Incremental Refactoring Using Seeds

Gabriela Czibula, István Gergely Czibula

Babeş-Bolyai University, 1, M. Kogălniceanu Street, Cluj-Napoca, 400084, Romania,
gabis@cs.ubbcluj.ro; istvanc@cs.ubbcluj.ro

Abstract: *Refactoring* is one major issue to improve the design of software systems, increasing the internal software quality. It is a disciplined technique for improving the structure of existing code without changing its observable behaviour. We have previously introduced a clustering based approach for identifying refactorings in an object oriented software system. Essentially, it takes the existing software system and restructure it using a *k-means* based clustering algorithm (*kRED*), in order to obtain a better design. But, in time, the software system evolves and new application classes are added for implementing new functional requirements. We propose in this paper a k-means based incremental clustering method, Incremental Refactoring Using Seeds (*IRUS*), that is capable to re-partition the existing software system, when new application classes are added to it. The method starts from the clusters obtained by applying *kRED* before the software system's extension. The result is reached more efficiently than running *kRED* again from the scratch on the extended software system. An experimental evaluation proving the method's efficiency is also reported.

Keywords: Software engineering, incremental refactoring, clustering.

1. Introduction

The need for continuing adaptation and evolution is intrinsic to any software application. It is due to the fact that software systems, during their life cycle, are faced with new requirements. These new requirements imply updates in the software systems structure, which have to be done quickly, due to tight schedules which appear in real life software development process. Evolution is achieved in a feedback driven and controlled maintenance process. If the consequent pressure for evolution to adapt to the new situation is resisted, the degree of satisfaction provided by the system in execution declines with time [13].

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 [7] 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].

We have developed in [14] 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 algorithm, kRED (k-means for REfactorings Determination), was developed. The proposed approach can be used to automatically identify refactorings that would improve the software system's internal structure.

Real applications evolve in time, and new application classes are added in order to met new requirements. Obviously, for restructuring the extended software system, kRED clustering algorithm can be applied over and over again, by reassembling the entire extended system, every time when the application classes set change. But this process can be inefficient, particularly for large software systems. What we want is to

extend the approach from [4] and to propose a k-means based incremental clustering method, named Incremental Refactoring Using Seeds (IRUS), that is capable to efficiently re-partition a software system, when a new application class is added to it. The method starts from the partition obtained by applying kRED algorithm before the class extension. The result is reached more efficiently than running kRED again from the scratch on the extended system.

The rest of the paper is structured as follows. Section 2 briefly presents the main aspects related to our previous approach for clustering based refactorings identification [4]. In Section 3 we motivate our work by illustrating the need for incremental refactoring. An incremental clustering approach for adaptive refactorings identification is introduced in Section 4. For the incremental process, an Incremental Refactoring Using Seeds algorithm (IRUS) is proposed. Section 5 indicates several existing approaches in the direction of automatic refactorings identification. An example illustrating how our approach works is provided in Section 6. Some conclusions of the paper and further research directions are outlined in Section 7.

2. *CARD* Approach. Background.

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. The inferring process is carried out with respect to a set of relevant characteristics or features of the analyzed objects. The resulting groups are to be built so that objects within a cluster to have high similarity with each other and low similarity with objects in other groups. Similarity and dissimilarity between objects are calculated using metric or semi-metric functions, applied to the features values characterizing the objects.

A large collection of clustering algorithms is available in the literature. [9] and [12] contain comprehensive overviews of existing techniques.

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 partitions. The partitioning process is iterative and heuristic; it stops when a "good" partitioning is achieved. A partitioning is "good", as we said, when the intra-cluster similarities are high and inter-cluster similarities are low.

We have introduced in [4] a clustering approach CARD for identifying refactorings that would improve the class structure of a software system. CARD consists of three steps:

1. **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.
2. **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.
3. **Refactorings extraction**. The newly obtained software structure is compared with the original one in order to provide a list of refactorings which transform the original structure into an improved one.

In the following we will briefly review our approach from [4] for clustering based refactorings identification.

At the Grouping step we have proposed to re-group entities from the 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).

The objects to be clustered are the elements from the considered software system, i.e. $S = \{e_1, e_2 ..., e_n\}$, where $e_i, 1 \leq i \leq n,$ can be an application class, a method from a class or an attribute from a class. An element $e_i \in S$ is considered to be an entity.

