Clustering Based Adaptive Refactoring

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

Abstract: - It is well-known that maintenance and evolution represent important stages in the lifecycle of any software system, about 66% from the total cost of the software systems development. Improving the software systems design through refactoring is one of the most important issues during the evolution of object oriented software systems. Refactoring aims at changing a software system in such a way that it does not alter the external behavior of the code, but improves its internal structure. In this paper we approach the problem of adaptive refactoring, and we propose an adaptive method to cope with the evolving structure of any object oriented application. Namely, we handle here the case when new application classes are added to the software system and the current restructuring scheme must be accordingly adapted. The approach proposed in this paper extends our previous clustering based approach for identifying refactorings in an object oriented software system. We also provide an example illustrating the efficiency of the proposed approach.

Key–Words: Restructuring, adaptive refactoring, clustering

1 Introduction

The software systems, during their life cycle, are faced with new requirements. These new requirements imply updates in the software systems structure, that have to be done quickly, due to tight schedules which appear in real life software development process. The structure of a software system has a major impact on the maintainability of the system. This structure is the subject of many changes during the system lifecycle. Improper implementations of these changes imply structure degradation that leads to costly maintenance. 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.

Refactoring is a solution adopted by most modern software development methodologies (extreme programming and other agile methodologies), in order 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 functionality and refactoring the code to improve its internal consistency and clarity.

Fowler defines in [9] 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" [5].

We have developed in [7] a clustering based approach, named *CARD* (*Clustering Approach for Refactorings Determination*) that uses clustering for improving the class structure of a software system. In this direction, a partitional clustering algoritm, *kRED* (*k-means for REfactorings Determination*), was developed. The algorithm suggests the refactorings needed in order to improve the structure of the software system. The main idea is that clustering is used in order to obtain a better design, suggesting the needed refactorings.

Real applications evolve in time, and new application classes are added in order to met new requirements. Consequently, restructuring of the modified system is nedeed to keep the software structure clean and easy to maintain. Obviously, for obtaining the restructuring that fits the new applications classes, the original restructuring scheme can be applied from scratch on the whole extended system. However, this process can be inefficient, particularly for large software systems. That is why we propose an adaptive method to cope with the evolving application classes set. The proposed method extends our original ap-

proach previously introduced in [7].

The rest of the paper is structured as follows. Section 2 briefly presents the main aspects related to clustering and our previous approach for clustering based refactorings identification [7]. A clustering based approach for adaptive refactorings identification is introduced in Section 3. For the adaptive process, a *Core Based Adaptive Refactoring* algorithm (*CBAR*) is proposed. Section 4 indicates several existing approaches in the direction of automatic refactorings identification. An example illustrating how our approach works is provided in Section 5. Some conclusions of the paper and further research directions are outlined in Section 6.

2 Background

In this section we present some background related to *clustering* and our previous approach for clustering based refactorings identification.

2.1 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 [20], being considered the most important *unsupervised learning* problem. The inferring process is, usually, 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 $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$ 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, $\mathcal{A} = \{A_1, A_2, \ldots, A_l\}$, and is therefore described by a l-dimensional vector $O_i = (O_{i1}, \ldots, O_{il}), O_{ik} \in \Re, 1 \leq i \leq n, 1 \leq k \leq l$. Usually, the attributes associated to objects are standardized in order to ensure an equal weight to all of them [20].

The measure used for discriminating objects can be any *metric* or *semi-metric* function $d : \mathcal{O} \times \mathcal{O} \longrightarrow \Re$ (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. [20], [21] and [22] contain comprehensive overviews of the existing techniques. Most clustering algorithms are based on two popular techniques known as *partitional* and *hierarchical* clustering.

In the following, a short overview of the partitioning methods is presented.

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 $k, k \leq n$, 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 [22].

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 $\mathcal{K} = \{K_1, K_2, ...K_p\}$ is defined as:

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

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:

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

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) [21].

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.

