Software Refactoring at the Class Level using Clustering Techniques

Abdulaziz Alkhalid

Department of Systems and Computer Engineering
Carleton University
1125, Colonel by Drive, Ottawa, ON, Canada, K1S 5B6
akhalid@sce.carleton.ca

Mohammad Alshayeb and Sabri A. Mahmoud

Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
{alshayeb, smasaad} @kfupm.edu.sa

Software becomes more and more complex as it adapts new requirements, is enhanced or is modified. Thus, the quality of the software decreases. Therefore, there is a need to reduce the software's complexity and improve its quality. Refactoring reduces software complexity and improves quality by restructuring the code into a more readable form that improves its internal structure without changing its external functionality. However, it is a challenging task and requires effort from the software designer. In this paper, we propose a method for identifying ill-structured software at the class level that provides heuristic refactoring advice to software designers in order to create balance between coupling and cohesion using pattern recognition techniques. To identify the ill-structured code we use three clustering techniques, namely, the Single Linkage algorithm (SLINK), the Complete Linkage algorithm (CLINK) and the Weighted Pair-Group Method using Arithmetic averages (WPGMA). In addition to these clustering techniques, we also use the Adaptive K-Nearest Neighbour (A-KNN) algorithm and compare its performance with the other clustering techniques. The results show that software structuring at the class level using A-KNN is superior to SLINK, CLINK and WPGMA in terms of performance and computational complexity.

Keywords: Software refactoring, code restructuring, clustering, coupling, cohesion
ACM Classification: D.2.7

1. INTRODUCTION

Refactoring changes a program's design in a way that does not alter the external behaviour of the code while improving its internal structure (Opdyke, 1992; Fowler, 1999). Refactoring is a challenging and time-consuming task that requires effort to identify and apply (Alshayeb *et al*, July 2001). The processes of refactoring can be automated and supported by tools (Scientific-Toolworks, Accessed: 1 November, 2009; JetBrains, Accessed: 2 November, 2009). However, the challenging task is to identify the code that needs to be refactored. In this paper, we use pattern recognition techniques to identify the code that needs to be refactored at the class level.

Manuscript received: 3 August 2011
Communicating Editor: Colin Fidge

Recognition is the process of identifying a pattern, vaguely defined, that could be given a name, as a member of a category which is already known (Duda *et al*, 2000). It is the act of taking in raw data and taking an action based on the “category” of the pattern (Duda *et al*, 2000). Pattern recognition systems use different models of classification such as Syntactic (structural), Artificial Neural Networks (biologically motivated), Template Matching, Statistical (geometric), and Hybrid approaches (Duda *et al*, 2000).

Coupling is an indication of the interdependencies among software modules. Cohesion is an indication of the relative functional strength of a module (Pressman, 2010). A software design should have low coupling, that is the collaboration between modules should be as low as possible. A software design should also be highly cohesive, that is a module should have a small, focused set of responsibilities (Pressman, 2010).

The objective of this paper is to use pattern recognition techniques to help designers identify how their code can be refactored, at the class level, in order to maximize cohesion and minimize coupling. This balancing process may require moving methods between classes in the system. We consider two approaches for moving methods between classes. The first approach considers moving methods from one class to another while keeping the number of classes in the system unchanged. Thus, the number of classes before and after refactoring remains the same. We call this approach “software refactoring at the class level using clustering with a fixed number of classes”. The second approach considers moving methods among classes with possible changes to the number of classes in the system. Hence, the number of the classes after refactoring might be different from the number of classes before refactoring. We call this approach “software refactoring at the class level using clustering with an adaptive number of classes”.

2. LITERATURE REVIEW

A survey of software refactoring has been presented by Mens and Tourwe (Mens and Tourwe, 2004). They discussed several aspects of software refactoring like refactoring activities, the techniques used by these activities, the entities of software that have been refactored, different aspects of software refactoring tools and the effects of refactoring. Tichelaar *et al* (2000) provided a meta-model for software refactoring and validated their study by a prototype tool. Simon *et al* (2001) presented a way to support making decisions on where to apply refactoring by using some metrics and they described an approach for typical refactoring. Tahvildari and Kontogiannis (2003) presented a framework to detect cases in which transformation in the code could be applied in order to enhance the quality of software. This framework supports code transformation for object-oriented legacy systems. They used a set of metrics for object-oriented software in order to analyze the impact of these possible transformations.

Lung *et al* (2004; 2006) used numerical taxonomy clustering on different phases related to software development like design, reverse engineering, and maintenance. Their approaches include partitioning, restructuring, increasing cohesion and reducing coupling. They used traditional clustering techniques for software restructuring at the function level. They provided a way to partition a function into different functions based on some similarity measurements. However, their work was applied only on the structural code. Anquetil and Lethbridge (2003) presented a mechanism for entities clustering using similarity measures based on shared features. In our own previous work (Alkhalid *et al*, 2010) we proposed an Adaptive K-Nearest Neighbour (AKNN) clustering and used this algorithm to perform refactoring at the function/method level. The approach was able to identify ill-structured functions in software and provided suggestions to enhance cohesion. We applied the approach on a set of experimental units using three traditional clustering techniques (viz. the Single Linkage algorithm (SLINK), the Complete Linkage algorithm (CLINK), and the Weighted Pair-Group Method using Arithmetic averages (WPGMA)), in addition to applying it using AKNN. AKNN was found to provide better refactoring results and required less computational complexity than traditional clustering techniques. In this paper we continue this research at the class level.

3. CLUSTERING

Clustering, within pattern recognition and statistics, belongs to a class of unsupervised learning. It is a method for clustering data keeping most similar patterns in the same cluster and most dissimilar patterns in different clusters (Fix and Hodges, 1951; Kaufman and Rousseeuw, 1990; Rugaber *et al*, 1996; Kin *et al*, 2000; Wattanachon and Lursinsap, 2004). In the literature there are many clustering implementations (Anderberg, 1973; Everitt, 1980; Romesburg, 1990). All of these clustering techniques share a common characteristic of grouping data such that all individuals in each group are similar or at least they are more similar to each other than other individuals in other groups. The groups are called clusters and the individuals are called items, objects or entities (Duda *et al*, 2000).

3.1 Entities and Features

To cluster a set of entities into groups, the features of these entities must be extracted. Based on the feature values, the entities are organized into groups. In this paper, since our focus is on providing helpful suggestions for refactoring of classes, we first need to determine the entities that should be put into clusters. For software refactoring at the class level, methods are chosen as entities. This is because methods are the basic computational elements of classes. Class constructors are considered as regular methods, because the constructor may contain initialization statements to the data-members inside the class.

The features are used to calculate how close two entities are. An entity may have many features. Entities are more similar if they share more common features. Different class data-members may be related to different functional tasks. Therefore, the class data-members are used as features for the entities. Thus, there is a one to one mapping between entities and methods and there is another one to one mapping between features and data-members.

A method which is represented by an entity can access all data-members inside the class; it can also access data-members of other classes by using instances of those classes. Thus, each feature is measured on a quantitative scale representation. The feature value is the number of times the method accesses the data-members represented by the feature. The method can access a data-member directly or indirectly. In the direct manner, the method accesses data-members in the containing class or in other classes. The method can access a data-member indirectly by invoking other methods that use the data-member. Thus, the value of the feature can be measured by using the following formula:

$$v(f_i, p_j) = d_a(f_i, p_j) + i_a(f_i, p_j); \qquad i \in [1..a], j \in [1..b] \tag{1}$$

where

Function $V(f_i, p_j)$ is the value of the feature f_i for the entity p_j.

Function d_a is the number of the direct accesses to the data-member represented by feature f_i in the method represented by entity p_j.

Function i_a is the number of indirect accesses to the data-member represented by feature f_i in the method represented by entity p_j.