The feature set characterizing the entities is considered to be 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 is the number l of application classes from S.

For a given entity from the software system S, the dissimilarity degree between the entity and the application classes from S is considered. So, each entity $e_i, 1\le i\le n,$ from the software system is characterized by a l-dimensional vector: $(e_{i1},e_{i2},...,e_{il})$, where e_{ij} $(\forall j,1\le j\le l)$ is computed as follows [4]:

$$e_{ij}=\begin{cases}1-\dfrac{|p(e_i)\cap p(C_j)|}{|p(e_i)\cup p(C_j)|} & if\ p(e_i)\cap p(C_j)\neq\phi\\ \infty & otherwise\end{cases} \quad (1)$$

where, for a given entity $e\in S$, $p(e)$ defines a set of relevant properties of e [4].

As in a vector space model based clustering [11], the distance between two entities e_i and e_j from the software system S is expressed as a measure of dissimilarity between their corresponding vectors. In [4], the Euclidian distance is used for discriminating the l-dimensional entities from the software system.

The main idea of the kRED algorithm that is applied in order to re-group entities from a software system is the following. The number of clusters is considered the number l of application classes and the initial centroids are chosen as the application classes from S. Then, as in the classical k-means approach, the clusters (centroids) are recalculated, i.e., each object is assigned to the closest cluster (centroid) until convergence is achieved.

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

3. Motivation

Let us consider a software system S. As presented in Section 2, *kRED* algorithm provides a restructuring model that gives the refactorings needed in S in order to improve its design.

During the evolution and maintenance of S, new application classes are added to it in order to met new functional requirements. Let us denote by S' the software system S after extension. Consequently, restructuring of S' is needed 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 S'. However, this process can be inefficient, particularly for large software systems.

That is why we extend the approach from [4] and we propose an incremental method to cope with the evolving application classes set. Namely, we handle here the case when a new application class is added to the software system and the current restructuring model must be accordingly adapted. The main idea is that instead of applying *kRED* algorithm from scratch on the modified system S', we adjust the partition obtained by *kRED* algorithm for the initial system S, considering the newly added application class. We aim, this way, at reducing the time needed for obtaining the results, without altering the accuracy of the restructuring process.

4. Our Proposal for Incremental Refactoring

In this section we introduce our approach for incremental refactoring, starting from the approach introduced in [4].

4.1 Formal problem study

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

Let $K=\{K_1,K_2,...,K_l\}$ be the partition (set of clusters) discovered by applying *kRED* algorithm on the software system S. Each cluster from the partition is a set of entities, $K_j=\{e_1^j,e_2^j,...,e_{n_j}^j\}, 1\le j\le l.$ The centroid

(cluster mean) of the cluster K_j is denoted by f_j, where $f_j = \left(\frac{\sum_{k=1}^{n_j} e_{k1}^j}{n_j}, ..., \frac{\sum_{k=1}^{n_j} e_{kl}^j}{n_j} \right)$.

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

Let now consider that the software system S is extended with a new application class, C_{l+1}.

Let us denote by S' the extended system. Consequently, the feature set characterizing the entities from S' will be extended with a new feature, corresponding to the newly added application class. After extension, the modified software system becomes $S' = \{e_1', e_2', ..., e_n', e_{n+1}', .., e_{n+m}'\}$, where

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

We mention than each entity from S' is characterized by a l+1 dimensional vector, i.e. $e_i' = (e_{i1}, e_{i2}, ..., e_{il}, e_{i,l+1})$, where e_{ij} is computed as illustrated in (1), $\forall i, 1 \le i \le n+m$.

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

The partition K' of the extended software system S' corresponds to its improved structure. Following the idea from [4], the number of clusters from K' should be the number of application classes from S', i.e. l+1.

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

$$d(e_i^j, f_j) \le d(e_i^j, f_r), \forall j, r, 1 \le j, r \le l, r \ne j \quad (2)$$

We denote by $K_j', 1 \le j \le l$, the set containing the same entities as K_j, after the extension. By $f_j', 1 \le j \le l$, we denote the mean (center) of the set K_j'. These sets $K_j', 1 \le j \le l$, will not necessarily represent clusters after the feature set extension. The newly arrived feature (application class) 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 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.

