Adaptive Refactoring Using a Core-Based Clustering Approach

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

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

Abstract: Improving the software systems design through refactoring is the most important issue 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. We have previously intoduced an original approach, named *CARD* (*Clustering Approach for Refactorings Determination*) that uses clustering for improving the class structure of a software system. In this paper we extend our approach and 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. We provide an example illustrating the efficiency of the proposed approach.

Key–Words: Restructuring, 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. 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 [3] is the process of improving the design of software systems, aiming 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.

We have developed in [1] 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 approach previously introduced in [1].

The rest of the paper is structured as follows. The clustering based approach for adaptive refactorings identification previously introduced in [1] is described in Section 2. For the adaptive process, a *Core Based Adaptive Refactoring* algorithm (*CBAR*) is proposed. Section 3 indicates several existing approaches in the direction of automatic refactorings identification. An example illustrating how our approach works is provided in Section 4. Some conclusions of the paper and further research directions are outlined in Section 5.

2 Our approach

2.1 *kRED* Clustering Algorithm for Refactorings Identification. Background

In the following we briefly describe *CARD* approach that we have introduced in [1] for identifying refactorings that would improve the class structure of a software system. First, the existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing re-

lationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system. After data was collected, the set of entities extracted at the previous step are regrouped in clusters using a clustering algorithm. The goal of this step is to obtain an improved structure of the existing software system. The last step is to extract the refactorings which transform the original structure into an improved one, by comparing the newly obtained software structure with the original one.

For re-grouping entities from the software system, a vector space model based *clustering* algorithm, named *kRED* (*k-means for REfactorings Determination*), was introduced in [1]. In the proposed 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 $e_i \in S$ as an *entity*. Using a vector space model based clustering [5], we have considered the attribute set characterizing the entities as the set of application classes from the software system S, $\mathcal{A} = \{C_1, C_2, \ldots, C_l\}$. For each entity e_i from the software system S, a l dimensional vector was defined, $e_i = (e_{i1}, e_{i2}, \ldots, e_{il})$ [1], where e_{ij} expresses the dissimilarity degree between the entity e_i and the application class C_j. Finally, the *distance* $d(e_i, e_j)$ between two entities e_i and e_j from the software system $\mathcal{S}$ was computed as a measure of dissimilarity between their corresponding vectors, using the *Euclidian distance*.

The main idea of the *kRED* algorithm introduced in [1] 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.

More details about $kRED$ algorithm can be found in [1].

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

2.2 Our proposal for adaptive refactoring

Let us consider a software system $\mathcal{S}$. As presented in Section 2.1, 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 [1] 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 [1].

2.3 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 [1], 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.

$$d(e_i^j, f_j) \leq d(e_i^j, f_r), \forall j, r,\ 1 \leq j, r \leq l,\ r \neq j. \quad (1)$$

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 $e_i^{j'}$ is still correctly placed into its cluster K'_j. For that, we express the distance of $e_i^{j'}$ 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 (2) and (3) hold for an extended entity $e_i^{j'}$ *($1 \leq j \leq l$)*

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

and

$$d(e_i^{j'}, f'_j) \leq d(e_i^{j'}, C_v), \forall\, v \in \{l+1, l+2, \ldots, l+s\} \quad (3)$$

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

From lack of space, we will not give the proof of Theorem 1. We have to notice that the inequality in (2) imposes only intra-cluster conditions. An entity is compared against its own cluster in order to decide its new affiliation to that cluster.

2.4 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* $StrongCore_j = \{e_i^{j'} | e_i^{j'} \in K'_j, e_i^{j'}$ *satisfies the set of inequalities (2) and (3)*$\}$ *the set of all objects in* K'_j *satisfying inequalities (2) and (3) 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 (2) and (3).*
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 (2) and (3) for at least the average number of attributes for which (2) and (3) 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 (2) 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 (2) (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 (2) and (3) 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 $\mathcal{S}$, 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 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.

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).*

3 Related Work

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

Deursen et al. have approached the problem of *refactoring* in [10]. 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 [11] 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 [7]. 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 [2].

Based on some elementary metrics, the approach in [9] aids the user in deciding what kind of refactoring should be applied.

The paper [8] 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 [12]. This approach focuses on automated support for identifying ill-structured or low

cohesive functions. The paper [6] 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 [1].

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

4 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.

```
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 $kRED$ and $CBAR$ algorithms for restructuring the extended system $\mathcal{S}'$. We mention that both 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

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 larger 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 $kRED$.

5 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 order 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. We also aim at applying the adaptive algorithm on open source case studies and real software systems.

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] 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.

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

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

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

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

[6] 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.

[7] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.

[8] 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.

[9] 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.

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

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

[12] 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.