# Hierarchical Clustering Based Automatic Refactorings Detection

ISTVAN GERGELY CZIBULA, GABRIELA (SERBAN) CZIBULA
Babeş - Bolyai University
Department of Computer Science
1, M. Kogalniceanu Street, Cluj-Napoca
ROMANIA
istvanc@cs.ubbcluj.ro, gabis@cs.ubbcluj.ro

*Abstract:* - The structure of software systems is subject of many changes during the systems lifecycle. A continuous improvement of the software systems structure can be made using *refactoring*, that assures a clean and easy to maintain software structure. In this paper we are focusing on the problem of restructuring object oriented software systems using hierarchical clustering. We propose two hierachical clustering based algorithms which takes an existing software and reassembles it using clustering, in order to obtain a better design, suggesting the needed refactorings. We evaluate the proposed algorithms using the open source case study JHotDraw and a real software system, providing a comparison with previous approaches.

*Key–Words:* - Software engineering, system design, object-oriented systems, refactoring, clustering, algorithm

## 1 Introduction

The structure of a software system has a major impact on the maintainability 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.

Fowler defines in [1] 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" [2].

In this paper we propose two hierarchical clustering algorithms that would help developers to identify the appropriate refactorings in a software system. Our approach takes an existing software and reassembles it using hierarchical clustering, in order to obtain a better design, suggesting the needed refactorings. Applying the proposed refactorings remains the decision of the software engineer.

The main contribution of this paper is to improve the approach from [3], by defining two *agglomerative hierarchical* clustering algorithms for identifying refactorings in order to recondition the class structure of software systems. The proposed algorithms can be useful for assisting software engineers in their daily work of restructuring software systems.

The rest of the paper is structured as follows. Section 2 presents existing approaches in the field of improving software systems structure and in the field of refactoring. The main aspects related to the clustering approach for systems design improvement (that we have previously introduced in [3]) are exposed in Section 3. The hierarchical clustering algorithms for identifying refactorings are introduced in Section 4. Section 5 provides an experimental evaluation of the proposed algorithms. A comparison of the approach proposed in this paper with other similar approaches is given in Section 6. Conclusions of the paper and further research directions are outlined in Section 7.

## 2 Related Work

In this section we present some approaches existing in the literature in the fields of *software clustering* and *refactoring*.

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 [5], technique that was added to the low-coupling and high-cohesion heuristics in order to capture patterns that appear commonly in software systems. In [6], a partition of a software system is refined by identifying components that belong to the wrong subsystem, and by placing them into the correct one. The paper describes a program that attempts to reverse engineer

software in order to provide better 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 [5] and [6] were not tested on large software systems, they were promising.

Mancoridis et al. introduce in [7] 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. Some extensions of *Bunch* are presented in [8], allowing user-directed clustering and incremental software structure maintenance.

A variety of software clustering approaches has 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 [5]; deducing clusters from file and procedure names [9]; utilizing the connectivity between software objects [10, 11, 12]; or looking at the problem at hand as an optimization problem [7].

Another approach for software clustering was presented in [9] 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 [13] 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 [14] 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 are various approaches in the literature in the field of *refactoring*, but few approaches exist in the direction of automatic detection of refactorings.

Deursen et al. have approached the problem of *refactoring* in [18]. The authors illustrate the difference between refactoring test code and refactoring production code, and they describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

Xing and Stroulia present in [19] an approach for detecting refactorings by analyzing the system evolution at the design level.

A search based approach for refactoring software systems structure is proposed in [20]. 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 [21]. Based on some elementary metrics, the approach in [22] aids the user in deciding what kind of refactoring should be applied. The paper [23] 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. A clustering based approach for program restructuring at the functional level is presented in [24]. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [25] 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.

We have previously introduced in [3] a clustering approach for identifying refactorings in order to improve the structure of software systems. 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 [3]. The existing clustering approaches handle methods decomposition [24] or system decomposition into subsystems [25].

# 3 A Clustering Approach for Refactorings Determination - CARD

In this section we briefly describe the clustering approach (*CARD*) that we have previously introduced in [3] in order to find adequate refactorings to improve the structure of software systems.

*CARD* approach consists of three steps shortly described below.