Besides the clusters $K_1', K_2', ..., K_l'$, the partition K' should also contain a cluster corresponding to the newly added application class C_{l+1}. The initial centroid of this cluster is considered to be the application class itself, i.e. $f_{l+1}' = C_{l+1}$.

We therefore take as starting point the previous partitioning K into clusters (as explained above) and study in which conditions an extended entity $e_i^{j'}$ is still correctly placed into its cluster K_j'.

First, we will introduce in Definition 1 some auxiliary definitions.

Definition 1.

Assuming the theoretical model described above, let us consider a cluster $K_j, 1 \le j \le l$, from the partition *K* of the software system *S*.

a. We denote by OA_j and we call it the set of over average extended entities, the set of extended (l+1 dimensional) entities

$e_i^{j'}$ from cluster $K_j^{'}$ having the value for the l+1-th feature greater than the average value, i.e. as given in Equation (3).

$$OA_j = \left\{ e_i^{j'} \mid 1 \le i \le n_j, e_{i,l+1}^{j} \ge \frac{\sum_{k=1}^{n_j} e_{k,l+1}^{j}}{n_j} \right\} \quad (3)$$

b. We denote by $max_j \in [1, n_j]$ the index of the extended (l+1 dimensional) entity from cluster K_j having the larger value for the l+1-th feature, i.e. satisfying Equation (4).

$$max_j = \arg\max_{i \in [1, n_j]} \{e_{i,l+1}^{j}\} \quad (4)$$

The entity $e_{max_j}^{j'}$ is called the *highest* extended entity from the cluster $K_j^{'}$.

Based on Definition 1, it can be easily proved that the *highest* extended entity from a cluster K_j ($\forall j, 1 \le j \le l$) belongs to the set of *over average* extended entities from $K_j^{'}$, i.e Lemma 1 holds.

Lemma 1. Let us consider a cluster K_j, $1 \le j \le l$, from the partition K of the software system S. Within cluster $K_j^{'}$ relation (5) holds.

$$e_{max_j}^{j'} \in OA_j \quad (5)$$

Proof

We prove below this statement.

Based on Definition 1, we have inequalities:

$$e_{max_j,l+1}^{j} \ge e_{k,l+1}^{j} \quad \forall k, 1 \le k \le n_j \quad (6)$$

By summing inequalities (6) for all entities from cluster K_j we obtain:

$$e_{max_j,l+1}^{j} \ge \frac{\sum_{k=1}^{n_j} e_{k,l+1}^{j}}{n_j} \quad (7)$$

Consequently, from (7) and based on Definition 1, we can conclude that $e_{max_j}^{j'} \in OA_j$ and Lemma 1 is proven.

In Theorem 1 we give sufficient conditions for an extended entity $e_i^{j'}$ in order to be correctly placed in its cluster $K_j^{'}$.

Theorem 1.

If an extended entity $e_i^{j'} \in OA_j$ from the set of *over average* extended entities from a cluster $K_j^{'}$ satisfies inequality (8)

$$d(e_i^{j'}, f_j^{'}) \le d(e_i^{j'}, C_{l+1}) \quad (8)$$

then the entity $e_i^{j'}$ is closer to the center $f_j^{'}$ than to any other center $f_r^{'}, 1 \le j, r \le l+1, r \ne j$, i.e it is correctly placed in cluster $K_j^{'}$.

Proof

We prove below this statement.

First, we will prove that the entity $e_i^{j'}$ is closer to the center $f_j^{'}$ than to any other center $f_r^{'}, 1 \le r \le l, r \ne j$.

$$d^2(e_i^{j'}, f_j^{'}) - d^2(e_i^{j'}, f_r^{'}) =$$

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

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

Using the inequality (2), we have:

$$d^2(e_i^{j'}, f_j^{'}) - d^2(e_i^{j'}, f_r^{'}) \le$$

$$\left(\frac{\sum_{k=1}^{n_j} e_{k,l+1}^{j}}{n_j} - e_{i,l+1}^{j} \right)^2 - \left(\frac{\sum_{k=1}^{n_r} e_{k,l+1}^{r}}{n_r} - e_{i,l+1}^{j} \right)^2$$

