Improving Systems Design Using a Clustering Approach

István Gergely Czibula [†] **and Gabriela Şerban**[††],

Babeş-Bolyai University, Faculty of Mathematics and Computer Science,
1, M. Kogălniceanu Street, RO-400084 Cluj-Napoca, Romania

Summary
Clustering is a division of data into groups of similar objects, a data mining activity that aims to differentiate groups inside a given set of objects, with respect to a set of relevant attributes of the analyzed objects. *Refactoring* is the process of improving the design of software systems. Its goal is to change a software system in such a way that it does not alter the external behavior of the code, but improves its internal structure ([9]). This paper aims at presenting a new approach for improving systems design using *clustering*. 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 ([18]) based on two newly defined measures. A comparison with previous approaches is also provided.
Key words:
Software Engineering, Refactoring, System Design, Clustering.

Introduction

The original design of a software system is rarely prepared for every new requirement which appears over the software system's life cycle. Due to tight schedules which appear in real life software development process, different people have to make quickly changes in the systems. Without continuously restructuring the code, the system becomes difficult to understand and change, and therefore it is often costly to maintain.

In many software development methodologies (extreme programming and other agile methodologies), refactoring is a solution to keep the software structure clean and easy to maintain. Nowadays, refactoring becomes an integral part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity.

In [9], 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" ([14]).

All existing Integrated Development Environments offers support for automatic application of various refactorings. In this paper we are focusing on developing a technique that would help developers to identify the appropriate refactorings.

Our approach takes an existent software and reassembles it using clustering, in order to obtain a better design, suggesting the needed 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 [1], 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 [4] aids the user in deciding what kind of refactoring should be applied. The paper [7] 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 [3] a clustering based approach for program restructuring at the functional level is presented. This approach focuses on automated support

Manuscript received December 5, 2006.
Manuscript revised December 25, 2006.

for identifying ill-structured or low cohesive functions. The paper [15] 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.

However, to our knowledge, there is no approach in the literature that uses clustering in order to improve the class structure of a system. The existing clustering approaches handle methods decomposition ([3]) or system decomposition into subsystems ([15]).

The main contributions of this paper are:
- To propose a new *k-means* based clustering approach 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.
- To evaluate the obtained results on an open source case study ([18]) based on two newly defined measures.

The rest of the paper is structured as follows. Section 2 presents the main aspects related to the problem of *clustering*. Our approach (*CARD*) for determining refactorings using a clustering technique is proposed in Section 3. Section 4 provides an experimental evaluation of the proposed approach using the open source case study JHotDraw ([18]). Some conclusions and further work are outlined in Section 5.

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 ([8]), being considered the most important *unsupervised learning* problem. The inferring process is carried out with respect to a set of relevant characteristics or attributes of the analyzed objects. 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 $O = \{O_1, O_2, ..., O_n\}$ be the set of objects to be clustered. Using the vector-space model, each object is measured with respect to a set of l initial attributes, $A = \{A_1, A_2, ..., A_l\}$, and is therefore described by a l-dimensional vector $O_i = \{O_{i1}, O_{i2}, ..., O_{il}\}, O_{ik} \in \Re$, $1 \le i \le n, 1 \le k \le l$. Usually, the attributes associated to objects are standardized in order to ensure an equal weight to all of them ([8]).

The measure used for discriminating objects can be any *metric* or *semi-metric* function $d : O \times O \to \Re$ (Minkowski distance, Euclidian distance, Manhattan distance, Hamming distance, etc). The distance between two objects expresses the dissimilarity between them. Consequently, the *similarity* between two objects O_i and O_j is defined as

$$sim(O_i, O_j) = \frac{1}{d(O_i, O_j)}.$$

A large collection of clustering algorithms is available in the literature. [8], [16] and [12] 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 *k-means* clustering, that is why, in the following, an overview of the partitioning clustering methods is presented.