**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: inheritance relations, aggregation relations, dependencies between the entities from the software system. All these collected data will be used in the **Grouping** step.

**Grouping** - The set of entities extracted at the previous step are re-grouped in clusters using a clustering algorithm (*HARD* 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.

A software system $S$ is considered in [3] as a set $S = \{s_1, s_2, ..., s_n\}$, where $s_i$ $(1 \leq i \leq n)$ can be an application class, a class method or a class attribute.

As described above, at the **Grouping** step of *CARD*, the software system $S$ has to be re-grouped. This re-grouping can be viewed as a ***partition*** $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of $S$. 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 Hierarchical Clustering Algorithms for Refactorings Detection

In this section we introduce two hierarchical agglomerative clustering algorithms ($HARD_1$ and $HARD_2$), which aim at identifying a **partition** of a software system $S$ that corresponds to an improved structure of it. The proposed algorithms can be used in the **Grouping** step of *CARD*.

Both algorithms are based on the idea of hierarchical agglomerative clustering and use a measure for evaluating a partition of the software system from the system's design point of view. $HARD_1$ uses an heuristic for merging two clusters in the agglomerative process, and $HARD_2$ uses an heuristic for determining the number of clusters to obtain.

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 [26] 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).

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

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

We have chosen the distance between two entities as expressed in Equation (1), because it emphasizes the idea of cohesion. The authors define in [27] cohesion as "*the degree to which module components belong together*". 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. A theoretical validation of this statement is given in [4].

Based on the definition of distance $d$ given in Equation (1) it can be easily proved that $d$ is a semimetric function, so it can be used for discriminating the entities from the software system in a clustering approach. We will consider the distance $dist(k, k')$ between two clusters $k \in \mathcal{K}$ and $k' \in \mathcal{K}$ $(k \neq k')$ given by the average link metric, as expressed in Equation (2). We mention that we use *average link* as linkage metric, because we have obtained better results with this metric.

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

Starting from the idea that we intend to obtain high cohesive clusters and as a high cohesion between two entities from a cluster is given by a low distance (dissimilarity) between them, we will search for the clustering with the lowest overall dissimilarity. For each cluster, we define its diversity as the sum of pairwise entities dissimilarities, and we seek to minimize that sum over all clusters.

Consequently, the diversity of a partition $\mathcal{K} = \{K_1, K_2, \ldots, K_p\}$, $DIV(\mathcal{K})$, is defined as given in Equation (3).

$$DIV(\mathcal{K}) = \sum_{i=1}^{p} div(K_i), \tag{3}$$

where $div(K_i)$ represents the diversity of cluster $K_i$ and is defined as:

$$div(K_i) = \begin{cases} \sum\limits_{e \in K_i,\ e' \in K_i,\ e \neq e'} d(e, e') & \text{if } |K_i| \neq 1 \\ \infty & otherwise \end{cases}$$

Intuitively, the *diversity* of a cluster indicates the *cohesion degree* between the entities from the corresponding application class. This is due to the fact that if an entity $e$ should belong to an application class (cluster) $K_i$, then it is very likely that the *distance* (Equation (1)) between $e$ and the elements from $K_i$ is less than the *distance* between $e$ and all the other elements from the other clusters.

For these reasons, we intend to minimize the *diversity* of a partition, in order to maximize the cohesion of the corresponding application classes from the software system. We mention that we are currently working on giving a rigorous proof for this statement.

### 4.1 $HARD_1$ algorithm

The heuristic used in $HARD_1$ for merging two clusters in the agglomerative clustering process is that, at a given step, the two most 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 clusters (application classes).

The main steps of $HARD_1$ algorithm are:

- Each entity from the software system is put in its own cluster (singleton).
- The following steps are repeated until the partition remains unchanged:
  - Select the two most similar clusters from the current partition, i.e, the pair of clusters that minimizes the distance from Equation (2). Let us denote by $dmin$ this minimum distance.
  - If $dmin \leq distMin$ (the given threshold), then the clusters selected at the previous step will be merged, otherwise the partition remains unchanged. Let us denote by $\mathcal{K}$ the obtained partition.
  - Compute the diversity of partition $\mathcal{K}$ (see Equation (3)).
  - From all the generated partitions we retain the one with the smallest diversity.