Parameter a is the number of the features (or data-members) in the system, and

Parameter b is the number of the entities (or methods) in the system.

$$i_a(f_i + p_j) = \sum_{k=0}^{t} m_k n_k; \quad k \neq j \tag{2}$$

where

t is the number of the other methods that are called inside the method represented by entity p_j.

m_k is the number of the invocations to the method represented by entity p_k inside the method represented by entity p_j, and

$n_k = d_a(f_i, p_k)$: the number of direct accesses to the data-member represented by feature f_j in the method represented by entity p_k.

We used entity-feature matrix in which the rows represent the entities (functions / methods) and columns represent features (data-members) (Lung *et al*, 2006). For each entity there is a representative row in the entity-feature matrix and for each feature (data-member) there is a representative column. The values in the row describe the features of the related entities. For any two entities in the entity-feature matrix, there are three types of matches: n-0/0-n, n-m, and 0-0. The first type means there is no-match between the two entities. This provides a negative contribution in the coefficient resemblance. In other words, this is an indication of dissimilarity between the two entities. This type of match is considered in the resemblance coefficient. Matches of the second type n-m mean that the two entities share at least one feature (m>0 and n>0). This provides a positive contribution to the resemblance coefficient. This type of matches is also considered in the similarity coefficient. The last type (0-0) means a no-match (i.e., no feature is accessed by any of the two entities). Since the columns represent all features in the system, it is expected to have many no-matches (0-0 matches) in the matrix. This is because each method uses a small number of features. Several studies in restructuring showed that better results could be obtained by ignoring 0-0 matches (Anquetil and Lethbridge, 2003; Lung *et al*, 2004; Lung *et al*, 2006). These matches will generate a distortion in the similarity measure. Therefore, these matches are not considered in the similarity coefficient.

3.2 Similarity Measure

In this paper, we extend the similarity measure that was used by other studies for refactoring at the function, architecture and package levels (Lung *et al*, 2004; Lung *et al*, 2006; Alkhalid *et al*, 2010; Alkhalid *et al*, 2011) to be used at the class level. The entity-feature matrix describes the entities of the class; hence we need to find similarity between these entities and then we can define the resemblance coefficient. Previous studies used a specific resemblance coefficient to find similarity between lines of code or similarity between classes (Lung *et al*, 2004; Lung *et al*, 2006; Alkhalid *et al*, 2010). In this paper, we extend the use of this resemblance coefficient to the class level. In fact, the more features two entities share, the more similar they are (Lung *et al*, 2004; Lung *et al*, 2006; Alkhalid *et al*, 2010). If the entities share many features, this indicates that those two entities have closely related functionalities. Thus, they should be kept in the same class in order to maintain high cohesion. Formula (3) presents an adaptive resemblance coefficient to be used in calculating the values of the similarity matrix which describes the similarity between each entity and all other entities.

$$\textit{Coeff}\ \text{(resemblance coefficient)} = \frac{\text{similarity factor}}{\text{similarity factor} + \text{dissimilarity factor}} \tag{3}$$

$$similarity\ factor = \sum_{k=0}^{a} min\ (n_k, m_k) \qquad (4)$$

where
a is the number of n-m matches between two entities.

$$dissimilarity\ factor = \text{number of } n\text{-}0/0\text{-}n \text{ matches between two entities}$$

Therefore, if no common feature is shared between two entities, the Coeff will be 0. If there are no n-0/0-n matches between two entities, the Coeff will be 1. The value of Coeff will be between 0 and 1.

4. DATA CLUSTERING TECHNIQUES

This section describes the three hierarchical agglomerative algorithms used in our study.

4.1 SLINK, CLINK, WPGMA Clustering

The Single Linkage algorithm (SLINK) (Murtagh, 1983; Jain *et al*, 1999), the Complete Linkage algorithm (CLINK) (Defays, 1977; Murtagh, 1983; Jain *et al*, 1999) and the Weighted Pair-Group Method using Arithmetic averages (WPGMA) (Murtagh, 1983; Jain *et al*, 1999) are three hierarchical agglomerative algorithms with different feature weights. SLINK, CLINK, and WPGMA follow the same mechanism by assigning each element to a group (cluster). In the next step, one cluster is merged with the closest cluster; the algorithm merges the two closest clusters. This procedure is repeated until there is only one cluster. However, each algorithm uses a different approach to compute the distance between clusters. SLINK measures the distance between clusters by measuring the distance between the closest pair of elements, taking one element from each cluster. The minimum value of these distances is considered as the distance between the two clusters. CLINK takes the distance between the most distant pair of elements, one from each cluster and then the distance between every possible element pairs is computed and the maximum value of these distances is used. In WPGMA, the distance between the two clusters is taken as the average of the distances between all pairs of elements in the two clusters.

4.2 Adaptive K-Nearest Neighbour Clustering (A-KNN)

In this study, in addition to the use of SLINK, CLINK and WPGMA, we also use A-KNN (Alkhalid *et al*, 2010), a general purpose clustering method that is based on the KNN clustering algorithm, and then we compare its performance with SLINK, CLINK and WPGMA. A-KNN has the advantage of reducing the amount of computation needed. The first step in A-KNN, like most other clustering techniques, is to consider each element as a cluster. A-KNN uses a labeling approach; thus, in the first step each element will be labeled with a unique identifier that represents the cluster identity. A-KNN can work with different values of K. Suppose that we are working with K=3 (A-3NN), then, in the second step, the algorithm selects the three nearest neighbours to the element that will be clustered to check their labels. If at least two out of the three elements have the same label, the algorithm labels the current element with the same label of those two elements. If the three element share different labels, the algorithm labels the current element with the same label of the closest element; in this case it works in the same way as the Nearest Neighbour (NN) algorithm. The algorithm repeats the clustering process until no more changes occur in the clustering tree. Then the algorithm outputs one cluster tree. The advantage of A-KNN in reducing computations is based on the fact that the similarity matrix in A-KNN is calculated only once. Thus, the number of computations is decreased significantly while in the traditional clustering techniques (SLINK, CLINK, and WPGMA) the similarity matrix is calculated in each iteration. Different values of K =1, 3, ..., 9 may be applied. Our experimental results and those of Defays (Defays 1977) indicate that the best results are achieved with K = 3 or 5. We used K = 3 in our experiments since it requires less computation.

5. REFACTORING AT THE CLASS LEVEL

We consider two approaches for refactoring at the class level. The first approach suggests moving methods to one of the existing classes in the system. The methods that are added first to one class are more similar to the class than those which are added later. The second approach suggests moving methods to existing classes of the system or to new classes. Thus, if the similarity among all methods is high, then a small number of classes will be needed. On the other hand, if the similarity among the methods is low, then a large number of classes will be needed. The following sections present the details of these two methods.