2.2 *CARD* clustering approach for refactorings identification

We have introduced in [7] a clustering approach *CARD* for identifying refactorings that would improve the class structure of a software system. *CARD* consists of three steps:

- **Data collection** - The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system. All these colletcted 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 grouping algorithm. 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.

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 elements from the considered software system, i.e., $\mathcal{S} = \{e_1, e_2, \ldots, e_n\}$, where $e_i, 1 \leq i \leq n$ can be an application class, a method from a class or an attribute from a class. In the following, we will refer an element $e_i \in \mathcal{S}$ as an *entity*.

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 $\mathcal{S}$, $\mathcal{A} = \{C_1, C_2, \ldots, C_l\}$, i.e., the cardinality of the vector space model in our approach is the number l of application classes from the software system $\mathcal{S}$.

The focus is to group similar entities from $\mathcal{S}$ in order to obtain high cohesive groups (clusters).

In the literature many cohesion measures exist, like the ones defined in [6, 14]. 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, for a given entity from the software system $\mathcal{S}$, the dissimilarity degree between the entity and the application classes from $\mathcal{S}$.

So, each entity e_i $(1 \leq i \leq n)$ from the software system S is characterized by a l-dimensional vector: $(e_{i1}, e_{i2}, \ldots, e_{il})$, where e_{ij} $(\forall j,\ 1 \leq j \leq l)$ is computed as follows:

$$e_{ij} = \begin{cases} 1 - \frac{|p(e_i) \cap p(C_j)|}{|p(e_i) \cup p(C_j)|} & \text{if } \ p(e_i) \cap p(C_j) \neq \emptyset \\ \infty & otherwise \end{cases}, \tag{2}$$

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

- If e is an attribute, then $p(e)$ consists of: the attribute itself, the application class where the attribute is defined, and all methods from $\mathcal{S}$ 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 attributes from $\mathcal{S}$ accessed by the method, all the methods from $\mathcal{S}$ used by method e, and all methods from $\mathcal{S}$ that overwrite method e.

- If e is an application class, then $p(e)$ consists of: the application class itself, all attributes and methods defined in the class, all interfaces implemented by class e and all classes extended by class e.

As in a vector space model based clustering [22], we consider the *distance* between two entities e_i and e_j from the software system $\mathcal{S}$ as a measure of dissimilarity between their corresponding vectors. In our approach we will consider *Euclidian distance*, i.e., the distance between e_i and e_j is expressed as in Equation (3):

$$d(e_i, e_j) = \sqrt{\sum_{k=1}^{l} (e_{ik} - e_{jk})^2} \tag{3}$$

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

- Intuitively, as the elements from the vector characterizing an entity represents 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 [22].
- 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 $\mathcal{S}$.

(ii) The initial centroids are chosen as the application classes from $\mathcal{S}$.

(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 performed number of steps exceeds the maximum allowed number of iterations.

We mention that the partition obtained by *kRED* algorithm represents a new and improved structure of it, which indicates the refactorings needed to restructure the system.

3 Our approach

In the following we will present our core based clustering approach for adaptive refactoring.

Let us consider a software system $\mathcal{S}$. As presented in Subsection 2.2, the *kRED* algorithm provides a restructuring scheme that gives the refactorings needed in $\mathcal{S}$ in order to improve its structure.

During the evolution and maintenance of $\mathcal{S}$, new application classes are added to it in order to met new functional requirements. Let us denote by $\mathcal{S}'$ the software system $\mathcal{S}$ after extension. Consequently, restructuring of $\mathcal{S}'$ is nedeed to keep its structure clean and easy to maintain. Obviously, for obtaining the restructuring that fits the new applications classes, the original restructuring scheme can be applied from scratch, i.e., *kRED* algorithm should be applied considering all entities from the modified software system $\mathcal{S}'$. However, this process can be inefficient, particularly for large software systems.

That is why we extend the approach from [7] and we propose an adaptive method to cope with the evolving application classes set. Namely, we handle here the case when new application classes are added to the software system and the current restructuring scheme must be accordingly adapted. The main idea is that instead of applying *kRED* algorithm from scratch on the modified system $\mathcal{S}'$, we adapt the partition obtained by *kRED* algorithm for the initial system $\mathcal{S}$, considering the newly added application classes. Using the adaptive process, we aim at reducing the time needed for obtaining the results, without altering the accuracy of the restructuring process.