$\Leftrightarrow$

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

$$\left(\frac{\sum_{k=1}^{n_j} e_{k,l+1}^{j}}{n_j} - \frac{\sum_{k=1}^{n_r} e_{k,l+1}^{r}}{n_r} \right) \cdot$$

$$\left(\frac{\sum_{k=1}^{n_j} e_{k,l+1}^{j}}{n_j} + \frac{\sum_{k=1}^{n_r} e_{k,l+1}^{r}}{n_r} - 2 \cdot e_{i,l+1}^{j} \right)$$

As $e_i^{j'} \in OA_j$, based on Equation (3), the inequality above becomes:

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

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

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

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

$$d(e_i^{j'}, f_j^{'}) \leq d(e_i^{j'}, f_r^{'}), \forall r, 1 \leq r \leq l, r \neq j \quad (9)$$

So, we have proved that the entity $e_i^{j'}$ is closer to the center $f_j^{'}$ than to any other center $f_r^{'}, 1 \leq r \leq l, r \neq j$.

It is obvious that inequality (8) indicates that the entity $e_i^{j'}$ is closer to the center $f_j^{'}$ than to the center $f_{l+1}^{'}$.

Consequently, from (8) and (9), we can conclude that the entity $e_i^{j'}$ is closer to the center $f_j^{'}$ than to any other center $f_r^{'}, 1 \leq r \leq l+1, r \neq j$, and this ends the proof of Theorem 1.

Remark 1.

We have to remark the following:

Inequality in (3) imposes only intra-cluster conditions. An entity is compared only against its own cluster in order to decide its new affiliation to that cluster.

Theorem 1 gives only sufficient (not necessary) conditions for an extended entity $e_i^{j'}$ in order to be correctly placed in its cluster $K_j^{'}$.

When the software system S is extended with a new application class, it is very likely that inequality (8) holds, as it is very likely that the entities from S are closer to other entities from S than to the newly added application class C_{l+1}.

As indicated in Lemma 1, $e_{max_j}^{j'} \in OA_j$. Consequently, considering the remark above, it is very likely that at least one extended entity from cluster $K_j^{'}$, i.e. $e_{max_j}^{j'}$, is correctly placed in its cluster $K_j^{'}$.

4.2 The *incremental refactoring using seeds* algorithm

We will use the property enounced in Theorem 1 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 (feature set) extension.

In our view, these entities form the *center* of their cluster. We will use these *cluster centers* as seed for clustering.

As we have mentioned in Remark 1, it is very likely that entity $e_{max_j}^{j'}$ satisfies inequality (8), and, consequently, is correctly placed in its cluster $K_j^{'}$. That is why we will consider that the *center* of cluster $K_j^{'}$ contains at least the *highest* extended entity $e_{max_j}^{j'}$ from it. It is also obvious that the extended entities from $K_j^{'}$ that are closer to the extended centroid $f_j^{'}$ of cluster j than to any other extended centroid $f_r^{'}, 1 \leq r \leq l+1, r \neq j$ have also to belong to the *center* of cluster $K_j^{'}$.

Definition 2.

We denote by $Center_j = e_{max_j}^{j'} \bigcup \{e_i^{j'} \mid e_i^{j'} \in K_j^{'}, d(e_i^{j'}, f_j^{'}) \leq d(e_i^{j'}, f_r^{'}), 1 \leq r \leq l+1, r \neq j\}$. We denote by

CENTERS the set $\{Center_j, 1 \le j \le l\}$ of all clusters centers.

We mention that the initial number of centroids (clusters) in the incremental clustering algorithm is the number of application classes after the extension of S, i.e. l+1.

The *cluster centers*, chosen as we described above, will serve as seeds in the incremental clustering process. All entities in $Center_j$ will surely remain together in the same group if clusters do not change. This will not be the case for all center entities, but for most of them.

We have presented above the idea for choosing the initial l centroids and clusters. Considering that a new application classes is added to the software system S, the l+1-th centroid is chosen as the newly added class, i.e. $f'_{l+1} = C_{l+1}$.

The incremental algorithm starts by calculating the old clusters' centers. These centers will be the new initial clusters from which the incremental process begins. Next, the algorithm proceeds in the same manner as *kRED* algorithm [4] does.

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

We give next the Incremental Refactoring Using Seeds algorithm.