5.1 Software Refactoring at the Class Level with a Fixed Number of Classes (Clusters)

To cluster a set of entities into groups, the features of these entities should be extracted. Based on the feature values, the entities are organized into groups. The entities to be clustered, in refactoring at the class level, are the methods of the class and the number of clusters corresponds to the number of classes. Thus, if we have *n* classes with *m* methods, then there will be *n* cluster centres and *m* methods for clustering. In other words, each method will be assigned to one of the *n* clusters (i.e., classes). The assigning process is based on the similarity measure. The similarity measure between a method and a class is the number of attributes that the method uses from that class. For instance, if a method m_1 uses three attributes of *Class A* and two attributes of *Class B*, then method m_1 is more similar to *Class A* than *Class B*. This means that the method m_1 will be assigned to the cluster centre representing *Class A*. The algorithm is described as follows:

```
1. Calculate the similarity between the methods and the containing classes (cohesion).
2. Calculate the similarity between all the methods and other classes (coupling).
3. Find the maximum similarity for each method.
4. Move the method with maximum similarity to the class.
```

Using this approach, each method will be assigned to a class in which the method uses more of the class' attributes. The suggested clustering mechanism depends on the similarity measure. Thus, if we have *n* classes and *m* methods in those classes, then an $m \times n$ similarity matrix is used. The rows represent the methods, and columns represent the classes (cluster centres), and the values in the array represent the similarity values between the methods and the corresponding classes. The similarity value between a class and a method is the number of the class attributes that the method uses.

$$Sim(m_j, C_i) = \text{No of the } C_i\text{'s attributes which the method uses} \quad (5)$$

where: $1 \le i \le n, 1 \le j \le m$

In reality, the values of this matrix are indirectly representing the coupling and cohesion values. For example, if m =16 and $n = 4$, then we have 16 methods to be assigned to 4 class centres. Suppose a *method* m_5 belongs to *Class* C_2. Then, in the similarity matrix, the element $el_{[5,2]}$ represents the cohesion values which is the number of attributes that the method uses from its own class. The elements $el_{[5,1]}$, $el_{[5,3]}$ and $el_{[5,4]}$ in row 5 represent the coupling values with classes C_1, C_3, and C_4, respectively, (i.e., the number of attributes which the method accesses from those classes).

5.2 Software Refactoring at the Class Level with an Adaptive Number of Classes (Clusters)

In this approach we refactor the code without restricting the number of classes; therefore, new classes may be created. This clustering technique suggests several options for the classes' structure, so that the software designer can choose the best structure for the current system.

5.3 Class Refactoring Approach

The aim of the classes refactoring is to enhance the structure of the software. As a result of clustering, the approach provides information about the existing structure of the classes and heuristic guidelines to improve the current structure. These guidelines can be used to aid the software designer in refactoring the current system. Our approach is an extension of approaches for code restructuring proposed by Lung *et al* (2004; 2006) and Alkhalid *et al* (2010). Figure 1 shows the approach.

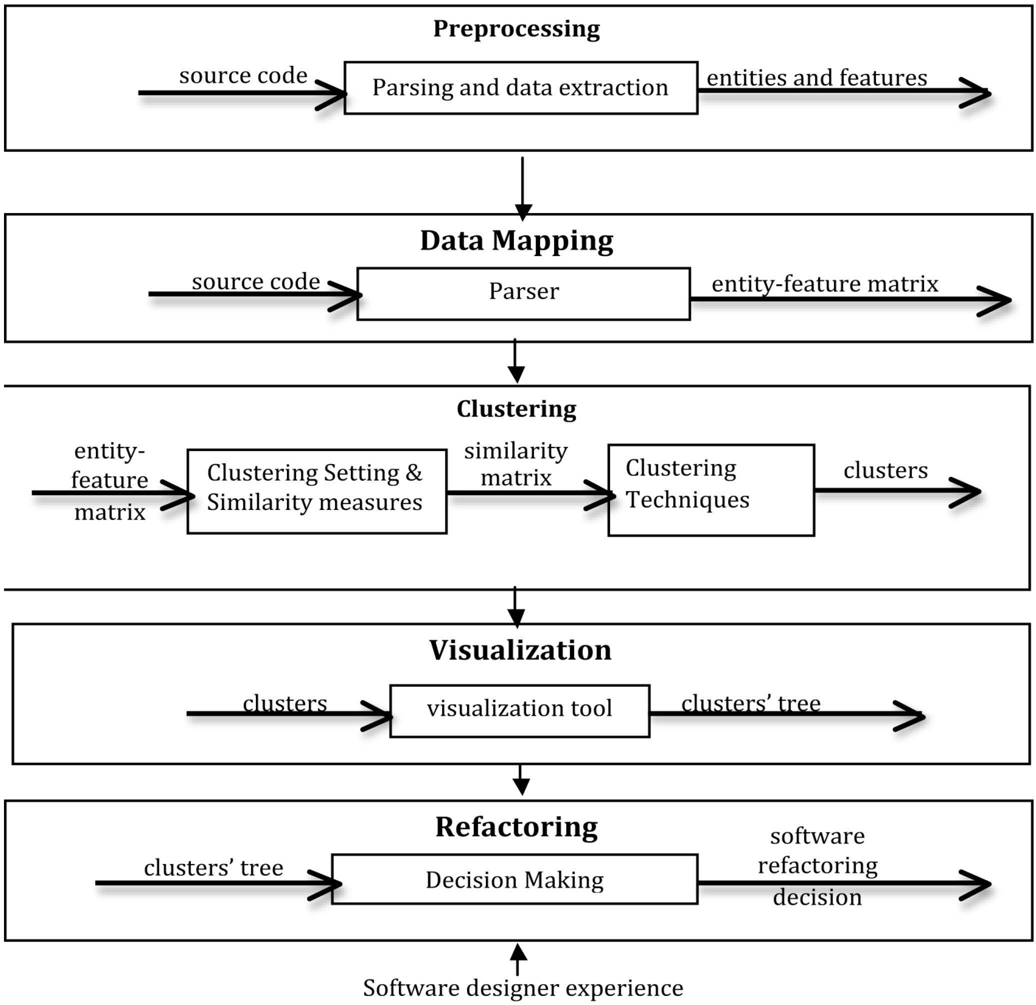


Figure 1: Approach to classes refactoring (Lung *et al*, 2004; 2006) and adapted from Alkhalid *et al* (2010)

The approach consists of five phases:

1. Preprocessing: The source code is parsed, and a list of entities and features is extracted.
2. Data Mapping: The entity-feature matrix is generated. Each entity may describe one or more features. Entities are the elements that will be clustered based on the features they share. Thus, the more features two entities share, the more related those entities are.
3. Clustering: The resemblance coefficient metric is calculated to measure the similarity between every two entities. Mainly, the similarity is measured by the common features that two entities share. After the resemblance coefficient is calculated, the clusters are formed by using SLINK, CLINK, WPGMA and A-KNN techniques.
4. Visualization: The results of the previous phase are displayed as a tree structure. The tree shows the structure of the classes. Closely related entities are grouped in the same class cluster.
5. Refactoring: The clustering tree provides heuristic advice on how to refactor the code. This heuristic advice could be used by the software designer to aid refactoring.

6. EXPERIMENTAL RESULTS

We conducted a number of experiments to evaluate the effectiveness of the proposed approaches. In the first approach, software refactoring at the class level with a fixed number of classes, we use an introduced mechanism in which methods are grouped according to their similarity with the classes.

In the second proposed approach, software refactoring at the class level with an adaptive number of classes, we use SLINK, CLINK and WPGMA, and A-KNN algorithms. The following sub-sections describe the setting and the results of these experiments.

6.1 Experimental Results on Software Refactoring at Class Level using Clustering with a Fixed Number of Classes

Below we present the results of applying the approach on different source codes. We started by applying the approach on a test source code to illustrate the different phases of the algorithm. Then, we applied it on an open source system. After each experiment, we present the analysis and the results of the approach.

6.1.1 Experimental Results on Test Source Code

We applied refactoring at the class level using clustering with a fixed number of classes approach on the source code shown in Figure 2. In this source code, we have four classes A, B, C and D and 16 methods {`method0, method1,..., method15`}. Figure 2 shows the source code of the classes and the methods before clustering. The similarity between methods is recognized from the similarity matrix produced from the entity-feature matrix as described below.