We give bellow the $HARD_1$ algorithm.

Algorithm $HARD_1$ is
**Input:** - the software system $\mathcal{S} = \{s_1, \ldots, s_n\}$,
- the threshold $distMin$.
**Output:** - the partition $\mathcal{K} = \{K_1, K_2, ..., K_p\}$, i.e., the new structure of $\mathcal{S}$.
**Begin**
$\mathcal{K} \leftarrow \{K_1, \ldots, K_n\}$ //the initial partition
$minDiv \leftarrow DIV(\mathcal{K})$ // the minimum diversity
//the partition with the min. diversity
$\mathcal{K}_{min} \leftarrow \mathcal{K}$
While $\mathcal{K}$ remains unchanged do
//the most similar clusters are chosen
$(K_i, K_j) \leftarrow argmin_{(K_{i^*}, K_{j^*})} dist(K_i^*, K_j^*)$
$d \leftarrow dist(K_i, K_j)$
If $d < distMin$ then
$K_{new} \leftarrow K_i \cup K_j$
$\mathcal{K} \leftarrow (\mathcal{K} \setminus \{K_i, K_j\}) \cup \{K_{new}\}$
// the diversity of the new patition
$div \leftarrow DIV(\mathcal{K})$
If $div < minDiv$ then
$\mathcal{K}_{min} \leftarrow \mathcal{K}$
$minDiv \leftarrow div$
endif
$noClus \leftarrow noClus - 1$
endif
endwhile
//$\mathcal{K}_{min}$ is the output partition
**End.**

### 4.2 $HARD_2$ algorithm

$HARD_2$ is an adaptation of the traditional agglomerative clustering algorithm that stops when $p$ clusters are reached, $p$ being determined using a heuristic.

In the following we will present the heuristic for choosing the number $p$ of clusters. This heuristic will provide a good enough choice for the number of application classes in the restructured software system.

In order to determine the number $p$ of clusters, we are focusing on determining $p$ representative entities, i.e., a representative entity for each cluster.

The main idea of $HARD_2$'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 $HARD_2$ algorithm, if such a non-deterministic case exists, the first selection is chosen. Improvements of $HARD_2$ algorithm will deal with these kind of situations.

The main steps of $HARD_2$ 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 step is repeated until $p$ clusters are reached:
  - Select the two most similar clusters from the current partition, i.e, the pair of clusters that minimizes the distance from Equation (2).
  - Merge the two clusters obtained at the previous step into a new cluster. Let us denote by $\mathcal{K}$ the obtained partition.
  - Compute the diversity of partition $\mathcal{K}$ (see Equation (3)).
  - From all the generated partitions we retain the one with the smallest diversity.

We give next $HARD_2$ algorithm.

Algorithm $HARD_2$ is
**Input:** - the software system $\mathcal{S} = \{s_1, \ldots, s_n\}$,
- the threshold $distMin$.
**Output:** - the partition $\mathcal{K} = \{K_1, K_2, ..., K_p\}$, i.e., the new structure of $\mathcal{S}$.
**Begin**
//determine the number of clusters
$p \leftarrow n$ //the initial number of clusters
//the index of the first representative
$i_1 \leftarrow argmax_{i=1,n} \left\{ \sum_{j=1, j \neq i}^{n} d_E(s_i, s_j) \right\}$
//$nr$ is the number of chosen clusters
$nr \leftarrow 1$
While $nr < p$ do
$D \leftarrow \{j \mid 1 \leq j \leq n,\ j \notin \{i_1, ..., i_{nr}\},\ d = min_{l=1,nr} \{d_E(s_j, s_{i_l})\},\ d > distMin\}$
If $D = \emptyset$ then
//the number of clusters is decreased
$p \leftarrow p - 1$
else
//another representative is chosen
$nr \leftarrow nr + 1$
$i_{nr} \leftarrow argmax_{j \in D} \{min_{l=1,nr-1} \{d_E(s_j, s_{i_l})\}\}$
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
$minDiv \leftarrow DIV(\mathcal{K})$ // the minimum diversity
//the partition with the min. diversity
$\mathcal{K}_{min} \leftarrow \mathcal{K}$
$noClus \leftarrow n$ //the number of clusters
While $noClus > p$ do
//the most similar clusters are chosen
$(K_i, K_j) \leftarrow argmin_{(K_{i^*}, K_{j^*})} dist(K_i^*, K_j^*)$
$K_{new} \leftarrow K_i \cup K_j$
$\mathcal{K} \leftarrow (\mathcal{K} \setminus \{K_i, K_j\}) \cup \{K_{new}\}$
// the diversity of the new patition
$div \leftarrow DIV(\mathcal{K})$

```
    If div < minDiv then
      K_min ← K
      minDiv ← div
    endif
    noClus ← noClus − 1
  endwhile
  //K_min is the output partition
End.
```