2.1 Partitioning Methods. The *k-means* Clustering Algorithm

A well-known class of clustering methods is the one of the partitioning methods, with representatives such as the *k-means* algorithm or the *k-medoids* algorithm. Essentially, given a set of n objects and a number $k, k \le n$, such a method divides the object set into k distinct and non-empty clusters. The partitioning process is iterative and heuristic; it stops when a "good" partitioning is achieved. Finding a "good" partitioning coincides with optimizing a criterion function defined either locally (on a subset of the objects) or globally (defined over all of the objects, as in *k-means*). These algorithms try to minimize certain criteria (a squared error function); the squared error criterion tends to work well with isolated and compact clusters ([12]).

Partitional clustering algorithms are generally iterative algorithms that converge to local optima. The most widely used partitional algorithm is the iterative *k-means* approach. The objective function that the *k-means* optimizes is the squared sum error (*SSE*). The *SSE* of a partition $K = \{K_1, K_2, ..., K_p\}$ is defined as:

$$SSE(K) = \sum_{j=1}^{p} \sum_{i=1}^{n_j} d^2(O_i^j, f_j)$$

where the cluster K_j is a set of objects $\{O_1^j, O_2^j, ..., O_{n_j}^j\}$ and f_j is the centroid (mean) of K_j:

$$f_j = \left(\frac{\sum_{k=1}^{n_j} O_{k1}^j}{n_j}, ..., \frac{\sum_{k=1}^{n_j} O_{kl}^j}{n_j} \right).$$

Hence, the *k-means* algorithm minimizes the intra-cluster distance. The *k-means* algorithm partitions a collection of *n* objects into *k* distinct and non-empty clusters, data being grouped in an exclusive way (each object will belong to a single cluster) ([16]). The algorithm starts with *k* initial centroids, then iteratively recalculates the clusters (each object is assigned to the closest cluster - centroid), and their centroids until convergence is achieved.

The main disadvantages of *k-means* are:
- The performance of the algorithm depends on the initial centroids. So, the algorithm gives no guarantee for an optimal solution.
- The user needs to specify the number of clusters in advance.

3. Clustering Approach for Refactorings Determination (*CARD*)

In this section we propose a new *k-means* based clustering approach (*CARD*) that aims at finding adequate refactorings in order to improve the structure of software systems.

CARD approach consists of three steps:
- **Data collection**. The existent 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 grouping algorithm (*kRED* 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 software structure in order to provide a list of refactorings which transform the original structure into an improved one.

The above described steps offer a general view of our approach. In the following we introduce a theoretical model on which our clustering approach is based, and a more detailed description of *CARD*.

3.1 Theoretical Model

In this subsection we present a theoretical model that will be used in order to explain and evaluate our approach.

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

- $Class(S) = \{C_1, C_2, ..., C_l\}, Class(S) \subset S$, is the set of applications classes in the initial structure of the software system *S*.
- Each application class C_i $(1 \le i \le l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, ..., m_{ip_i}, a_{i1}, a_{i2}, ..., a_{ir_i}\}, 1 \le p_i < n,$ $1 \le r_i < n$, where m_{ij} $(1 \le j \le p_i)$ are methods and a_{ij} $(1 \le j \le r_i)$ 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 all 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 our approach, the software system *S* has to be re-grouped. In our view, this re-grouping is represented as a *partition* of *S*.

Definition 1. ***Partition of a software system S.***
The set $K = \{K_1, K_2, ..., K_v\}$ is called a *partition* of the software system $S = \{s_1, s_2, ..., s_n\}$ iff
- $1 \le v \le n$;
- $K_i \subseteq S, K_i \ne \emptyset, \forall i, 1 \le i \le v$;

– $S = \bigcup_{i=1}^{v} K_i$ *and* $K_i \bigcap K_j = \varnothing, \forall i, j 1 \le i, j \le v, i \ne j$.

In our approach, we use a *k-means* based clustering algorithm in order to obtain a partition of the software system. In the following we will refer to K_i as the *i*-th *cluster* of K, to K as a *set of clusters* and to an element s_i from *S* as an *entity*. A cluster K_i from the partition K represents an application class in the new structure of the software system.

3.2 Our approach

Based on the theoretical model described in Subsection 3.1, let us consider a software system $S = \{s_1, s_2, ..., s_n\}$. Our focus is to improve the design of the software system *S* by determining a *partition* of *S* that corresponds to an improved structure of the software system.

In the following we will describe the steps of *CARD*.