Table 1 shows the similarity matrix for the source code of Figure 2. The rows represent the methods to be clustered, and the columns represent the classes. The values inside the table represent the similarity between each method and all other classes in the system. For instance, the similarity of `method0` with class A is 1, because it uses one attribute of this class. The similarity of `method0` with class B is 2, because it uses two instances of class B. The similarity of the same method with classes C and D is 0, because it does not use any attributes of them. The clustering algorithm maps the methods from `method0` until `method15` and assigns each method to the nearest class (i.e., cluster centre). For example, for `method0`, the nearest class (cluster center) is class B, because the similarity between `method0` and Class B has the maximum similarity between `method0` and all other classes. Thus, clusters will be formed gradually around the four class cluster centres. In each step of clustering, one of the clusters will be assigned to one of the class cluster centres.

```
package refactoring;
public class A {
  int a_attr1, a_attr2, a_attr3, a_attr4;
  B b1=new B();
  public A() {
  }
  public int method0(){
    return a_attr1+
           b1.b_attr1+b1.b_attr2;
  }
  public void method1(){
    this.a_attr1=1;
    this.a_attr2=2;
  }
  public void method2(){
    this.a_attr3=7;
    this.a_attr4=8;
  }
  public void method3(){
    this.a_attr1=0;
    this.a_attr2=0;
    this.a_attr3=0;
    this.a_attr4=0;
  }
}
```

(a)

```
package refactoring;
public class B {
  int b_attr1, b_attr2, b_attr3, b_attr4;
  C c1=new C();
  D d1=new D();
  public B() {
  }
  public void method4(){
    this.b_attr1=0;
    this.b_attr2=0;
    this.b_attr3=0;
    this.b_attr4=0;
  }
  public int method5(){
    return b_attr1 + c1.c_attr1 +
          d1.d_attr1 + d1.d_attr4;
  }
  public int method6(){
    return this.b_attr4+
           this.b_attr3;
  }
  public void method7(){
    d1.d_attr1=7;
 d1.d_attr2=d1.d_attr3+d1.d_attr4;
  }
}
```

(b)

```
package refactoring;
public class A {
  int a_attr1, a_attr2, a_attr3, a_attr4;
  B b1=new B();
  public A() {
  }
  public int method0(){
    return a_attr1+
           b1.b_attr1+b1.b_attr2;
  }
  public void method1(){
    this.a_attr1=1;
    this.a_attr2=2;
  }
  public void method2(){
    this.a_attr3=7;
    this.a_attr4=8;
  }
  public void method3(){
    this.a_attr1=0;
    this.a_attr2=0;
    this.a_attr3=0;
    this.a_attr4=0;
  }
}
```

(c)

```
package refactoring;
public class B {
  int b_attr1, b_attr2, b_attr3, b_attr4;
  C c1=new C();
  D d1=new D();
  public B() {
  }
  public void method4(){
    this.b_attr1=0;
    this.b_attr2=0;
    this.b_attr3=0;
    this.b_attr4=0;
  }
  public int method5(){
    return b_attr1 + c1.c_attr1 +
          d1.d_attr1 + d1.d_attr4;
  }
  public int method6(){
    return this.b_attr4+
           this.b_attr3;
  }
  public void method7(){
    d1.d_attr1=7;
 d1.d_attr2=d1.d_attr3+d1.d_attr4;
  }
}
```

(d)

Figure 2: Source code of classes A, B, C, D before refactoring

Class	Method	A	B	C	D
A	$method_0$	1	2	0	0
	$method_1$	2	0	0	0
	$method_2$	2	0	0	0
	$method_3$	4	0	0	0
B	$method_4$	0	4	0	0
	$method_5$	0	1	1	2
	$method_6$	0	2	0	0
	$method_7$	0	0	0	4
C	$method_8$	0	0	3	0
	$method_9$	0	0	2	0
	$method_{10}$	0	0	2	0
	$method_{11}$	0	0	4	0
D	$method_{12}$	2	1	0	1
	$method_{13}$	0	0	4	0
	$method_{14}$	0	0	0	2
	$method_{15}$	0	0	0	4

Table 1: Similarity Matrix for the original source code in Figure 2

We updated the classes by using the results of the clustering technique. These classes are renamed to UpdatedA, UpdatedB, UpdatedC and UpdatedD. These four new classes have the same attributes, but with different methods' distribution. Figure 3 shows the source code after clustering.

The Lack of Cohesion in Methods (LCOM) metric is used to measure cohesion (Chidamber and Kemerer, 1994), and the Coupling Through Abstract Data Type (CTA) metric is used to measure coupling (Li, 1998). Table 2 shows LCOM values for the four classes before clustering while Table 3

Class A	Class B	Class C	Class D
{method0∩method1}=1 {method0 ∩ method2}=0 {method0 ∩ method3}=1 {method1 ∩ method2}=0 {method1 ∩ method3}=2 {method2 ∩ method3}=2	{method4 ∩ method5}=1 {method4 ∩ method6}=2 {method4 ∩ method7}=0 {method5 ∩ method6}=0 {method5 ∩ method7}=2 {method6 ∩ method7}=0	{method8 ∩ method9}=2 {method8 ∩ method10}=1 {method8 ∩ method11}=3 {method9 ∩ method10}=0 {method9 ∩ method11}=2 {method10 ∩ method11}=2	{method12 ∩ method13}=0 {method12 ∩ method14}=0 {method12 ∩ method15}=1 {method13 ∩ method14}=0 {method13 ∩ method15}=0 {method1 ∩ method15}=2
LCOM= max((2-3),0)=0	LCOM= max((3-3),0)=0	LCOM= max((1-5),0)=0	LCOM= max((4-2),0)=2

Table 2: LCOM values before clustering

```
package refactoring;
class UpdatedA {
  int a_attr1, a_attr2, a_attr3,
      a_attr4;
  D d1=new D();
  B b1=new B();
  public UpdatedA() {
  }
  public void method1(){
     this.a_attr1=1;
     this.a_attr2=2;
  }
  public void method2(){
     this.a_attr3=7;
     this.a_attr4=8;
  }
  public void method3(){
     this.a_attr1=0;
     this.a_attr2=0;
     this.a_attr3=0;
     this.a_attr4=0;
  }
  public void method12(){
     d1.d_attr1 = a_attr2 +
        a_attr3 + b1.b_attr1;
  }
}
```

(a)

```
package refactoring;
class UpdatedB {
  int b_attr1, b_attr2,
      b_attr3, b_attr4;
  A a1=new A();
  public UpdatedB() {
  }
  public int method0(){
     return a1.a_attr1+
            b_attr1+b_attr2;
     }
  public void method4(){
    this.b_attr1=0;
    this.b_attr2=0;
    this.b_attr3=0;
    this.b_attr4=0;
  }
  public int method6(){
     return this.b_attr4+
            this.b_attr3;
  }
}
```

(b)

```
package refactoring;
class UpdatedC {
int c_attr1, c_attr2, c_attr3,
    c_attr4;
  public UpdatedC() {
  }
  public void method8(){
this.c_attr1=this.c_attr2+this.c_attr3;
  }
  public void method9()  {
    this.c_attr1=1;
    this.c_attr2=2;
  }
  public void method10(){
    this.c_attr3=7;
    this.c_attr4=8;
  }
  public void method11(){
    this.c_attr1=0;
    this.c_attr2=0;
    this.c_attr3=0;
    this.c_attr4=0;
  }
  public int method13()  {
    return  c_attr1+c_attr2+c_attr3+c_attr4;
  }
}
```