For both $HARD_1$ and $HARD_2$ algorithms we have chosen the value 1 for the threshold $distMin$, because distances greater than 1 are obtained only for unrelated entities (Equation (1)). Our intuition for choosing the value for the threshold $distMin$ was also experimentally confirmed. In the future we plan to find the most appropriate value for the threshold $distMin$ using supervised learning techniques [29] and to give a rigurous proof for our selection.

The main refactorings identified by $HARD_1$ and $HARD_2$ algorithms are:

1. ***Move Method*** **[1] refactoring**. It moves a method $m$ of a class $C$ to another class $C'$ that uses the method most; the method $m$ of class $C$ 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 [23].

   This refactoring is identified by $HARD_1$ and $HARD_2$ algorithms by moving the method $m$ in the cluster corresponding to the application class $C'$.

2. ***Move Attribute*** **[1] 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 [23].

   This refactoring is identified by $HARD_1$ and $HARD_2$ algorithms by moving the attribute $a$ in the cluster corresponding to the application class $C'$.

3. ***Inline Class*** **[1] refactoring**. It moves all the 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 [23].

   This refactoring is identified by $HARD_1$ and $HARD_2$ algorithms by decreasing the number of application classes in the new structure of $S$. Classes $C$ and $C'$ with their corresponding entities (methods and attributes) will be merged in the same cluster.

4. ***Extract Class*** **[1] 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 [23].

   This refactoring is identified by $HARD_1$ and $HARD_2$ algorithms by increasing the number of application classes in the new structure of $S$. 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 $HARD_1$ and $HARD_2$ algorithms 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 Evaluation

In order to validate our clustering approach, we will consider three evaluations, which are described in Subsections 5.1, 5.2 and 5.3. In the following, we will briefly describe the *Data Collection* step from our approach.

Each of the systems evaluated in Subsections 5.1, 5.2, and 5.3 are written in Java. In order to extract from the systems the data needed in the *Grouping* step of our approach (Section 3) we use ASM 3.0 [30]. 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 Examples

The first case study, described in this subsection, contains two simple examples to provide the reader with easy to follow examples of refactoring extractions.

**Example 1.**

We aim at illustrating how the *Move Method* refactoring is obtained after applying $HARD_1$ and $HARD_2$ algorithms. We have chosen this example in order to compare our approach with the one in [23], as this example is the only result provided by the authors.

Let us consider the Java code example from [23] 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 $HARD_1$ and $HARD_2$ algorithms, 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, $HARD_1$ and $HARD_2$ algorithms propose the refactoring *Move Method* **methodB1()** from **Class_B** to **Class_A**.

We mention that the refactoring proposed by our algorithms coincides with the one given in [23].

**Example 2.**

We aim to illustrate how the *Move Attribute* refactoring is obtained after applying $HARD_1$ and $HARD_2$ algorithms. Let us consider the Java code example shown below.

```
public class Class_A {
   public static int attributeA2;
   public static int attributeA1;
   public static void methodA1() {
     methodA2();
   }
    public static void methodA2() {
     attributeA2 = 0;
   }
    public static void methodA3() {
     attributeA2 = 0;
     methodA1();
     methodA2();
   }
}

public class Class_B {
   private static int attributeB1;
   private static int attributeB2;
    public static void methodB1() {
     attributeB1 = 0;
     Class_A.methodA1();
   }
    public static void methodB2() {
     attributeB1 = 0;
     attributeB2 = 0;
     Class_A.attributeA1 = 12;
   }
    public static void methodB3() {
     attributeB1 = 0;
     methodB1();
     methodB2();
     Class_A.attributeA1 = 12;
   }
    public static void methodB4() {
     attributeB1 = 0;
     methodB2();
   }
}
```

Analyzing the code presented above, it is obvious that the attribute **attributeA1** has to belong to **Class_B**, because it is used only by methods from **Class_B**. Thus, the refactoring *Move Attribute* should be applied to this attribute.