Data Collection
The existent software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existent relationships between them. The relevant entities can be extracted from existent documents of the software system, like: source code, UML diagrams, or other documents that provide the needed information. This step is particular to the analyzed software system. A detailed explanation of this step for our case study is provided in Section 4.

Grouping
At this step we propose to re-group entities from a software system using a vector space model based *clustering* algorithm, more specifically a variant of the *k-means* clustering algorithm, named *kRED* (*k-means for REfactorings Determination*).

It is well known that the violation of the principle "Put together what belong together" is the main symptom for ill-structured software systems. In order to capture this aspect, we have to measure the degree to which some parts of the system belong together.

In our approach the objects to be clustered are the entities from the software system *S*, i.e., $O = \{s_1, s_2, ..., s_n\}$. As we intend to group methods and attributes in classes, we will consider the attribute set as the set of application classes from the software system *S*, $A = \{C_1, C_2, ..., C_l\}$, i.e., the cardinality of the vector space model in our approach is the number *l* of application classes from the software system *S*. Our focus is to group similar entities from *S* in order to obtain high cohesive groups (clusters).

In the literature there exist many cohesion measures, like the ones defined in [19], [13], [10]. We will adapt the generic cohesion measure introduced in [10] 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, for a given entity from the software system *S*, the dissimilarity degree between the entity and the application classes *C* from *S*, $\forall C, C \in Class(S)$.

So, each entity s_i $(1 \le i \le n)$ from the software system *S* is characterized by a *l*-dimensional vector: $(s_{i1}, s_{i2}, ..., s_{il})$, where $s_{ij}, (\forall j, 1 \le j \le l)$ is computed as given in Equation (2):

$$s_{ij} = \begin{cases} -\dfrac{|\, p(s_i) \bigcap p(C_j) \,|}{|\, p(s_i) \bigcup p(C_j) \,|} & \text{if } p(s_i) \bigcap p(C_j) \ne \varnothing \\ \infty & \text{otherwise} \end{cases} \quad (2)$$

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 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 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 attributes and methods defined in the class.

We will consider that the distance between two entities s_i and s_j from the software system *S* is expressed using the *Euclidian distance* between their associated vectors, as:

$$d_E(s_i, s_j) = \sqrt{\sum_{k=1}^{l} (s_{ik} - s_{jk})^2}.$$

We have chosen Euclidian distance in our approach, because of the following reasons:

- Intuitively, as the elements from the vector characterizing an entity represent the dissimilarity degree to the application classes, the *Euclidian distance* assigns low distances to entities that have to belong to the same application class.
- It is the most widely used distance measure in clustering ([12]).
- We have obtained better results using *Euclidian distance* than using other metrics.

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

(i) The initial number of clusters is the number l of application classes from the software system S.
(ii) The initial centroids are chosen as the application classes from S.
(iii) As in the classical *k-means* approach, the clusters (centroids) are recalculated, i.e., each object is assigned to the closest cluster (centroid).
(iv) Step (iii) is repeatedly performed until two consecutive iterations remain unchanged, or the number of steps performed exceeds the maximum number of iterations allowed.

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

- The reason for choosing at steps (i) and (ii) the number of classes and as centroids the classes is that we intend to group entities (methods, attributes) around classes.
- If, at a given moment, a cluster becomes empty, this means that the number of clusters will be decreased.
- Because the initial centroids are the application classes from the software system, the dependence of the algorithm on the initial centroids is eliminated.
- Because the initial number l of centroids (clusters) is known (the number of application classes), a *k-means* based clustering approach is suitable.

3.3 Refactorings Extraction

In this section we briefly discuss about the refactorings that *CARD* approach is able to identify.

Let us consider that S is the analyzed software system, as defined in Subsection 3.1, and that $K = \{K_1, K_2, ..., K_l\}$ is the partition provided by *kRED*, i.e., the new structure of S. The main refactorings identified by *kRED* algorithm are:

1. ***Move Method* ([9]) refactoring**.
   It moves a method m_{ij} of a class C_i to another class C_u that uses the method most; the method m_{ij} of class C_i 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 ([7]).

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

2. ***Move Attribute* ([9]) refactoring**.
   It moves an attribute a_{ij} of a class C_i to 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 ([7]).