In this section we will introduce our approach for adaptive refactoring, starting from the approach introduced in [7].

3.1 Theoretical model

Let $\mathcal{S} = \{e_1, e_2, \ldots, e_n\}$ be the set of entities from the software system. Each entity is measured with respect to a set of l attributes, $\mathcal{A} = \{C_1, C_2, \ldots, C_l\}$ (the application classes from S) and is therefore described by a l-dimensional vector: $e_i = (e_{i1}, e_{i2}, \ldots, e_{il}), e_{ik} \in \Re^+, 1 \leq i \leq n, 1 \leq k \leq l$. By l we denote the number of application classes from $\mathcal{S}$.

Let $\mathcal{K} = \{K_1, K_2, \ldots, K_l\}$ be the partition (set of clusters) discovered by applying *kRED* algorithm on the software system $\mathcal{S}$. Each cluster from the partition is a set of entities, $K_j = \{e_1^j, e_2^j, \ldots, e_{n_j}^j\}, 1 \leq j \leq l$. The centroid (cluster mean) of the cluster K_j is denoted by f_j, where $f_j = \left(\frac{\sum_{k=1}^{n_j} e_{k1}^j}{n_j}, \ldots, \frac{\sum_{k=1}^{n_j} e_{kl}^j}{n_j} \right)$.

The measure used for discriminating two entities from $\mathcal{S}$ is the *Euclidian distance* between their corresponding l dimensional vectors, denoted by d.

Let us consider that the software system $\mathcal{S}$ is extended by adding s ($s \geq 1$) new application classes, $C_{l+1}, C_{l+2}, \ldots, C_{l+s}$. Consequently, the set of attributes will be extended with s new attributes, corresponding to the newly added application classes. After extension, the modified software system becomes $\mathcal{S}' = \{e'_1, e'_2, \ldots, e'_n, e'_{n+1}, e'_{n+2}, \ldots, e'_{n+m}\}$, where

- $e'_i, 1 \leq i \leq n$ is the entity $e_i \in \mathcal{S}$ after extension.
- $e'_i, \forall n+1 \leq i \leq n+m$ are the entities (classes, methods and attributes) from the newly added application classes $C_{l+1}, C_{l+2}, \ldots, C_{l+s}$.

We mention than each entity from the extended software system is characterized by a $l+s$ dimensional

vector, i.e. $e'_i = (e_{i1}, \ldots, e_{il}, e_{i,l+1}, \ldots, e_{i,l+s}), \forall 1 \leq i \leq n+m$.

We want to analyze the problem of grouping the entities from $\mathcal{S}'$ into clusters, after the software system's extension and starting from the partition $\mathcal{K}$ obtained by applying $kRED$ algorithm on the software system $\mathcal{S}$ (before application class extension). We aim to obtain a performance gain with respect to the partitioning from scratch process.

The partition $\mathcal{K}'$ of the extended software system $\mathcal{S}'$ corresponds to its improved structure. Following the idea from [7], the number of clusters from $\mathcal{K}'$ should be the number of application classes from $\mathcal{S}'$, i.e. l+s.

We start from the fact that, at the end of the initial *kRED* clustering process, all entities from $\mathcal{S}$ are closer to the centroid of their cluster than to any other centroid. So, for any cluster $K_j \in \mathcal{K}$ and any entity $e_i^j \in K_j$, inequality below holds.

$$d(e_i^j, f_j) \leq d(e_i^j, f_r), \forall j, r,\ 1 \leq j, r \leq l,\ r \neq j. \tag{4}$$

We denote by $K'_j, 1 \leq j \leq l$, the set containing the same entities as K_j, after the extension. By $f'_j, 1 \leq j \leq l$, we denote the mean (center) of the set K'_j. These sets $K'_j, 1 \leq j \leq l$, will not necessarily represent clusters after the attribute set extension. The newly arrived attributes (application classes) can change the entities' arrangement into clusters, formed so that the intra-cluster similarity to be high and inter-cluster similarity to be low. But there is a considerable chance, when adding one or few attributes to entities, that the old arrangement in clusters to be close to the actual one. The actual clusters could be obtained by applying the *kRED* clustering algorithm on the set of extended entities. But we try to avoid this process and replace it with one less expensive but not less accurate. With these being said, we agree, however, to continue to refer the sets K'_j as clusters.