We have applied $HARD_1$ and $HARD_2$ algorithms, introduced in Section 4, and the *Move Attribute* refactoring for **attributeA1** was determined.

The obtained clusters are:

- Cluster 1:
  $\{$**Class_A, methodA1(), methodA2(), methodA3(), attributeA2**$\}$.
- Cluster 2:
  $\{$**Class_B, methodB1(), methodB2(), methodB3(), methodB4(), attributeA1, 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, $HARD_1$ and $HARD_2$ algorithms propose the refactoring *Move Attribute* **attributeA1** from **Class_A** to **Class_B**.

## 5.2 JHotDraw Case Study

Our second evaluation is the open source software JHotDraw, version 5.1 [31]. 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 $HARD_1$ and $HARD_2$ algorithms in comparison with the current design of JHotDraw. As JHotDraw has a good class structure, $HARD_1$ and $HARD_2$ algorithms should generate a nearly identical class structure. We evaluate how similar is a partition the partition of JHotDraw determined after applying $HARD_1$ and $HARD_2$ algorithms with its actual partition (that is considered a good partition, as JHotDraw is well known for a good design).

After applying $HARD_1$ algorithm on JHotDraw we have obtained the following results:

(i) The algorithm obtains a new class after the regrouping 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) In the obtained partition there are no misplaced attributes.

(iii) In the obtained partition there are four misplaced methods.

In our view, the refactoring identified at (i) can be justified. All these methods provide similar functionalities, that is why, in our view, these methods can be extracted in a new class in order to avoid duplicated code, applying *Extract Class* refactoring.

After applying $HARD_2$ algorithm we have obtained the following results:

(i) The algorithm obtains a new class after the regrouping 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.

In our view, the refactorings identified at (ii) can also be justified. **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.

## 5.3 A Real Software System

This subsection describes the last case study used for experimentally evaluate our approach. As shown in Subsection 5.2, $HARD_1$ obtains better results on JHotDraw than $HARD_2$, that is why in this subsection we select only $HARD_1$ algorithm for evaluation. The chosen case study is a DICOM (*Digital Imaging and Communications in Medicine*) [32] and HL7 (*Health Level 7*) [33] compliant PACS (*Picture Archiving and Communications System*) system, facilitating medical images management, offering quick access to radiological images, and making the diagnosis process easier.

The analyzed application is a distributed system, currently used by hospitals in locations such as Romania, United Kingdom, South Africa, Bulgaria and the Republic of Moldova. It is a large system that consists

of several subsystems in form of stand-alone and web-based applications. We have applied *HARD* algorithm on one of the subsystems from this application.

For confidentiality reasons, we will refer the analyzed application as $\mathcal{A}$. $\mathcal{A}$ is a stand-alone Java application used for archiving radiological images for long time storage (on CD, DVD or tape). $\mathcal{A}$ consists of **675** classes, **5759** methods and **2970** attributes.

After applying $HARD_1$ algorithm, a total of $56$ refactorings have been suggested: $4$ *Move Attribute* refactorings, $51$ *Move Method* refactorings, and $1$ *Extract Class* refactoring.

The obtained results have been analyzed by the developers of $\mathcal{A}$ and the following conclusions were made: $17$ refactorings identified by $HARD_1$ were accepted by the developers as useful in order to improve the system; $13$ refactorings were acceptable for the developers, but they concluded that these refactorings are not necessary in the current stage of the project; $26$ refactorings were strongly rejected by the developers.

Analyzing the obtained results, based on the feedback provided by the developers, we have concluded the following.