Algorithm *IRUS* is

{**Input**:
- the software system $S = \{e_1, e_2 ..., e_n\}$ of l-dimensional entities
- l, the number of classes from S
- the newly added class, C_{l+1}
- the extended software system $S' = \{e'_1, e'_2, .., e'_n, e'_{n+1}, .., e'_{n+m}\}$ of l+1-dimensional extended entities
- the metric d between entities
- *noMaxIter* the maximum allowed number of iterations.
- $K = \{K_1, K_2, ..., K_l\}$ the partition of entities in S reported by *kRED*.

Output:
- the re-partitioning of the entities from S', $K' = \{K'_1, K'_2, ..., K'_{l+1}\}$ in S' }.

Begin
// The old cluster centers are computed
For $j \leftarrow 1, l$ do
@ Compute $Center_j$ as in Definition 2
// The centroid j is determined
$f'_j \leftarrow$ the mean of objects in $Center_j$
EndFor
// The centroid corresponding to the newly
// added application class is computed
$f'_{l+1} = C_{l+1}$
// The incremental process starts with the initial
// centers
$K' = \{f'_1, f'_2, ..., f'_{l+1}\}$
While (K' changes) and (there were not performed *noMaxIter* iterations) do
For $j \leftarrow 1, l+1$ do
// The clusters are recalculated
$K'_j = Center_j \bigcup \{e_i^{j'} \mid e_i^{j'} \in Center_j$
$and\ \forall f'_k, d(e_i^{j'}, f'_j) \le d(e_i^{j'}, f'_k)\}$
EndFor
For $j \leftarrow 1, l+1$ do
If $K'_j = \phi$ then
@ remove element K'_j from K'
// the no. of clusters is decreased
Else
$f'_j \leftarrow$ the mean of K'_j
Endif
EndFor
EndWhile
@ K' is the output partition
End.

Remark 2.

We mention that the time complexity for calculating the centers in the incremental clustering process does not grow the global complexity of *IRUS* algorithm.

5. Literature Review

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

There are a lot of approaches in the literature in the field of *software clustering* which deal with the software decomposition problem.

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 [19], technique that was added to the low-coupling and high-cohesion heuristics in order to capture patterns that appear commonly in software systems. In [20], a partition of a software system is refined by identifying components that belong to the wrong subsystem, and by placing them in the correct one. The paper describes a program that attempts to reverse engineer software in order to better provide 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 [19] and [20] were not tested on large software systems, they were promising.

Mancoridis et al. introduce in [15] 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. In [16], some extensions of *Bunch* are presented, 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 [19]; deducing clusters from file and procedure names [1]; utilizing the connectivity between software objects [3, 10, 18]; or looking at the problem at hand as an optimization problem [15].

Another approach for software clustering was presented in [1] 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 [25] 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. In [26], Tzerpos and Holt introduce 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.

There were various approaches in the literature in the field of *refactoring*, also. But, only very limited support exists in the literature for detecting refactorings.

Deursen et al. have approached the problem of *refactoring* in [28]. 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 [29] 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 [21]. The authors use an evolutionary algorithm for identifying refactorings that improve the system structure. A weighted multi-objective search is applied, 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.

An approach for restructuring programs written in Java starting from a catalog of bad smells is introduced in [6].

Based on some elementary metrics, the approach in [24] aids the user in deciding what kind of refactoring should be applied.

The paper [22] 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 [30]. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [14] 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 [4].

Fatiregun et al. [5] 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. Mens et al. [23] 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 incremental refactoring, as it is approached in this paper.

6. Experimental Evaluation

For experimentally validating *IRUS* algorithm, we will consider two case studies: a simple Java code example [22] and the open source case study JHotDraw.

As a quality measure for the incremental process, we take the movement degree of the entities from the clusters centers. More stable they are, better was the decision to choose them as centers for the incremental clustering process. Assuming that K' is the output partition provided by *IRUS* algorithm, we express the *centers stability* as:

$$CS(CENTERS) = \frac{1}{l}\sum_{i=1}^{l} cs(Center_i, K') \quad (10)$$

In Equation (10), $cs(Center_i, K') = \frac{\sum_{j \in M_{Center_i}} \frac{|Center_i \cap K_j|}{|Center_i \cup K_j|}}{|M_{Center_i}|}$ (where $M_{Center_i} = \{j \mid 1 \le j \le l, Center_i \cap K_j \neq \phi\}$) is the set of clusters from K' that contain elements from the center $Center_i$) is the *stability* of center $Center_i$.