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

3. ***Inline class* ([9]) 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 ([7]).

   This refactoring is identified by *kRED* algorithm by decreasing the number of elements in the partition K. Consequently, the number of application classes in the new structure of S becomes l-1, and classes C_i and C_u with their corresponding entities (methods and attributes) will be merged in the same cluster K_t, i.e.,

$\exists t, 1 \le t \le l-1, s.t.\ C_i \in K_t, C_i \subset K_t, C_u \in K_t, C_u \subset K_t$.

From the clustering point of view, this case appears when, at a given iteration, a cluster becomes empty.

We have currently implemented the above enumerated refactorings, but *kRED* algorithm can also identify other refactorings, like: *Pull Up Attribute*, *Pull Down Attribute*, *Pull Up Method*, *Pull Down Method*, *Collapse Class Hierarchy*. Future improvements will deal with these situations, also.

4. Experimental Evaluation

In order to validate our clustering approach, we will consider two evaluations, which are described in Subsections 4.1 and 4.2. In the following, we will briefly describe the *Data Collection* step from our approach.

Each of the systems evaluated in Subsections 4.1 and 4.2 are written in Java. In order to extract from the systems the data needed in the *Grouping* step of our approach (Subsection 3.2) we use ASM 3.0 ([2]). 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).

4.1 Code Refactoring Example

We aim at illustrating how the *Move Method* refactoring is obtained after applying *kRED* algorithm.

Let us consider the Java code example shown in Figure 1. The example is similar to the one presented in [7]. We have chosen this example in order to compare our approach with the one in [7], as this example is the only result provided by the authors.

```
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();
  }
}
```

Fig. 1 Code example.

Analyzing the code presented in Figure 1, 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 *kRED* algorithm, introduced in Section 3, 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, *CARD* 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 [7].

4.2 JHotDraw Case Study

Our second evaluation is the open source software JHotDraw, version 5.1 ([18]). 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. Table 1 gives an overview of the system's size.

Table 1: JHotDraw Statistic

Classes	**173**
Methods	**1375**
Attributes	**475**

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 *CARD* approach introduced in Section 3 on JHotDraw, i.e., how accurate are the results obtained after applying *kRED* 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 *kRED* algorithm and the original one) we propose two measures. Both measures evaluate how similar is a partition of the software system *S* determined after applying *kRED* algorithm with a good partition of the software system (as the actual partition of JHotDraw is considered to be).

In the following, let us consider the theoretical model presented in Subsection 3.1 and a refactoring technique *T* (a technique that determines a refactoring of the software system *S* to be analyzed). We assume that the software system *S* has a good design (as JHotDraw has) and $K = \{K_1, K_2, ..., K_l\}$ is a partition reported after applying *kRED* algorithm. We mention that *l* represents the number of application classes from the software system *S*.

Definition 2. ***ACCuracy of a refactoring technique - ACC.***.

Let *T* be a refactoring technique. The accuracy of *T* with respect to a partition *K* and the software system *S*, denoted by $ACC(S,K,T)$, is defined as:

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

$$acc(C_i,K,T) = \frac{\sum_{j \in M_{C_i}} \frac{|C_i \cap K_j|}{|C_i \cup K_j|}}{|M_{C_i}|}$$ (where M_{C_i} is the set of clusters from *K* that contain elements from the application class C_i, $M_{C_i} = \{j \mid 1 \le j \le l, C_i \cap K_j \neq \varnothing\}$), is the accuracy of *T* with respect to the application class C_i.

In our view, *ACC* defines the degree to which the partition *K* is similar to *S*. For a given application class $C_i \in Class(S)$, $acc(C_i,K,T)$ defines the degree to which application class C_i, all its methods and all its attributes were discovered by *T* in a single cluster.

Based on Definition 2, it can be proved that $ACC(S,K,T) \in [0,1]$.

$ACC(S,K,T) = 1$ iff $acc(C_i,K,T) = 1$, $\forall C_i \in Class(S)$, i.e., each application class was discovered in a single cluster. In all other situations, $ACC(S,K,T) < 1$.

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

Definition 3. ***PRECision of a refactoring technique - PREC.***.