$HARD_1$ successfully identified classes with low cohesion (classes with more than one responsibility), misplaced constants (constants used only on a subtree of a class hierarchy, but defined in some base class). These kind of weaknesses can be discovered only if the developer manually inspects all the classes, or if a bug arises. That is why automatic detection by $HARD_1$ of these kind of weaknesses can prevent system failure or other kind of bugs and also save a lot of manual work.

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 appear frequently in projects developed in JAVA. In order to correctly deal with these aspects, we have to improve only the **Data collection** step of our approach, without modifying $HARD_1$ 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 consider 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* [34], etc. Future improvements of our approach will deal with these aspects, also.

# 6 Comparative analysis with existing approaches

In this section we aim at providing a comparison between $HARD_1$ algorithm and similar existing approaches. We have chosen only $HARD_1$ for evaluation, because it obtains better results on JHotDraw than $HARD_2$, as shown in Subsection 5.2.

A complete comparison between our approach and the approaches from [21], [22], [24] and [25] can not be provided, because of the following reasons:

- The obtained results for relevant case studies are not available. There are given only short examples indicating the obtained refactorings.
- The techniques [24] and [25] address particular refactorings: the one in ([24]) focuses on automatic support only for identifying ill-structured or low cohesive functions and the technique in [25] focuses on system decomposition into subsystems.

A comparison between our approach and the one presented in [23] is illustrated in Subsection 5.1. The example from Subsection 5.1 is the only result given by the paper [23].

The only approach on the topic studied in this paper, that partially gives the results obtained on a relevant case study (like JHotDraw) is [20]. The authors use an evolutionary algorithm in order to obtain a list of refactorings using case study JHotDraw.

The advantages of *CARD* approach using $HARD_1$ algorithm, in comparison with the approach presented in [20] are illustrated bellow:

- Our technique is deterministic, in comparison with the approach described in [20]. The evolutionary algorithm from [20] is executed **10** times, in order to judge how stable are the results, while $HARD_1$ algorithm from our approach is executed just **once**.
- The technique from [20] reports **10** misplaced methods, while in our approach there are only **4** misplaced methods.
- The overall running time for the technique from [20] is about **300** minutes (30 minutes for one run), while $HARD_1$ algorithm in our approach provide the results in about **4.73** 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.

The authors from [23] present a short example illustrating the *Move method* refactoring. We have applied $HARD_1$ algorithm on the code example from [23] and the suggested refactoring was obtained by our algorithm, also.

A clustering approach for identifying refactorings in order to improve the structure of software systems is developed in [3]. For this purpose, a clustering algorithm (named *kRED*) is introduced and an evaluation of *kRED* algorithm on JHotDraw case study is provided.

A comparison between $HARD_1$ algorithm introduced in this paper and *kRED* algorithm is illustrated in Table 1. The comparison is made considering the following characteristics: the number of misplaced methods, the number of misplaced attributes, the running time and if the algorithm identifies or not the *Extract Class* refactoring. We mention that the execution was made on similar computers.

Table 1: Comparative results

| Alg. | No. of misplaced methods | No. of misplaced attributes | Running time | *Extract Class* refactoring |
|---|---|---|---|---|
| $HARD_1$ | 4 | 0 | 4.73 | Yes |
| *kRED* | 4 | 4 | 5 | No |

From Table 1 we can conclude that:

- The results obtained by $HARD_1$ are better than the results provided by *kRED* algorithm, as the numbers of methods and attributes misplaced by $HARD_1$ are less than those misplaced by *kRED* algorithm.
- The running time of $HARD_1$ is less than the running time of *kRED*.
- $HARD_1$ algorithm, unlike *kRED*, identifies the *Extract Class* refactoring, also.

We cannot make a complete comparison with other refactoring approaches existing in the literature, because, for most of them, the obtained results for relevant case studies are not available. Most approaches give only short examples indicating the obtained refactorings. Other techniques address particular refactorings: the one in [24] focuses on automated support only for identifying ill-structured or low cohesive functions and the technique in [25] focuses on system decomposition into subsystems.

# 7 Conclusions and Further Work

Starting from the approach introduced in [3], we have presented in this paper two hierarchical clustering algorithms ($HARD_1$ and $HARD_2$) that can be used for improving software systems design.