In our view, *CS* defines the stability of *clusters centers* after the incremental clustering process. For a given cluster center $Center_i \in CENTERS$, $cs(Center_i, K')$ defines the degree to which all entities from $Center_i$ were discovered in a single cluster.

Based on (10), it can be easily proved that $CS(CENTERS) \in [0,1]$. $CS(CENTERS)$ is equal to 1 iff $cs(Center_i, K') = 1, \forall i = 1,...,l$, i.e. each cluster center was discovered in a single cluster. In all other situations, $CS(CENTERS) < 1$.

The worst case is when each entity in $Center_i$ ends in a different final cluster, and this happens for every center from $CENTERS$. The best case is when every $Center_i$ remains compact and it is found in a single final cluster. It can be simply proved that the higher the value of $CS(CENTERS)$ is, better was the centers choice, i.e. we aim at maximizing *CS* measure.

6.1 Java code example

In this section we present an experimental evaluation of *IRUS* algorithm on a simple case study. We aim to provide the reader with an easy to follow example of incremental refactorings extraction. Let us consider the software system S consisting of the Java code example shown below. The code example below is taken from [22].

```
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 $K = \{K_1, K_2\}$ 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 now consider that the system S is extended with another class, **Class_C**. Let us denote by 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 **Class_C**, 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 $K' = \{K_1', K_2', K_3'\}$ 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 K' of the extended system 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 IRUS algorithm, the partition K obtained after applying kRED algorithm before the system's extension.

We comparatively present in Table 1 the results obtained after applying *kRED* and *IRUS* algorithms for restructuring the extended system S'. We mention that both algorithms have identified the partition K' corresponding to the improved structure of S'.

Table 1. The results

No *l* of classes from S	2
No of entities from S	12
No of classes from S'	3
No of entities from S'	18
No of *kRED* iterations for *l*+1 application classes	3
No of *IRUS* iterations for *l*+1 application classes	2
CS(CENTERS)	1

From Table 1 we observe that *IRUS* algorithm finds the solution in a smaller number of iterations than *kRED* algorithm. This confirms that the time needed by *IRUS* to obtain the results is reduced, and this leads to an increased efficiency of the incremental process. For larger software systems, it is very likely that the number of iterations performed by *IRUS* will be significantly reduced in comparison with the number of iterations performed by *kRED*. We also notice that the *centers stability* is 1, and this leads to the conclusion that the choice of *clusters centers* was very good.

6.2 JHotDraw case study

Our second evaluation is JHotDraw case study [8]. 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.

Let us consider JHotDraw system from which we have removed one application class: **StorableInput**. We denote the resulting system by S. Therefore, S consists of 172 application classes, i.e *l*=172. After applying *kRED* algorithm on S we have obtained a partition K in which there were 2 misplaced methods and 1 misplaced attribute. 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.

Let now extend S with the application class that was initially removed from JHotDraw, **StorableInput**. We denote by S' the extended software system, which, in fact, is the entire JHotDraw system. Consequently, the number of application classes from S' is 173.

There are two possibilities to obtain the restructured partition K' of the extended system S' (JHotDraw).

Table 2. The misplaced entities on S

Entity	Type	**Target class**
PertFigure.writeTasks	Method	**StorableOutput**
PolygonFigure.distanceFromLine	Method	**Geom**
ColorEntry.fName	**Attribute**	**ColorMap**

Table 3. The misplaced entities on S'

Entity	Type	**Target class**
PertFigure.writeTasks	Method	**StorableOutput**
PertFigure.readTasks	Method	**StorableInput**
PolygonFigure.distanceFromLine	Method	**Geom**
StandardDrawingView.drawingInvalidated	Method	**DrawingChangeEvent**
ColorEntry.fName	Attribute	**ColorMap**
ColorEntry.fColor	**Attribute**	**ColorMap**

A. To apply kRED algorithm from scratch on the extended JHotDraw system.

B. To adapt, using *IRUS* algorithm, the partition K obtained after applying *kRED* algorithm before the extension, i.e on S.

In the following we will detail variants **A** and **B**.

A. After applying *kRED* algorithm for JHotDraw case study (S'), we have obtained a partition K' which contains 4 misplaced methods and 2 misplaced attributes [4]. The names of the elements (methods, attributes) that were proposed to be moved is shown in the first column of Table 3. The suggested target class is shown in the second column.

B. We have adjusted, using *IRUS* algorithm, the partition K obtained after applying *kRED* algorithm before the system's extension. The partition K' obtained this way coincides with the one reported by applying *kRED* algorithm on JHotDraw, i.e it contains 4 missplaced methods and 2 missplaced attributes as shown in Table 3.

From our perspective, all the proposed refactorings can be justified. Consider, for example, the **PertFigure.writeTasks** method presented below [8].