(c)

```
package refactoring;
class UpdatedD {
 int d_attr1, d_attr2, d_attr3,
     d_attr4;
 B b1=new B();
 C c1=new C();
 public UpdatedD() {
 }
 public int method5(){
     return
b1.b_attr1+c1.c_attr1+d_attr1+d_attr4;
 }
 public void method7(){
     d_attr1=7;
     d_attr2=d_attr3+d_attr4;
 }
 public void method14(){
     this.d_attr3=7;
     this.d_attr4=8;
 }
 public void method15(){
    this.d_attr1=0;
    this.d_attr2=0;
    this.d_attr3=0;
    this.d_attr4=0;
 }
}
```

d

Figure 3: Source code of classes A, B, C, D after refactoring

Class UpdatedA	Class UpdatedB	Class UpdatedC	Class UpdatedD
{method1 ∩method2}=0 {method1 ∩ method3}=2 {method1 ∩ method12}=2 {method2 ∩ method3}=2 {method2 ∩ method12}=1 {method3 ∩ method12}=2	{method0 ∩ method4}=2 {method0 ∩ method6}=0 {method4 ∩ method6}=2	{method8 ∩ method9}=2 {method8 ∩ method10}=1 {method8 ∩ method11}=3 {method8 ∩ method13}=3 {method9 ∩ method10}=0 {method9 ∩ method11}=2 {method9 ∩ method13}=2 {method10 ∩ method11}=2 {method10 ∩ method13}=2 {method11 ∩ method13}=4	{method5 ∩ method7}=2 {method5 ∩ method14}=1 {method5 ∩ method15}=2 {method7 ∩ method15}=4 {method14 ∩ method15}=2
LCOM= max((1-5),0)=0	LCOM=max((1-2),0)=0	LCOM= max((1-9),0)=0	LCOM= max((0-5),0)=0

Table 3: LCOM values after clustering

shows the LCOM value after clustering. The two tables also show the intersection values between methods (how many shared instances there are between every two methods of the classes).

Table 4 shows the values of CTA before and after refactoring.

To summarize the results, Table 5 shows the changes of LCOM and CTA after the refactoring process.

Table 5 clearly shows that there is a decrease in the values of LCOM and CTA. Since the values of LCOM after clustering are less than the values of LCOM before clustering, the cohesion is better after refactoring. CTA value after clustering is also less than the value before clustering, which means that the coupling is improved after refactoring.

	Class A	Class B	Class C	Class D
Before	1	2	0	3
After	2	1	0	0

Table 4: Values of CTA before clustering

	A	B	C	D
LCOM	0	0	0	-2
CTA	+1	-1	0	-3

Table 5: Change in LCOM and CTA values

6.1.2 Experimental Results using an Open Source System

We tested the technique on an open source project called CSGestionnaire (Consept Accessed: 15 October, 2009). CSGestionnaire is software developed for French medico-social institutions which helps in calculating budgets, preparing and printing invoices for patients, and some other features. Table 6 shows the similarity matrix for all classes and its methods inside the printing package. It consists of 9 classes and 59 methods.

Method / Class	Factures Emises Printing	Logger	Data Provider	Filtered Data Provider	Factures Printing	Preview Panel	Gestion naire Printing	Printing	Factures Debiteurs Printing
FacturesEmisesPrinting									
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTwoLinesCell	0	0	0	0	0	0	0	0	0
drawTableHeader	1	0	0	0	0	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableSpecialLine	0	0	0	0	0	0	0	0	0
drawFactureTableLines	2	0	0	3	0	0	0	0	0
drawFactureTail	2	0	0	0	0	0	0	0	0
drawFactureTable	0	0	0	0	0	0	0	0	0
drawFootnote	1	0	0	0	0	0	0	0	0
drawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	0	0	0	0	0	0	0
getPageCount	1	0	0	0	0	0	0		
FacturesPrinting									
getStrValue	0	0	0	0	0	0	0	0	0
drawEtablissementRect	0	0	0	0	3	0	0	0	0
drawFactureRect	0	0	0	0	3	0	0	0	0
drawReferenceRect	0	0	0	0	3	0	0	0	0
drawDomiciliation BancaireRect	0	0	0	0	3	0	0	0	0
drawResidentRect	0	0	1	0	3	0	0	0	0
drawAssureRect	0	0	1	0	3	0	0	0	0
drawDebiteurRect	0	0	1	0	3	0	0	0	0
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTwoLinesCell	0	0	0	0	0	0	0	0	0
drawTableHeader	0	0	0	0	1	0	0	0	0
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableLines	0	0	0	1	1	0	0	0	0
drawFactureTail	0	0	0	1	1	0	0	0	0
drawFactureTable	0	0	0	0	0	0	0	0	0
drawFootnote	0	0	0	0	1	0	0	0	0
drawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	1	1	1	0	0	0	0
getPageCount	0	0	0	0	1	0	0	0	0

Continued on following page

PreviewPanel									
setPageNum	0	0	0	0	0	1	0	0	0
getPageNum	0	0	0	0	0	1	0	0	0
getPageCount	0	0	0	0	0	0	1	0	0
Printing									
doPrinting	0	0	0	0	0	0	0	0	0
createA4Paper	0	0	0	0	0	0	0	1	0
FacturesDebiteursPrinting									
getStrValue	0	0	0	0	0	0	0	0	0
drawTitle	0	0	0	0	0	0	0	0	2
drawEtablissement	0	0	0	0	0	0	0	0	4
drawDestinataire	0	0	1	0	0	0	0	0	2
drawFactureInfo	0	0	2	0	0	0	0	0	3
drawFactureHeading	0	0	0	0	0	0	0	0	0
drawOneLineCell	0	0	0	0	0	0	0	0	0
drawTableHeader	0	0	0	0	0	0	0	0	1
drawTableCell	0	0	0	0	0	0	0	0	0
drawTableLine	0	0	0	0	0	0	0	0	0
drawTableLines	0	0	0	1	0	0	0	0	2
drawFactureTail	0	0	0	1	0	0	0	0	3
drawFactureTable	0	0	0	0	0	0	0	0	1
drawTalon	0	0	1	0	0	0	0	0	3
drawFootnote	0	0	0	0	0	0	0	0	1
DrawFacturePage	0	0	0	0	0	0	0	0	0
Print	0	0	1	1	0	0	0	0	1
getPageCount	0	0	0	1	0	0	0	0	1

Table 6: Similarity matrix for all classes and methods inside the printing package.

We applied our approach by using the previous similarity matrix. The approach suggested one update, which is to move the method drawFactureTableLines from class FacturesEmisesPrinting to class FilteredDataProvider. This will increase cohesion and decrease coupling. However, this reflects the fact that the methods were organized in a very efficient way inside classes to increase cohesion in classes and decrease coupling among them. It is clear that if we flip any two methods between two classes and apply the approach again then the approach will suggest moving the methods back to their classes and this shows how the approach will help to increase cohesion and reduce coupling.

6.2 Experimental Results on Software Refactoring at the Class Level using an adaptive Number of Classes

In this section, we present the results of performing refactoring at class level without fixing the number of clusters. This is done in two phases. In the first phase, we use traditional clustering techniques as a clustering engine. The clustering techniques that are used are SLINK, CLINK, and WPGMA. We conducted three experiments, each using one of those clustering techniques. In the second phase, we used A-KNN as an engine for the clustering phase. The results are compared to determine the best algorithm for the clustering. We tested the technique on an open source project called Java Line Of Code (JLOC) (Blanz Accessed: 19 October, 2009). JLOC provides analysis of the Lines of Code (LOC) for any project. JLOC is written in Java and consists of five classes BasicFileInfo, CommonCounter, Gui, class, Table and one interface ILineCounter. Currently, C++, Java, VB, SQL, Makefile and Matlab files are supported. It counts the total number of comment lines, blank lines and actual source code lines. The classes contain 27 methods and 23 data-members. The entity-feature matrix for the system consist of 27 lines that represent all entities (methods) and 23 columns that represent all features (Data members). The similarity matrix for the system consists of 27 rows and 27 columns, and it represents the similarity between 27 distinct entities.

6.2.1 Experiments using SLINK, CLINK, WPGMA

Figures 4, 5 and 6 show the clustering tree for the three algorithms SLINK, CLINK, WPGMA algorithms respectively.