The partition $\mathcal{K}'$ should also contain clusters corresponding to the newly added application classes. The initial centroids of these clusters are considered to be the newly added application classes themselves, i.e. $f'_j = C_j, \forall l+1 \leq j \leq l+s$.

We therefore take as starting point the previous partitioning into clusters (as explained above) and study in which conditions an extended object $e_i^{j\prime}$ is still correctly placed into its cluster K'_j. For that, we express the distance of $e_i^{j\prime}$ to the center of its cluster, f'_j, compared to the distance to the center f'_r of any other cluster K'_r.

Theorem 1 When inequalities (5) and (6) hold for an extended entity $e_i^{j\prime}$ *($1 \leq j \leq l$)*

$$e_{iv}^j \geq \frac{\sum_{k=1}^{n_j} e_{kv}^j}{n_j}, \forall\, v \in \{l+1, l+2, \ldots, l+s\} \tag{5}$$

and

$$d(e_i^{j\prime}, f'_j) \leq d(e_i^{j\prime}, C_v), \forall\, v \in \{l+1, l+2, \ldots, l+s\} \tag{6}$$

then the entity $e_i^{j\prime}$ is closer to the center f'_j than to any other center $f'_r, 1 \leq j,\ r \leq l+s,\ r \neq j$.

Proof

We prove below this statement.

First, we will prove that the entity $e_i^{j\prime}$ is closer to the center f'_j than to any other center $f'_r, 1 \leq j,\ r \leq l,\ r \neq j$.

$$d^2(e_i^{j\prime}, f'_j) - d^2(e_i^{j\prime}, f'_r) =$$

$$d^2(e_i^j, f_j) + \sum_{v=l+1}^{l+s} \left(\frac{\sum_{k=1}^{n_j} e_{kv}^j}{n_j} - e_{iv}^j \right)^2 -$$

$$d^2(e_i^j, f_r) - \sum_{v=l+1}^{l+s} \left(\frac{\sum_{k=1}^{n_r} e_{kv}^r}{n_r} - e_{iv}^j \right)^2.$$

Using the inequality (4), we have:

$$d^2(e_i^{j\prime}, f'_j) - d^2(e_i^{j\prime}, f'_r) \leq \sum_{v=l+1}^{l+s} \left(\frac{\sum_{k=1}^{n_j} e_{kv}^j}{n_j} - e_{iv}^j \right)^2 -$$

$$\sum_{v=l+1}^{l+s} \left(\frac{\sum_{k=1}^{n_r} e_{kv}^r}{n_r} - e_{iv}^j \right)^2$$

$$\Leftrightarrow$$

$$d^2(e_i^{j\prime}, f'_j) - d^2(e_i^{j\prime}, f'_r) \leq \sum_{v=l+1}^{l+s} \left(\frac{\sum_{k=1}^{n_j} e_{kv}^j}{n_j} - \frac{\sum_{k=1}^{n_r} e_{kv}^r}{n_r} \right) \cdot$$

$$\left(\frac{\sum_{k=1}^{n_j} e_{kv}^j}{n_j} + \frac{\sum_{k=1}^{n_r} e_{kv}^r}{n_r} - 2 \cdot e_{iv}^j \right).$$

If the inequality (5) holds for every new attribute of $e_i^{j\prime}$, then the inequality above becomes:

$d^2(e_i^{j\prime}, f'_j) - d^2(e_i^{j\prime}, f'_r) \leq$

$$-\sum_{v=l+1}^{l+s}\left(\frac{\sum_{k=1}^{n_j} e_{kv}^{j}}{n_j}-\frac{\sum_{k=1}^{n_r} e_{kv}^{r}}{n_r}\right)^2 \Leftrightarrow$$