```
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 responsibility 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 uses 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.distanceFrom Line** to **Geom** class is justifiable.

In the partition K' there are two misplaced attributes: **ColorEntry.fName** and **ColorEntry.fColor** which are placed in **ColorMap**. From our perspective, these refactoring can 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.

We comparatively present in Table 4 the results obtained after applying *kRED* and *IRUS* algorithms for restructuring the extended system S'.

Table 4. Comparative results on JHotDraw case study

Quality measure	*kRED* for 173 classes	*IRUS* for 173 classes
No. of iterations	6	4
CS	-	0.9021

From Table 4 we observe the following:

- IRUS algorithm finds the solution in a smaller number of iterations than kRED algorithm. This confirms that the time

needed by IRUS to obtain the results is reduced, and this leads to an increased efficiency of the incremental process.

- The accuracy of the results provided by IRUS are preserved (the additional refactorings identified by IRUS were justified above).
- The choice of the clusters centers in the incremental process was good enough (the centers stability is close to 1).

7. Conclusions and Future Work

We have proposed in this paper an incremental method for adjusting a restructuring model of a software system when a new application class is added to it. The considered experiments prove that the result is reached more efficiently by using *IRUS* method rather than running *kRED* again from the scratch on the extended software system.

Further work will be done in order to isolate conditions to decide when it is more effective to adapt (using *IRUS*) the partitioning of the extended software system than to recalculate it from scratch using *kRED* algorithm. We aim at extending the experimental evaluation of the proposed approach by applying *IRUS* algorithm on other real software system. We will also study the appropriateness of adaptive fuzzy [17, 27] clustering algorithms for refactorings identification and also incremental extensions of these methods.

Acknowledgements

This work was supported by CNCSIS - UEFISCSU, project number PNII - IDEI 2286/2008.

REFERENCES

1. ANQUETIL, N., T. LETHBRIDGE, **Extracting Concepts from File Names; a New File Clustering Criterion**, In 20th Intl. Conf. Software Engineering, 1998, pp. 84-93.
2. BROWN, W. J., R. C. MALVEAU, III H. W. MCCORMICK, T. J. MOWBRAY, **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**, John Wiley & Sons, Inc., New York, NY, USA, 1998.
3. CHOI, S. C., W. SCACCHI, **Extracting and Restructuring the Design of Large Systems**, IEEE Softw., 7(1), 1990, pp. 66-71.
4. CZIBULA, I.G., G. SERBAN, **Improving Systems Design using a Clustering Approach**, Intl. Journal of Computer Science and Network Security (IJCSNS)}, 6(12), 2006, pp. 40-49.
5. HARMAN, M., D. FATIREGUN, R. HIERONS, **Evolving Transformation Sequences using Genetic Algorithms**, In Proc. of the 4th Intl. Workshop on Source Code Analysis and Manipulation (SCAM 04), Los Alamitos, California, USA, 2004, IEEE Computer Society, pp. 65-74.
6. DUDZIKAN, T., J. WLODKA, **Tool-supported Dicovery and Refactoring of Structural Weakness**, Master's thesis, TU Berlin, Germany, 2002.
7. FOWLER, M., **Refactoring: Improving the Design of Existing Code**, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
8. GAMMA, E., **JHotDraw Project**, http://sourceforge.net/projects/jhotdraw.
9. HAN, J., **Data Mining: Concepts and Techniques**, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
10. HUTCHENS, D. H., V. R. BASILI, **System Structure Analysis: Clustering with Data Bindings**, IEEE Trans. Softw. Eng., 11(8), 1985, pp. 749-757.
11. JAIN, A. K., M. N. MURTY, P. J. FLYNN, **Data Clustering: a Review**, ACM Computing Surveys, 31(3), 1999, pp. 264-323.
12. JAIN, A. K., R. C. DUBES, **Algorithms for Clustering Data**, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
13. LEHMAN, M. M., **Laws of Software Evolution Revisited**, In LNCS 1149 - EWSPT96, Springer Verlag, 1997, pp. 108-124.
14. LUNG, C.-H., **Software Architecture Recovery and Restructuring through Clustering Techniques**, In Proc. of the 3rd Intl. Workshop on Software

Architecture (ISAW '98), New York, NY, USA, 1998, ACM Press, pp. 101-104.