We have demonstrated the potential of our algorithms by applying them to the open source case study JHotDraw and a real software system 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 other search based approaches, clustering techniques [15, 16, 17], and machine learning techniques [29] in order to determine refactorings that improve the design of a software system.
- To improve the *distance* semi-metric used for discriminating the entities from the software system.
- To develop a tool (as a plugin for Eclipse) that is based on determining refactorings using $HARD_1$ and $HARD_2$ algorithms.
- To apply $HARD_1$ and $HARD_2$ algorithms for other case studies, like JEdit [35].
- To apply our approach in order to transform non object-oriented software into object-oriented systems.

**Acknowledgements:** This work was supported by the research project TD No. 411/2008, sponsored by the Romanian National University Research Council.

*References:*

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

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

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

[4] G. Serban and I.G. Czibula, On evaluating software systems design, *Studia Universitatis "Babes-Bolyai", Informatica*, LII(1), 2007, pp. 55–66.

[5] R. W. Schwanke and M. A. Platoff, Cross references are features, in *Proceedings of the 2nd International Workshop on Software configuration management*. New York, NY, USA: ACM Press, 1989, pp. 86–95.

[6] R. W. Schwanke, An intelligent tool for reengineering software modularity, in *ICSE '91: Proceedings of the 13th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 83–92.

[7] 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, in *IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98)*. Piscataway, NY: IEEE Press, 1998, pp. 45–52.

[8] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures, in *ICSM*, 1999, pp. 50–59.

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

[10] D. H. Hutchens and V. R. Basili, System structure analysis: clustering with data bindings, *IEEE Trans. Softw. Eng.*, vol. 11, no. 8, 1985, pp. 749–757.

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

[12] S. C. Choi and W. Scacchi, Extracting and restructuring the design of large systems, *IEEE Softw.*, vol. 7, no. 1, 1990, pp. 66–71.

[13] V. Tzerpos and R. C. Holt, Mojo: A distance metric for software clusterings, in *Working Conference on Reverse Engineering*, 1999, pp. 187–193.

[14] V. Tzerpos and R. C. Holt, ACDC: An algorithm for comprehension-driven clustering, in *Working Conference on Reverse Engineering*, 2000, pp. 258–267.

[15] N. P. Lin, C.-I. Chang, H.-E. Chueh, H.-J. Chen, and W.-H., Hao, *A Deflected Grid-based Algorithm for Clustering Analysis*, *WSEAS Transactions on Computers*, Issue 3, vol. 7, 2008, pp. 125–132.

[16] W., Barbakh and C., Fyfe, *A Novel Construction of Connectivity Graphs for Clustering and Visualization*, *WSEAS Transactions on Computers*, Issue 5, vol. 7, 2008, pp. 424–434.

[17] J.R., Avila, A.F., Ramirez, C.A., Cruz, and I., Vasquez-Alvarez, *The Clustering Algorithm for Nonlinear System Identification*, *WSEAS Transactions on Computers*, Issue 7, vol. 7, 2008, pp. 1179–1188.

[18] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, Refactoring test code, in *In M. Marchesi, editor, Extreme Programming and Flexible Processes; Proc. XP2001, 2001.*, 2001, pp. 92–95.

[19] Z. Xing and E. Stroulia, Refactoring detection based on UMLDiff change-facts queries, *WCRE*, 2006, pp. 263–274.

[20] O. Seng, J. Stammel, and D. Burkhart, 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*. New York, NY, USA: ACM Press, 2006, pp. 1909–1916.

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

[22] L. Tahvildari and K. Kontogiannis, 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 183–192.

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

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

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

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

[27] J. M. Bieman and B.-K. Kang, Measuring design-level cohesion, *Software Engineering*, vol. 24, no. 2, 1998, pp. 111–124.

[28] A. K. Jain, M. N. Murty, and P. J. Flynn, Data clustering: a review, *ACM Computing Surveys*, vol. 31, no. 3, 1999, pp. 264–323.

[29] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

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

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

[32] Digital Imaging and Communications in Medicine, http://medical.nema.org/.

[33] Health Level 7, www.hl7.org/.

[34] T. DeMarco, Structured analysis and system specification, *Software pioneers: contributions to software engineering*, 2002, pp. 529–560.

[35] JEdit Programmer's Text Editor, http://www.jedit.org.