$d^2(e_i^{j\prime}, f_j') - d^2(e_i^{j\prime}, f_r') \leq 0.$

Because all distances are non-negative numbers, it follows that:

$$d(e_i^{j\prime}, f_j') \leq (e_i^{j\prime}, f_r'), \forall r,\ 1 \leq r \leq l,\ r \neq j. \quad (7)$$

It is obvious that inequality (6) indicates that the entity $e_i^{j\prime}$ is closer to the center f_j' than to any other center $f_r', l+1 \leq j,\ r \leq l+s,\ r \neq j$.

Consequently, from (7) and (6), we can conclude that the entity $e_i^{j\prime}$ is closer to the center f_j' than to any other center $f_r', 1 \leq j,\ r \leq l+s,\ r \neq j$.

□

We have to notice that the inequality in (5) imposes only intra-cluster conditions. An entity is compared against its own cluster in order to decide its new affiliation to that cluster.

3.2 The *Core Based Adaptive Refactoring* Algorithm

We will use the property enounced in the previous subsection in order to identify inside each cluster $K_j', 1 \leq j \leq l$, those entities that have a considerable chance to remain stable in their cluster, and not to move into another cluster as a result of the software system's class (attribute set) extension. In our view, these entities form the *core* of their cluster. In the following definition we will consider that $1 \leq j \leq l$.

Definition 2

a) *We denote by* $StrongCore_j = \{e_i^{j\prime} | e_i^{j\prime} \in K_j', e_i^{j\prime}$ *satisfies the set of inequalities (5) and (6)*$\}$ *the set of all objects in* K_j' *satisfying inequalities (5) and (6) for each new attribute (class)* $v,\ l+1 \leq v \leq l+s$.

b) *Let* $sat(e_i^{j\prime})$ *be the set of all new attributes* $v,\ l+1 \leq v \leq l+s$, *for which object* $e_i^{j\prime}$ *satisfy inequalities (5) and (6).*
We denote by $WeakCore_j = \{e_i^{j\prime} | e_i^{j\prime} \in K_j', |sat(e_i^{j\prime})| \geq \frac{\sum_{k=1}^{n_j} |sat(e_k^{j\prime})|}{n_j}\}$ *the set of all entities in* K_j' *satisfying inequalities (5) and (6) for at least the average number of attributes for which (5) and (6) hold.*

c) $Core_j = StrongCore_j$ *iif* $StrongCore_j \neq \emptyset$; *otherwise,* $Core_j = WeakCore_j$.

We mention that the initial number of centroids (clusters) in the adaptive clustering algorithm is the number of application classes after the extension of S, i.e., l+s.

In the following we will present our idea in choosing the first l initial centroids and initial clusters from the partition that will be adapted. We will assume in the following that $1 \leq j \leq l$. For each new application class (attribute) $C_v,\ l+1 \leq v \leq l+s$, and each cluster K_j' there is at least one entity that satisfies the inequality (5) with respect to the attribute C_v. Namely, the entity that has the greatest value for attribute C_v between all entities in K_j' certainly satisfies inequality (5) (the maximum value in a set is greater or equal than the mean of the values in the set). But it is not sure that there is in cluster K_j' any entity that satisfies relations (5) and (6) for all new application classes (attributes) $C_{l+1}, \ldots, C_{l+s}$. If there are such entities ($StrongCore_j \neq \emptyset$), we know that, according to Theorem 1, they are closer to the cluster center f_j' than to any other cluster center $f_r', 1 \leq r \leq l+s, r \neq j$. Then, $Core_j$ will be taken to be equal to $StrongCore_j$ and will be the seed for cluster j in the adaptive algorithm. But if $StrongCore_j = \emptyset$, then we will choose as seed for cluster j other entities, the most stable ones between all entities in K_j'. These entities ($WeakCore_j$) can be less stable than would be the entites in $StrongCore_j$. This is not, however, a certain fact: the entities in the "weaker" set $WeakCore_j$ can be as good as those is $StrongCore_j$. This comes from the fact that Theorem 1 enounces a *sufficient* condition for the entities in K_j' to be closer to f_j' than to any other f_r', but not a *necessary* condition, too.