Let *T* be a refactoring technique. The precision of *T* with respect to a partition *K* and the software system *S*, denoted by $PREC(S,K,T)$, is defined as:

$$PREC(S,K,T) = \frac{1}{|Meth(S)|} \cdot \sum_{m \in Meth(S)} prec(m,K,T).$$

$$prec(m,K,T) = \begin{cases} 1 & \text{if } m \text{ was placed in the same class as in } S \\ 0 & \text{otherwise} \end{cases}$$

,is the precision of *T* with respect to the method *m*.

In our view, $PREC(S,K,T)$ defines the percentage of methods from *S* that were correctly discovered by *T* (we say that a method is correctly discovered if it is placed in its original application class).

Based on Definition 3, it can be proved that $PREC(S,K,T) \in [0,1]$.

$PREC(S,K,T) = 1$ iff $prec(m,K,T) = 1$, $\forall m \in Meth(S)$, i.e., each method was discovered in its original application class. In all other situations, $PREC(S,K,T) < 1$.

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

After applying *CARD* for JHotDraw case study, we obtain the following values for the measures *ACC* and *PREC*:

- ACC=0.9829.
- PREC=0.9956.

Regarding to *PREC*, there were only 6 methods that were misplaced in the partition obtained after applying *kRED* algorithm. The names of the methods that were proposed to be moved is shown in the first column of Table 2. The suggested target class is shown in the second column.

Table 2: The misplaced methods

Method	Target Class
PertFigure.writeTasks	StorableOutput
PertFigure.readTasks	StorableInput
PolygonFigure.distanceFromLine	Geom
StandardDrawingView.drawing-Invalidated	DrawingChangeEvent
ColorEntry.fName	ColorMap
ColorEntry.fColor	ColorMap

From our perspective, all the proposed refactorings can be justified.
Consider, for example, the **PertFigure.writeTasks** method presented below ([18]).

```
public void writeTasks(StorableOutput dw, Vector v) {
      dw.writeInt(v.size());
      Enumeration i = v.elements();
      while (i.hasMoreElements())
         dw.writeStorable((Storable) i.nextElement());
}
```

As we can observe from the source code above, the method **writeTasks** writes a list of **Storable** elements, without directly using attributes or methods from **PertFigure** class. The responsability of **StorableOutput** class is to manage the storage of different storable objects. So, in our opinion, the best place for **writeTasks** method would be the class **StorableOutput**.

The need for refactoring *Move Method* **PertFigure.readTasks** to **StorableInput** class can be similarly justified.
Another proposed refactoring is *Move Method* **PolygonFigure.distanceFromLine** to **Geom** class.

```
public static double distanceFromLine(int xa, int ya, int xb,
                                      int yb, int xc, int yc) {
      int xdiff = xb - xa;
      int ydiff = yb - ya;
      long l2 = xdiff * xdiff + ydiff * ydiff;
      if (l2 == 0)
         return Geom.length(xa, ya, xc, yc);
      double rnum = (ya - yc) * (ya - yb) - (xa - xc) * (xb - xa);
      double r = rnum / l2;
      if (r < 0.0 || r > 1.0)
         return Double.MAX_VALUE;
      double xi = xa + r * xdiff;
      double yi = ya + r * ydiff;
      double xd = xc - xi;
      double yd = yc - yi;
         return Math.sqrt(xd * xd + yd * yd);
}
```

This method computes the distance from a given point to a line. It does not directly use attributes and methods from **PolygonFigure** class. The class **Geom** consists of a set of utility methods, so, in our opinion, the move of method **PolygonFigure.distanceFromLine** to **Geom** class is justifiable.

4.3 Comparisons with other approaches

The only approach on the topic studied in this paper, that partially gives the results obtained on a relevant case study (like JHotDraw) is [1]. The authors use an evolutionary algorithm in order to obtain a list of refactorings using case study JHotDraw.

The advantages of *CARD* in comparison with the approach presented in [1] are illustrated bellow:

- We have applied our precision measure *PREC* (Definition 3) on the results obtained by the technique from [1] and we obtained a precision of **0.9949**, that is less that the precision (**0.9956**) obtained by our technique.
- The accuracy obtained by the refactoring technique from [1] 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).
- Our technique is deterministic, in comparison with the approach from [1]. The evolutionary algorithm from [1] is executed **10** times, in order to judge how stable are the results, while *kRED* algorithm from our approach is executed just **once**.
- The overall running time for the technique from [1] is about **300** minutes (30 minutes for one run), while

kRED algorithm in our approach provide the results in about **5** 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.

