ that contain elements from the application class C_i), is the accuracy of $\mathcal{K}$ with respect to application class C_i.*

ACC defines the degree to which the partition $\mathcal{K}$ is similar to S. For a given application class $C_i \in Class(S)$, $acc(C_i, \mathcal{K})$ defines the degree to which application class C_i, all its methods and all its attributes were discovered in a single cluster.

Based on Definition 2, it can be proved that $ACC(S, \mathcal{K}) \in [0, 1]$. $ACC(S, \mathcal{K}) = 1$ iff $acc(C_i, \mathcal{K}) = 1, \forall\ C_i \in Class(S)$, i.e., each application class was discovered in a single cluster. In all other situations, $ACC(S, \mathcal{K}) < 1$.

Larger values for ACC indicate better partitions with respect to S, meaning that ACC has to be maximized.

After applying *HARS* algorithm for JHotDraw case study, we have obtained the following results:

(i) $ACC = 0.974$.

(ii) The algorithm obtained a new class after the regroup-ing step, meaning that an *Extract Class* refactoring was suggested. The methods placed in the new class were: **PertFigure.handles**, **GroupFigure.handles**, **TextFigure.handles**, **StandardDrawing.handles**.

(iii) The algorithm suggested two *Move Attribute* refactorings: attributes **ColorEntry.fColor** and **ColorEntry.fName** were placed in **ColorMap** class.

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

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

- All the methods enumerated at (ii) provide similar functionalities ([7]), so, in our view, these methods can be extracted into a new class in order to avoid duplicated code, applying *Extract Class* refactoring.

- **ColorMap** and **ColorEntry** ([7]) 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, **ColorEntry.fColor** and **ColorEntry.fName** attributes can be placed in either of the two classes.

6 Advantages of our approach in comparison with previous approaches

A search-based approach for refactoring software systems structure wass proposed in [13]. The authors use an evolutionary algorithm in order to obtain a list of refactorings using JHotDraw case study.

The advantages of *CARD* approach using *HARS* algorithm, in comparison with the approach presented in [13] are illustrated bellow:

- The accuracy obtained by the refactoring technique from [13] cannot be determined, because the authors provide only the list of methods proposed to be refactored, and in order to compute ACC measure we need the complete resulting structure of the software system (including the attributes, also).

- In the approach from [13] 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 [13]. The evolutionary algorithm from [13] is executed **10** times, in order to judge how stable are the results, while *HARS* algorithm from our approach is executed just **once**.

- The overall running time for the technique from [13] is about **300** minutes (30 minutes for one run), while *HARS* algorithm in our approach provides the results in about **3.68** 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.

Based on the above considerations, *HARS* algorithm provides **better** results than the approach from [13].

In [4], a clustering approach for identifying refactorings in order to improve the structure of software systems was developed. For this purpose, a clustering algorithm, named *kRED*, was introduced.

The advantages of *HARS* algorithm introduced in this paper in comparison with *kRED* algorithm are illustrated below.

- The number of misplaced methods from both approaches is **4**.

- The overall running time for the technique from [4] is about **5** minutes , while *HARS* algorithm in our approach provide the results in about **3.68** minutes. We mention that the execution was made on similar computers.

- Unlike *kRED* algorithm, *HARS* algorithm identifies the *Extract Class* refactoring, also.

In [15], the authors describe a software vizualization tool which offers support to the developers in judging which refactoring to apply and provide a short example. A comparison between *HARS* and the approach from [15] was illustrated in Subsection 5.1.

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 [11], [17]) give only short examples indicating the obtained refactorings. Other techniques address particular refactorings: the one in [17] focuses on automated support only for identifying ill-structured or low cohesive functions and the technique in [11] focuses on system decomposition into subsystems.

7 Conclusions and Future Work

Based on the approach from [4], we have presented in this paper a new hierarchical agglomerative clustering algorithm (*HARS*) that can be used for restructuring object oriented software systems. *HARS* algorithm is used in order to obtain an improved structure of a software system, by identifying the needed refactorings. For this purpose, a heuristic that determines the number of application classes is proposed.

We have demonstrated the potential of our algorithm 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 use heuristics for the stopping criterion of *HARS* algorithm.

- To determine other distance metrics (or semi-metrics) between the entities from the software sysem.
- To use other search-based approaches in order to determine refactorings that would improve the design of a software system.
- To develop a tool (as a plugin for Eclipse) that is based on determining refactorings using *HARS* algorithm.
- To apply our approach in order to transform non object-oriented software into object-oriented systems.
- To perform a case study on a large software system for which the needed refactorings are already known.

References

[1] ObjectWeb: Open Source Middleware. http://asm.objectweb.org/.

[2] Bieman, J. M. and Kang, B.-K. Measuring design-level cohesion. *Software Engineering*, 24(2):111–124, 1998.

[3] Brown, W. J., Malveau, R. C., Hays W. McCormick, I., and Mowbray, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[4] Czibula, I. G. and Serban, G. Improving Systems Design Using a Clustering Approach. *International Journal of Computer Science and Network Security (IJCSNS)*, 6(12):40–49, 2006.

[5] Dudzikan, T. and Wlodka, J. Tool-supported discovery and refactoring of structural weakness, 2002. Masters' Thesis, TU Berlin.

[6] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, USA, 1999.

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

[8] Han, J. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[9] Jain, A. K. and Dubes, R. C. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[10] Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[11] Lung, C.-H. Software architecture recovery and restructuring through clustering techniques. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 101–104, New York, NY, USA, 1998. ACM Press.

[12] Manning, C. D. and Schutze, H. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[13] Seng, O., Stammel, J., and Burkhart, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.

[14] Simon, F., Loffler, S., and Lewerentz, C. Distance based cohesion measuring. In *Proceedings of the 2nd European Software Measurement Conference (FESMA)*, pages 69–83, Technologisch Instituut Amsterdam, 1999.

[15] Simon, F., Steinbrückner, F., and Lewerentz, C. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.

[16] Tahvildari, L. and Kontogiannis, K. A metric-based approach to enhance design quality through meta-pattern transformations. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 183–192, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Xu, X., Lung, C.-H., Zaman, M., and Srinivasan, A. Program restructuring through clustering techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*, pages 75–84, Washington, DC, USA, 2004. IEEE Computer Society.