The *cluster cores*, chosen as we described, will serve as seed in the adaptive clustering process. All entities in $Core_j$ will surely remain together in the same group if clusters do not change. This will not be the case for all core entities, but for most of them.

We have presented above the idea for choosing the initial l centroids and clusters. Considering that s new application classes are added to the software system S, the next s centroids are chosen as the newly added classes, i.e., $f_j' = C_j, \forall l+1 \leq j \leq l+s$.

The adaptive algorithm starts by calculating the old clusters' cores. The cores will be the new initial clusters from which the adaptive process begins. Next, the algorithm proceeds in the same manner as the classical *k-means* method does.

We mention that the algorithm stops when the clusters from two consecutive iterations remain unchanged or the number of steps performed exceeds a maximum allowed number of iterations.

We give next the *Core Based Adaptive Refactoring* algorithm.

Algorithm CBAR is
Input:
- the software system $\mathcal{S} = \{e_1, \ldots, e_n\}$ of l-dimensional entities
- l, the number of classes from $\mathcal{S}$
- the set of newly added classes $\{C_{l+1}, \ldots, C_{l+s}\}$
- the extended software system $\mathcal{S}' = \{e'_1, \ldots, e'_{n+m}\}$ of (l+s)-dimensional extended objects
- the metric d between objects in the l-dimensional space,
- s, the number of added classes
- $noMaxIter$ the maximum allowed number of iterations.
- $\mathcal{K} = \{K_1, \ldots, K_l\}$ the partition of entities in $\mathcal{S}$ reported by $kRED$.

Output:
- the re-partitioning of the entities from $\mathcal{S}'$, $\mathcal{K}' = \{K'_1, \ldots, K'_{l+s}\}$.

Begin
```
@ The old cluster cores are computed
For j ← 1,l do
   Core_j ← (StrongCore_j ≠ ∅)?StrongCore_j :
            WeakCore_j
 @ The initial centroids are determined
   f'_j ← the mean of objects in Core_j
EndFor
@ Centroids corresponding to the added
@ application classes are determined
For j ← l+1, l+s do
   f'_j ← C_j
EndFor
K' ← {f'_1, f'_2 ... f'_{l+s}}
While (K' changes) and (there were
   not performed noMaxIter iterations) do
    For j ← 1, l+s do
      @ The clusters are recalculated
      K'_j ← {e_i | ∀f_k, d(e_i, f_j) ≤ d(e_i, f_k)}
    EndFor
    For j ← 1, l+s do
      If K'_j = ∅ then
        @ remove element K'_j from K'
      //the no. of clusters is decreased
       Else
        f'_j ← the mean of K'_j
      Endif
   EndFor
EndWhile
@ K' is the output partition
```
End.

Remark 3 *We mention two main characteristics of* CBAR *algorithm: (a) the time complexity for calculating the cores in the clustering process does not grow the complexity of the global calculus; (b) the method for calculating the core of a cluster C (using inequality (2)) depends only on the current cluster (does not depend on other clusters).*

In order to identify possible improvements of *CBAR* algorithm, we have considered the following adaptation:

- When calculating the core $Core_j$ of a certain cluster K_j ($1 \leq j \leq l$), we will consider not only $StrongCore_j$, but $WeakCore_j$ as well. Consequently, if $StrongCore_j$ is not empty, then the seed for cluster j is set as the union between $StrongCore_j$ and $WeakCore_j$, as illustrated in Equation (8). Our intuition for this choice is that it is a large enough probability that the entities from $WeakCore_j$ to be stable in the cluster, and this will increase the efficiency of the adaptive process.

$$Core_j = (StrongCore_j \neq \emptyset)? \\ StrongCore_j \bigcup WeakCore_j : WeakCore_j. \tag{8}$$

In the following, we will dentote this variation of *CBAR* algorithm as $CBAR_v$.