15. MANCORIDIS, S., B. S. MITCHELL, C. RORRES, Y. CHEN, E. R. GANSNER, **Using Automatic Clustering to Produce High-level System Organizations of Source Code**, In IEEE Proc. of the 1998 Intl. Workshop on Program Understanding ({IWPC}'98), Piscataway, NY, 1998, IEEE Press, p. 45.
16. MANCORIDIS, S., B. S. MITCHELL, Y.-F. CHEN, E. R. GANSNER, **Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures**, In ICSM, 1999, pp. 50-59.
17. MOISE, G., **Applying Fuzzy Control in the Online Learning Systems**, Studies in Informatics and Control, 18(2), 2009, p. 165.
18. NEIGHBORS, J. M., **Finding Reusable Software Components in Large Systems**, In Working Conference on Reverse Engineering, 1996, pp. 2-10.
19. SCHWANKE, R. W., M. A. PLATOFF, **Cross References are Features**, In Proc. of the 2nd Intl. Workshop on Software configuration management, New York, NY, USA, 1989, ACM Press, pp. 86-95.
20. SCHWANKE, R. W., **An Intelligent Tool for Re-engineering Software Modularity**, In ICSE '91: Proc. of the 13th Intl. Conf. on Software Engineering, Los Alamitos, CA, USA, 1991, IEEE Computer Society Press, pp. 83-92.
21. SENG, O., J. STAMMEL, D, BURKHART, **Search-based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems**, In Proc. of the 8th Ann. Conf. on Genetic and Evolutionary Computation (GECCO '06), New York, NY, USA, 2006, ACM Press, pp. 1909-1916.
22. SIMON, F., F. STEINBRUCKNER, CLAUS LEWERENTZ, **Metrics Based Refactoring**, In Proc. of the 5th European Conf. on Software Maintenance and Reengineering (CSMR '01), Washington, DC, USA, 2001, IEEE Computer Society, pp. 30-38.
23. TAENTZER, G., T. MENS, O. RUNGE, **Analysing Refactoring Dependencies using Graph Transformation**, Software and System Modeling, 6(3), 2007, pp. 269-285.
24. TAHVILDARI, L., K. KONTOGIANNIS, **A Metric-based Approach to Enhance Design Quality Through Meta-pattern Transformations**, In Proc. of the 7th European Conf. on Software Maintenance and Reengineering (CSMR '03), Washington, DC, USA, 2003, IEEE Computer Society, pp. 183-192.
25. TZERPOS, V., R. C. HOLT, **Mojo: A Distance Metric for Software Clusterings**, In Working Conf. on Reverse Engineering, 1999, pp. 187-193.
26. TZERPOS, V., R. C. HOLT, **ACDC: An Algorithm for Comprehension-driven Clustering**, In Working Conf. on Reverse Engineering, 2000, pp. 258-267.
27. SANKARANARAYANASAMY, K., V. DHANALAKSHMI, S. ARUNACHALAM, T. PAGE, **A Fuzzy Analysis Approach to Part Family Formation in Cellular** M**anufacturing Systems**, Studies in Informatics and Control, 17(4), 2008, p. 433.
28. VAN DEURSEN, A., L. MOONEN, A. VAN DEN BERGH, G. KOK, **Refactoring test code**, 2001, pp. 92-95.
29. XING, Z., E. STROULIA, **Refactoring Detection Based on UMLdiff Change-facts Queries**, WCRE, 2006, pp. 263-274.
30. XU, X., C.-H. LUNG, M. ZAMAN, A. SRINIVASAN, **Program Restructuring through Clustering Techniques**, In Proc. of the 4th IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM'04), Washington, DC, USA, 2004, IEEE Computer Society, pp. 75-84.