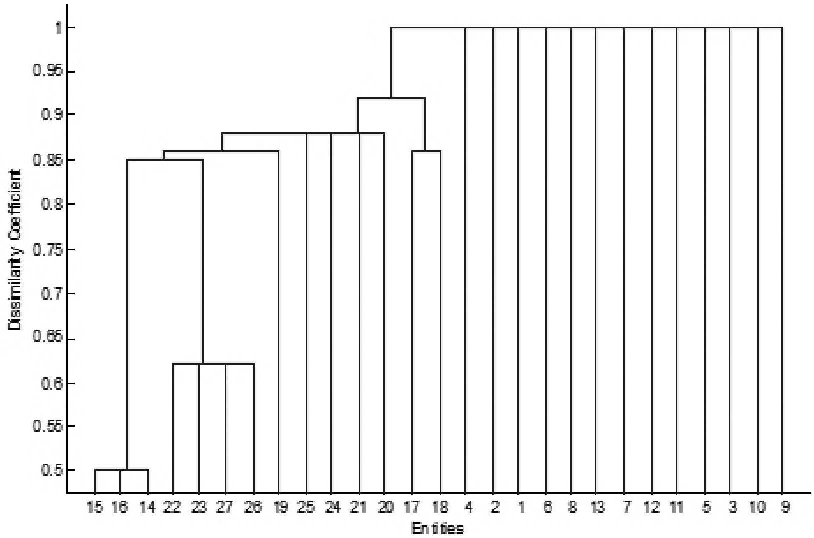


Figure 4: Clustering Hierarchy of JLOC using the SLINK algorithm

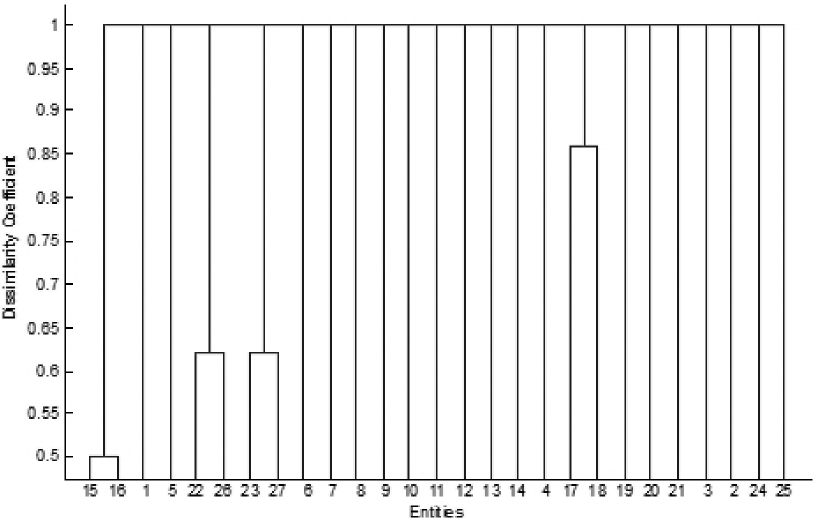


Figure 5: Clustering Hierarchy of JLOC using the CLINK algorithm

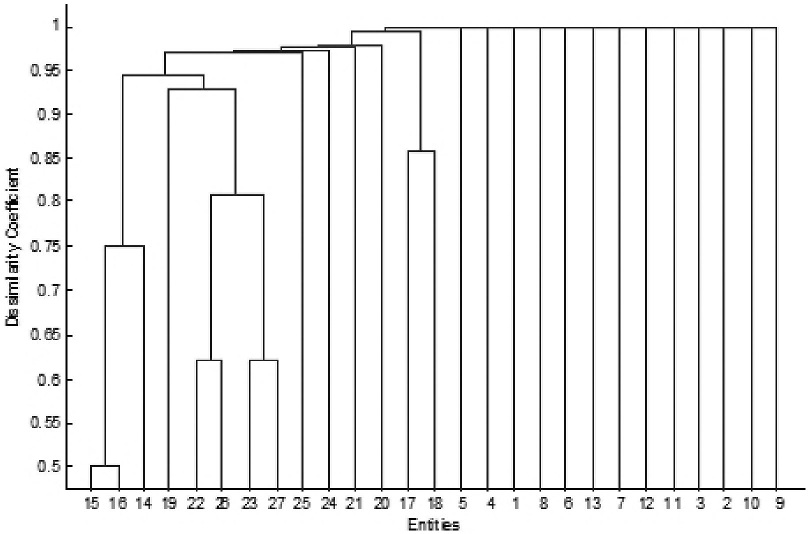


Figure 6: Clustering Hierarchy of JLOC using the WPGMA algorithm

SLINK and WPGMA show excellent behaviour. Both of them grouped methods 14, 15 and 16 in one cluster, methods 22, 23, 26 and 27 in another cluster, and methods 17 and 18 in a third cluster. The performance of CLINK is not as good as SLINK and WPGMA. CLINK merged methods 22 and 26 into one cluster, and methods 23 and 27 into another. However, it did not merge the two clusters into the same cluster. Moreover, CLINK did not merge methods 14, 15 and 16 correctly. SLINK clustering is easier to read. This saves the time required by the software designer to conclude how refactoring can be done. Thus, SLINK has an advantage over WPGMA.

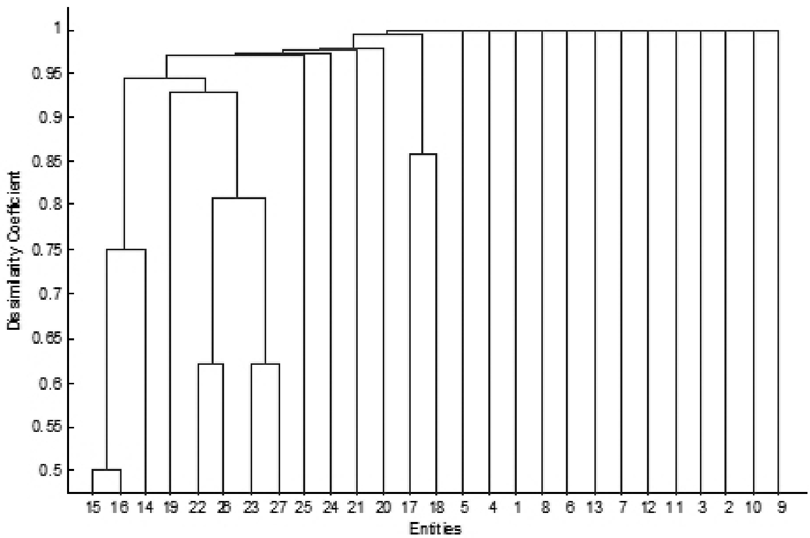


Figure 7: Clustering Hierarchy of the system JLOC using the A-KNN algorithm

6.2.2 Experiments Using A-KNN

The performance of A-KNN was tested and its performance compared with the performance of SLINK, CLINK and WPGMA. Figure 7 shows the clustering tree for the A-KNN algorithms.

A-KNN show a very similar output to SLINK. It groups methods 14, 15 and 16 into one cluster, methods 22, 23, 26 and 27 into another, and methods 17 and 18 into a third cluster. Even the clusters 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 were kept in the same cluster. Thus, A-KNN is better than the traditional CLINK and WPGMA clustering algorithms. Moreover, A-KNN is also better than SLINK since it results in the same clusters and requires less computation.