A comparison between our approach and the one presented in [7] is given in Subsection 4.1. A more detailed comparison cannot be provided, because the paper [7] presents only a short example, and no other results.

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 [3], [15]) give only short examples indicating the obtained refactorings. Other techniques address particular refactorings: the one in [3] focuses on automated support only for identifying ill-structured or low cohesive functions and the technique in [15] focuses on system decomposition into subsystems.

5. Conclusions and Further Work

We have presented in this paper a novel approach, *CARD*, for improving systems design using *clustering*. More precisely, we have introduced *kRED* algorithm, a *k-means* based clustering algorithm in order to obtain an improved structure of a software system.

CARD proposes a list of refactorings that can be useful for assisting software engineers in their daily works of refactoring software systems.

We have defined a theoretical model on which we base our approach. We have demonstrated the potential of *CARD* 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 *CARD* for other case studies, like JEdit ([6]).
- To use other approaches for clustering, such as hierarchical clustering ([12]), search based clustering ([11]), or genetic clustering ([17]).
- To improve the vector space model used for clustering.
- To use other search based approaches in order to determine refactorings that improve the design of a software system.
- To develop a tool (as a plugin for Eclipse) that is based on *CARD*, the approach presented in this paper.
- To apply our approach in order to transform non object-oriented software into object-oriented systems.

References

[1] 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
[2] http://asm.objectweb.org/ (2006)
[3] 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
[4] Tahvildari, L., Kontogiannis, K.: A mteric based approach to enhance design quality through meta-pattern transformations. Proceedings of the seventh European Conference on Software Maintenance and Reengineering (2003)
[5] Dudzikan, T., Wlodka, J.: Tool-supported dicovery and refactoring of structural weakness. Masters' Thesis, TU Berlin (2002)
[6] jEdit Programmer's Text Editor: http://www.jedit.org (2002)
[7] Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering. IEEE Computer Society Press (2001) 30–38
[8] Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers (2001)
[9] Fowler, M.: Improving the design of existing code. Addison-Wesley, New-York (1999)
[10] 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)
[11] 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)
[12] Jain, A., Murty, M.N., Flynn, P.: Data clustering: A review. ACM Computing Surveys **31** (1999) 264-323
[13] Bieman, J.M., Kang, B.-K.: Measuring Design-Level Cohesion. In: IEEE Transactions on Software Engineering **24** No. 2 (1998)
[14] McCormick, H., Malveau, R.: Antipatterns: Refactoring Software, Architectures, and Projects in Crises. John Wiley and Sons (1998)
[15] Lung, C.H.: Software Architecture Recovery and Restructuring through Clustering Techniques. ISAW3, Orlando, SUA (1998) 101-104
[16] Jain, A., Dubes, R.: Algorithms for Clustering Data. Prentice Hall, Englewood Cliffs, New Jersey (1998)

[17] Cole, R.M.: Clustering with genetic algorithms. Master's thesis, University of Western Australia (1998)
[18] JHotDraw Project: http://sourceforge.net/projects/jhotdraw (1997)
[19] Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. In: IEEE Transactions on Software Engineering **20**, No. 6 (1994) 476-493

István Gergely Czibula has graduated from Babeş-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science in 2002. He is a PhD student at the Department of Computer Science, Faculty of Mathematics and Computer Science from the Babeş-Bolyai University of Cluj-Napoca, Romania. His main research interest is software engineering.

Gabriela Şerban has graduated from Babeş-Bolyai University of Cluj-Napoca, Faculty of Mathematics and Computer Science in 1992. She has received the PhD degree in Computer Science in 2003, with the "cum laude" distinction. She is an associate professor at the Department of Computer Science, Faculty of Mathematics and Computer Science from the Babeş-Bolyai University of Cluj-Napoca, Romania. Her research interests include artificial intelligence, machine learning, multiagent systems, programming paradigms.