4 Related Work

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

Deursen et al. have approached the problem of *refactoring* in [17]. 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 [18] 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 [13]. 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 [8].

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. A clustering based approach for program restructuring at the functional level is presented in [19]. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [12] 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 based approach for identifying the most appropriate refactorings in a software system is introduced in [7]. Based on the approach from [7], a hierarchical clustering algorithm for refactorings identification is developed in [2].

Fatiregun et al. [23] applied genetic algorithms to identify transformation sequences for a simple source code, with 5 transformation array, whilst we have applied 6 distinct refactorings to 23 entities. Seng et al. [25] apply a weighted multi-objective search, in which metrics are combined into a single objective function. An heterogeneous weighed approach is applied here, since the weight of software entities in the overall system and refactorings cost are studied. Mens et al. [24] propose the techniques to detect the implicit dependencies between refactorings.

To our kowledge, there are no existing approaches in the literature in the direction of adaptive refactoring, as it is approached in this paper.

5 Experimental Evaluation

In this section we present an experimental evaluation of *CBAR* algorithm on a simple case study. We aim to provide the reader with an easy to follow example of adaptive refactorings extraction. Let us consider the software system $\mathcal{S}$ consisting of the Java code example shown below. The evaluation was also made for the variation $CBAR_v$ of $CBAR$ algorithm presented in Subsection 3.2.

```
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 *kRED* algorithm, and the *Move Method* refactoring for **methodB1()** was determined. A partition $\mathcal{K} = \{K_1, K_2\}$ was obtained, where $K_1 = \{$**Class_A, methodA1(), methodA2(), methodA3(), methodB1(), attributeA1, attributeA2**$\}$ and $K_2 = \{$**Class_B, methodB2(), methodB3(), attributeB1, attributeB2**$\}$.

Cluster K_1 corresponds to application class **Class_A** and cluster K_2 corresponds to application class **Class_B** in the new structure of the system. Consequently, *kRED* proposes the refactoring *Move Method* **methodB1()** from **Class_B** to **Class_A**.

Let us consider that the system is now extended with another class, **Class_C**. Let us denote by $\mathcal{S}'$ the extended software system.

```
public class Class_C {
  private static int attributeC1;
  private static int attributeC2;
  public static void methodC1(){
          Class_A.attributeA1=0;
          Class_A.methodA2();
    }
  public static void methodC2(){
          attributeC1=0;
          attributeC2=0;
```