6.2.3 Analysis of the Results

We applied refactorings based on the results of A-KNN and SLINK, since they show the best performance and their results are very similar. After refactoring, a new class was extracted, we call it TableSchema. The TableSchema class contains methods 12, 20, 24 and 25 (getRowCount, getColumnCount, setColumnName, getColumnName). We measured the values of coupling and cohesion before and after refactoring. We used Coupling Between Object Classes (CBO) (Chidamber and Kemerer, 1994) as a measure of coupling. CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. We used CBO instead of CTA because there was no coupling through abstraction between the classes. Table 7 shows the CBO and LCOM values before and after refactoring.

Table 7 shows that there is an increase in the value of CBO for class Table. This increase is normal because a new class (TableSchema) was added to the system. The class Table is coupled with the new class TableSchema. Thus, there is an increase by 1 in the CBO of the class Table.

The values of LCOM did not change. However, the code restructuring suggested moving methods from class Table to the new class TableSchema. This restructuring makes the two classes Table and TableSchema more cohesive because this will increase the functional cohesion of the two classes (Lethbridge and Laganiere, 2002). Functional cohesion occurs when parts of a class are grouped as they all contribute to a single well-defined task. The new class TableSchema provides the properties of the table. Functional cohesion is the strongest and best kind of cohesion and makes the class more reusable (McConnell, 2004). LCOM is not a good measure for functional cohesion. Therefore, cohesion of the two classes Table and TableSchema has been improved.

	CBO		LCOM	
Node	**Before**	**After**	**Before**	**After**
BasicFileInfo.java	0	0	45	45
CommonCounter.java	2	2	4	4
Gui.java	1	1	4	4
ILineCounter.java	1	1	0	0
Main.java	2	2	0	0
Table.java	3	4	0	0
TableSchema.java		0		0

Table 7: CBO and LCOM values of the system JLOC before and after refactoring

	Method	Class
1	Setup	AccessibleBundleTest
2	tearDown	
3	testToDisplayString_withoutArgAndWithArg	
4	Setup	AccessibleRelationSetTest
5	tearDown	
6	testAccessibleRelationSet	
7	testAddContains	
8	testNullOperations	
9	testDupes	
10	testGet	
11	testClear	
12	testRemove	
13	testToString	
14	test_constructor	
15	Setup	AccessibleStateSetTest
16	tearDown	
17	testAccessibleStateSet	
18	testAddContains	
19	testAddAll	
20	testRemove	
21	testClear	
22	testToString	
23	testToArray	
24	test_constructor	

Table 8: The distribution of methods in the three selected classes from the *accessibility* package.

6.3 Comparison between the two Approaches (Fixed and Adaptive Number of Classes)

In order to compare the two presented approaches, we conducted a set of experiments on another open source system, Apache Harmony (Blanz Accessed: 19 October, 2009). The Apache Harmony is a framework which includes different tools and libraries that provides a Java Standard Edition implementation. The source code of Apache Harmony consists of 4650 Java file each of them containing at least one class. Any change in any class may require many changes in other dependent classes in order to be able to compile the code with no errors. We only selected 3 classes with 15 methods from the accessibility package. We applied our refactoring approaches on these classes and we compared the results. Table 8 shows the distribution of methods in the classes.

6.3.1 Using a Fixed Number of Classes

We applied the first approach on the three classes. Table 9 shows the similarity matrix between the methods (rows) and class cluster centres (columns).

	Method/Class	AccessibleBundleTest	AccessibleRelationSetTest	AccessibleStateSetTest
1	Setup	2	0	0
2	tearDown	2	0	0
3	testToDisplayString_withoutArgAndWithArg	15	0	0
4	Setup	0	4	0
5	tearDown	0	2	0
6	testAccessibleRelationSet	0	3	0
7	testAddContains	0	7	0
8	testNullOperations	0	9	0
9	testDupes	0	15	0
10	testGet	0	3	0
11	testClear	0	2	0
12	testRemove	0	4	0
13	testToString	0	2	0
14	test_constructor	0	3	0
15	Setup	0	0	4
16	tearDown	0	0	2
17	testAccessibleStateSet	0	0	2
18	testAddContains	0	0	12
19	testAddAll	0	0	7
20	testRemove	0	0	3
21	testClear	0	0	4
22	testToString	0	0	5
23	testToArray	0	0	3
24	test_constructor	0	0	0

Table 9: Similarity matrix for some classes and methods from *accessibility* package.

The similarity matrix provided by the approach suggests the same original distribution for the methods inside the classes.

6.3.2 Using an Adaptive Number of Classes

We applied the second approach using SLINK, CLINK, WPGMA and A-KNN. All techniques provide similar results. Figure 8 shows an example of the results using SLINK. We applied refactoring using the suggested solution.

The approach suggested extracting a new class from the three classes. The class is called AccessibleStateSetTestSupplementary. It contains the methods testAccessibleStateSet, testAddContains, testRemove, testClear, and testToArray. All of these methods were extracted from the class Accessible StateSetTest. We applied refactoring according to these suggestions and then measured the values of LCOM and CBO before and after refactoring. Table 10 shows the change in the values of LCOM and CBO after refactoring. It shows that after refactoring the value of LCOM for the whole system has been reduced by 23 and the value of CBO has been increased by 1 in the whole system.

6.3.3 Analysis of the Results

The distribution of classes that was suggested by the first approach is the same as in the original source code. Thus, the approach can be used as a way for auto-packaging. The second approach

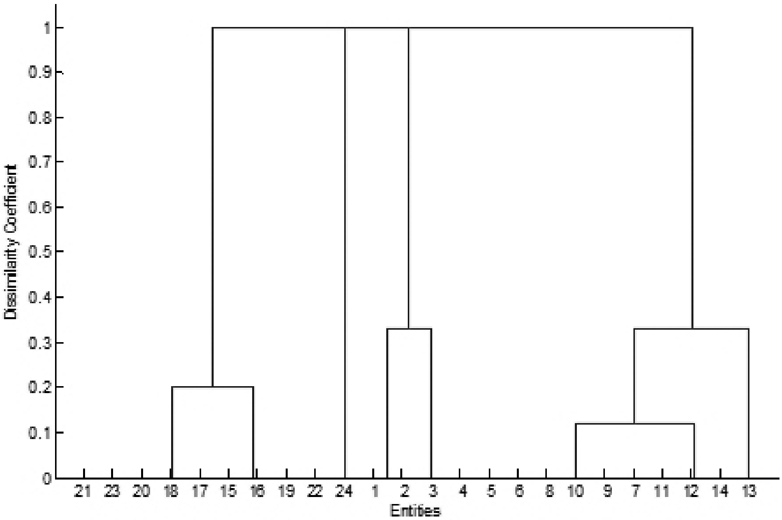


Figure 8: Clustering Hierarchy of for the selected classes and methods from the accessibility package using the SLINK algorithm

	CBO		LCOM	
Node	Before	After	Before	After
javax.accessibility.AccessibleBundleTest	1	1	22	22
javax.accessibility.AccessibleRelationSetTest	1	1	22	22
javax.accessibility.AccessibleStateSetTest	1	1	35	12
javax.accessibility.AccessibleStateSetTestSupplementary		1		0

Table 10: CBO and LCOM values of the selected classes in the accessibility package before and after refactoring.

suggested a different distribution which enhanced the quality of the original code by decreasing the value of the LCOM metric in the system. The increase in the values of CBO is expected because adding a new class to the system should increase coupling. As a result, the second approach is better than the first one because it suggested a way for packaging classes with better cohesion. However, the first approach is more simple and easy to use than the second approach. This is due to the fact that the first approach uses a similarity matrix only while the second approach uses an entity-feature matrix and the similarity matrix. Although the similarity matrix in the second approach is easy to derive from the entity-feature matrix, it still requires additional effort and computation in order to calculate the similarity coefficient value to fill the similarity matrix and then applies one of the clustering algorithms SLINK, LINK, WPGMA or AKNN.

7. CONCLUSIONS

Software refactoring at the class level has great importance in the field of software engineering. In this paper we used pattern recognition techniques to assist in software refactoring. We proposed two approaches to guide the software designer in refactoring at the class level. The first approach is "software refactoring at the class level with a fixed number of classes" and the second approach is "software refactoring at the class level using an adaptive number of classes". We explored the different settings and controllers of these two approaches. We conducted a set of experiments to check the efficiency of these approaches. The experiments are conducted on different experimental units from different sources. The approaches were shown to be helpful in providing a new structure for the system classes. In addition, we compared the performance of the two approaches. The second approach provides better refactoring advice than the first one. We also applied A-KNN which showed its superiority over the traditional SLINK, CLINK, and WPGMA clustering algorithms in terms of performance and computational complexity.

This work can be extended to investigate the performance of the A-KNN clustering algorithm in other fields of research.

ACKNOWLEDGEMENT

The authors acknowledge the support of King Fahd University of Petroleum and Minerals in the development of this work.

REFERENCES

ALKHALID, A., ALSHAYEB, M. and MAHMOUD, S. (2010): Software refactoring at the function level using new Adaptive K-Nearest Neighbour algorithm. *Advances in Engineering Software,* 41(10–11): 1160–1178.

ALKHALID, A., ALSHAYEB, M. and MAHMOUD, S. (2011): Software refactoring at the package level using clustering techniques. *IET Software,* 5(4): 415–424.

ALSHAYEB, M., LI, W. and GRAVES, S. (July 2001): An empirical study of refactoring, new design, and error-fix efforts in extreme programming. In *Proceeding: 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001)* Orlando, Florida.

ANDERBERG, M.R. (1973): *Cluster Analysis for Applications*. Academic Press.

ANQUETIL, N. and LETHBRIDGE, T.C. (2003): Comparative study of clustering algorithms and abstract representations for software remodularisation. In *IEE Proceedings: Software,* 185–201.

BLANZ, E. (Accessed: 19 October, 2009): JLOC, Source Forge, http://sourceforge.net/projects/jloc/

CHIDAMBER, S.R. and KEMERER, C.F. (1994): A metrics suite for object oriented design. *IEEE Transactions on Software Engineering,* 20(6): 476–493.

DEFAYS, D. (1977): An efficient algorithm for a complete link method. *The Computer Journal,* 20(4): 364–366.

DUDA, R.O., HART, P.E. and STROK, D.G. (2000): *Pattern Classification*.

EVERITT, B. (1980). *Cluster Analysis*. Heinemann Educational Books, Ltd.

FIX, E. and HODGES, J. (1951): Discriminatory analysis: Nonparametric discrimination: Consistency properties. USAF School of Aviation Medicine.

FOWLER, M. (1999): *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.

JAIN, A.K., MURTY, M.N. and FLYNN, P.J. (1999): Data clustering: A review. *ACM Computing Surveys (CSUR),* 31(3): 264–323.

JETBRAINS (Accessed: 2 November, 2009): Intelligent Java IDE, "http://www.jetbrains.com/idea/".

KAUFMAN, L. and ROUSSEEUW, P.J. (1990): *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley.

KIN, A.K., DUIN, R.P.W. and MAO, J. (2000): Statistical pattem recognition: A review. *IEEE Transactions on Pattern Analysis Machine Intelligence,* 22(1): 4–37.

LETHBRIDGE, T. and LAGANIERE, R. (2002): *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw-Hill

LI, W. (1998): Another metric suite for object-oriented programming. *Journal of Systems and Software,* 44(2): 155–162.

LUNG, C-H., XU, X., ZAMAN, M. and SRINIVASAN, A. (2006): Program restructuring using clustering techniques. *The Journal of Systems and Software,* 79(9): 1261–1279.

LUNG, C-H., ZAMAN, M. and NANDI, A. (2004): Applications of clustering techniques to software partitioning, recovery and restructuring. *The Journal of Systems and Software,* 73(2): 227–244.

McCONNELL, S. (2004): *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

MENS, T. and TOURWE, T. (2004): A survey of software refactoring. *IEEE Transactions on Software Engineering* 30(2): 126–139.

MURTAGH, F. (1983): A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal,* 26(4): 354–359.

OPDYKE, W.F. (1992): *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. University of Illinois at Urbana-Champaign.

PRESSMAN, R.S. (2010): *Software Engineering: A practitioner's Approach*.

ROMESBURG, H.C. (1990): *Cluster Analysis for Researchers*. Lulu.

RUGABER, S., STIREWALT, K. and WILLS, L. (1996): Understanding interleaved code. *Automated Software Engineering,* 3(1-2): 47–76.

SCIENTIFIC-TOOLWORKS (Accessed: 1 November, 2009). Understand for Java, "http://www.scitools.com/products/understand/".

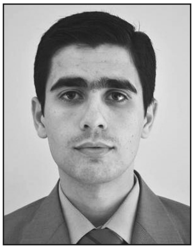
SIMON, F., STEINBRUCKNER, F. and LEWERENTZ, C. (2001): Metrics based refactoring. In *Proceeding of the Fifth European Conference on Software Maintenance and Reengineering*. 30–38.

TAHVILDARI, L. and KONTOGIANNIS, K. (2003): A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceeding of Seventh European Conference on Software Maintenance and Reengineering*. 183–192.

TICHELAAR, S., DUCASSE, S., DEMEYER, S. and NIERSTRASZ, O. (2000): A meta-model for language-independent refactoring. In *Proceeding of International Symposium on Principles of Software Evolution (ISPSE 2000),* 154–164.

WATTANACHON, U. and LURSINSAP, C. (2004): Agglomerative hierarchical clustering for nonlinear data analysis. In *IEEE International Conference on Systems, Man and Cybernetics*, 1420–1425.

BIOGRAPHICAL NOTES

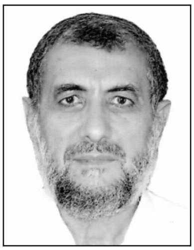


Abdulaziz Alkhalid

Mr Abdulaziz Alkhalid is a PhD student in the Department of Systems and Computer Engineering at Carleton University, Canada. Mr Al-Khalid received a bachelor degree in informatics engineering with a major in software engineering and information systems from Albaath University, Homs, Syria, in August 2005. He completed an MS in computer science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in January 2009. His research interests include software engineering and machine learning.

Mohammad Alshayeb

Dr Mohammad Alshayeb is an associate professor of software engineering in the Information and Computer Science Department at King Fahd University of Petroleum and Minerals. He received his certificate of software engineering, MS and PhD in computer science from the University of Alabama in Huntsville in 1999, 2000 and 2002 respectively. He received his BS in computer science from Mutah University, Jordan in 1995. Dr Alshayeb worked as a senior researcher and software engineer and managed software projects in the United States and the Middle East. He taught and coordinated industrial training courses, and provided consulting services to major industrial and educational institutes. He is a certified project manager (PMP). Dr Alshayeb's research interests include software refactoring, software quality, software measurement and metrics, object-oriented design and empirical studies in software engineering.



Sabri A. Mahmoud

Dr Sabri A. Mahmoud is a professor of computer science in the Information and Computer Science Department, King Fahd University of Petroleum and Minerals. Dr Mahmoud received his BS in electrical engineering from Sind University, Pakistan in 1972, received his MS in computer sciences from Stevens Institute of Technology, U.S.A., in 1980 and his PhD degree in information systems engineering from the University of Bradford, U.K., in 1987. His research interests include pattern recognition, Arabic document analysis and recognition (including Arabic text recognition and writer identification), image analysis (including time varying image analysis and computer vision), and Arabic computing. Dr Mahmoud is a senior member of IEEE. He has published over 70 papers in refereed journals and conference proceedings in his research areas of interest.