```
    }
  public static void methodC3(){
          attributeC1=0;
          methodC1();
          methodC2();
  }
}
```

Analyzing the newly added application class, it is obvious that the method **methodC1()** 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.

Consequently, a partition $\mathcal{K}' = \{K'_1, K'_2, K'_3\}$ of the extended system has to be obtained, with clusters K'_1, K'_2 and K'_3 corresponding to the restructured classes **class_A**, **class_B** and **class_C** respectively, i.e., $K'_1 = \{$**Class_A, methodA1(), methodA2(), methodA3(), methodB1(), methodC1(), attributeA1, attributeA2**$\}$, $K'_2 = \{$**Class_B, methodB2(), methodB3(), attributeB1, attributeB2**$\}$ and $K'_3 = \{$**Class_C, methodC2(), methodC3(), attributeC1, attributeC2**$\}$.

There are two possibilities to obtain the restructured partition $\mathcal{K}'$ of the extended system $\mathcal{S}'$.

1. To apply $kRED$ algorithm from scratch on the extended system containing all the entities from application classes **class_A**, **class_B** and **class_C**.
2. To adapt, using $CBAR$ algorithm, the partition $\mathcal{K}$ obtained after applying $kRED$ algorithm before the system's extension.

We comparatively present in Table 1 the results obtained after applying $kRED$, $CBAR$ and $CBAR_v$ algorithms for restructuring the extended system $\mathcal{S}'$. We mention that all algorithms have identified the partition $\mathcal{K}'$ corresponding to the improved structure of $\mathcal{S}'$.

Table 1: The results

No (l) of classes from $\mathcal{S}$	2
No of entities from $\mathcal{S}$	12
No (s) of newly added classes	1
No (l+s) of classes from $\mathcal{S}'$	3
No of entities from $\mathcal{S}'$	18
No of $kRED$ iterations for (l+s) attributes	3
No of $CBAR$ iterations for (l+s) attributes	2
No of $CBAR_v$ iterations for (l+s) attributes	2

From Table 1 we observe that *CBAR* algorithm finds the solution in a smaller number of iterations than $kRED$ algorithm. This confirms that the time needed by $CBAR$ to obtain the results is reduced, and this leads to an increased efficiency of the adaptive process. For large software systems, it is very likely that the number of iterations performed by $CBAR$ will be significantly reduced in comparison with the number of iterations performed by $kRED$.

We also notice that the number of iterations performed by $CBAR$ and $CBAR_v$ is the same, but we think that for larger systems $CBAR_v$ algorithm will find the solution in a smaller number of iterations than $CBAR$.

6 Conclusions and Future Work

We have proposed in this paper a new method for adapting a restructuring scheme of a software system when new application classes are added to the system. The considered experiment proves that the result is reached more efficiently using *CBAR* method than running *kRED* again from the scratch on the extended software system.

Further work will be done in the following directions:

- To isolate conditions to decide when it is more effective to adapt (using *CBAR*) the partitioning of the extended software system than to recalculate it from scratch using *kRED* algorithm.
- To apply the adaptive algorithm *CBAR* on open source case studies and real software systems.
- To study the apropriateness of other clustering algorithms [3, 4] for refactorings identification and to develop adaptive variants of them.
- To study the apropriateness of adaptive fuzzy clustering algorithms [1] for refactorings identification.

Acknowledgements: This work was supported by the research project ID_2286, No. 477/2008, sponsored by the Romanian National University Research Council (CNCSIS).

References:

[1] Chih-Hung Hsu, Tzu-Yuan Lee, Hui-Ming Kuo. Mining the Body Features to Develop Sizing Systems to Improve Business Logistics andMarketing Using Fuzzy Clustering Data Mining. *WSEAS Transactions on Computers*, 8(7), 2009, pp.1215–1224.

[2] I.G. Czibula and G. Serban. Software systems design improvement using hierarchical clustering. *WSEAS TRansactions on Electronics*, 5(7), 2008, pp.291–302.

[3] J. Avila, A. Ramirez, C. Cruz, and I. Vasquez-Alvarez. The clustering algorithm for nonlinear system identification. *WSEAS Transactions on Computers*, 7(7), 2008, pp.1179–1188.

[4] W. Barbakh and C. Fyfe. A novel construction of connectivity graphs for clustering and visualization. *WSEAS Transactions on Computers*, 7(5), 2008, pp.424–434.

[5] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[6] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] I.G. Czibula and G. Serban. Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[8] T. Dudzikan and J. Wlodka. Tool-supported discovery and refactoring of structural weakness. Master's thesis, TU Berlin, Germany, 2002.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

[11] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, New Jersey, 1998.

[12] Chung-Horng Lung. 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.

[13] Olaf Seng, Johannes Stammel, and David 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*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.

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

[15] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. 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] Ladan Tahvildari and Kostas 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*, pages 183–192, Washington, DC, USA, 2003. IEEE Computer Society.

[17] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. pages 92–95, 2001.

[18] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on UMLDiff change-facts queries. *WCRE*, pages 263–274, 2006.

[19] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan. 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.

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

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

[22] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, New Jersey, 1998.

[23] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algo- rithms. In *4th International Workshop on Source Code Analysis and Manipulation (SCAM 04)*. pages 65–74. Los Alamitos, California, USA. 2004. IEEE Computer Society Press.

[24] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transforma- tion. *Software and System Modeling*, 6(3). pages 269–285. 2007.

[25] O. Seng, J. Stammel, D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. ACM Press. pages 1906–1916. Seattle, Washington, USA. 2006.