Refactorings Detection Using Hierarchical Clustering

ISTVAN GERGELY CZIBULA
Babeš - Bolyai University
Department of Computer Science
1, M. Kogalniceanu Street, Cluj-Napoca
ROMANIA

GABRIELA (SERBAN) CZIBULA
Babeš - Bolyai University
Department of Computer Science
1, M. Kogalniceanu Street, Cluj-Napoca
ROMANIA

Abstract: Refactoring is a process that helps to maintain the internal software quality, during the whole software lifecycle. This paper aims at introducing a new hierarchical clustering algorithm that can be used for improving software systems design, by identifying the appropriate refactorings. The algorithm is named HARD (Hierarchical Clustering Algorithm for Refactorings Determination) and uses a newly introduced measure that estimates the “quality” of a software system. Clustering is used in order to recondition the class structure of a software system. The proposed approach can be useful for assisting software engineers in their daily work of refactoring software systems. We evaluate our approach using the open source case study JHotDraw and a real software system, providing a comparison with previous approaches.

Key–Words: System design, refactoring, clustering

1 Introduction

The structure of a software system has a major impact on the maintainability of the system. 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.

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].

In this paper we propose a new hierarchical clustering algorithm that would help developers to identify the appropriate refactorings in a software system. Our approach takes an existing software and reassembles it using hierarchical clustering, in order to obtain a better design, suggesting the needed refactorings. Applying the proposed refactorings remains the decision of the software engineer.

The main contribution of this paper is to improve the approach from [3], by defining a new agglomerative hierarchical clustering algorithm for identifying refactorings in order to recondition the class structure of software systems. The proposed approach can be useful for assisting software engineers in their daily work of restructuring software systems.

The rest of the paper is structured as follows. Section 2 presents existing approaches in the field of improving software systems structure and in the field of refactoring. The main aspects related to the clustering approach for systems design improvement (that we have previously introduced in [3]) are exposed in Section 3. A new hierarchical clustering algorithm for identifying refactorings is introduced in Section 4. Section 5 provides an experimental evaluation of our approach. A comparison of the approach proposed in this paper with other similar approaches is given in Section 6. Conclusions of the paper and further research directions are outlined in Section 7.

2 Related Work

There are various approaches in the literature in the field of refactoring, but few approaches exist in the direction of automatic detection of refactorings.

A search based approach for refactoring software systems structure is proposed in [11]. 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 [6]. Based on some elementary metrics, the approach in [15] aids the user in deciding what kind of refactoring should be applied. The paper [14] describes a software visualization tool which offers sup-
port 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 [16]. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [10] 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.

We have previously introduced in [3] a clustering approach for identifying refactorings in order to improve the structure of software systems. To our knowledge, there is no approach in the literature that uses clustering in order to improve the class structure of a software system, excepting the approach introduced in [3]. The existing clustering approaches handle methods decomposition [16] or system decomposition into subsystems [10].

3 A Clustering Approach for Refactorings Determination - CARD

In this subsection we briefly describe the clustering approach (CARD) that we have previously introduced in [3] in order to find adequate refactorings to improve the structure of software systems.

CARD approach consists of three steps shortly described below.

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.

Grouping - The set of entities extracted at the previous step are re-grouped in clusters using a clustering algorithm (HARD in our approach). The goal of this step is to obtain an improved structure of the existing software system.

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

A software system \( S \) is considered in [3] as a set \( S = \{s_1, s_2, ..., s_n\} \), where \( s_i \) (\( 1 \leq i \leq n \)) can be an application class, a class method or a class attribute.

As described above, at the Grouping step of CARD, the software system \( S \) has to be re-grouped. This re-grouping can be viewed as a partition \( K = \{K_1, K_2, ..., K_n\} \) of \( S \). In the following, we will refer \( K_i \) as the \( i \)-th cluster of \( K \), \( K \) as a set of clusters, and an element \( s_i \) from \( S \) as an entity. A cluster \( K_i \) from the partition \( K \) represents an application class in the new structure of the software system.

4 A New Hierarchical Clustering Algorithm for Refactorings Determination

In this section we introduce a new hierarchical agglomerative clustering algorithm (HARD), that aims at identifying a partition of a software system \( S \) that corresponds to an improved structure of it. HARD algorithm can be used in the Grouping step of CARD.

HARD is based on the idea of hierarchical agglomerative clustering, uses an heuristic for merging two clusters and a measure for evaluating a partition of the software system from the system’s design point of view.

The heuristic used in HARD is that, at a given step, the two most similar clusters (the pair of clusters that have the smallest distance between them) are merged only if the distance between them is less or equal to a given threshold, \( \text{distMin} \). This means that the entities from the two clusters are close enough in order to be placed in the same cluster (application class). This heuristic is particular to our approach and it will provide a good enough choice for merging two clusters (application classes).

In our clustering approach, the objects to be clustered are the entities from the software system \( S \), i.e., \( O = \{s_1, s_2, ..., s_n\} \). Our focus is to group similar entities from \( S \) in order to obtain high cohesive groups (clusters).

We will adapt the generic cohesion measure introduced in [13] that is connected with the theory of similarity and dissimilarity. In our view, this cohesion measure is the most appropriate to our goal. We will consider the dissimilarity degree between any two entities from the software system \( S \). Consequently, we will consider the distance \( d(s_i, s_j) \) between two entities \( s_i \) and \( s_j \) as expressed in Equation (1).

\[
    d(s_i, s_j) = \begin{cases} 
    1 - \frac{|p(s_i) \cap p(s_j)|}{|p(s_i) \cup p(s_j)|} & \text{if } p(s_i) \cap p(s_j) \neq \emptyset \\
    \infty & \text{otherwise} 
    \end{cases}
\]

where, for a given entity \( e \in S \), \( p(e) \) represents a set of relevant properties of \( e \), defined as follows. If \( e \) is an attribute, then \( p(e) \) consists of: the attribute itself, the application class where the attribute is defined, and all the methods from \( S \) that access the attribute. If \( e \) is a method, then \( p(e) \) consists of: the method itself,
the application class where the method is defined, all
the attributes from \( S \) accessed by the method, all
the methods from \( S \) used by method \( e \), and all methods
from \( S \) that overwrite method \( e \). If \( e \) is a class, then
\( p(e) \) consists of: the application class itself, all the
attributes and the methods defined in the class, all inter-
faces implemented by class \( e \) and all classes extended
by class \( e \).

We have chosen the distance between two entities
as expressed in Equation (1), because it emphasizes
the idea of cohesion. The authors define in [1] cohe-
sion as “the degree to which module components be-
long together”. Our distance, as defined in Equation
(1), highlights the concept of cohesion, i.e., entities
with low distances are cohesive, whereas entities with
higher distances are less cohesive. A theoretical vali-
dation of this statement is given in [12].

Based on the definition of distance \( d \) given in
Equation (1) it can be easily proved that \( d \) is a semi-
metric function, so it can be used for discriminating
the entities from the software system in a clustering
approach. We will consider the distance \( dist(k, k') \)
between two clusters \( k \in \mathcal{K} \) and \( k' \in \mathcal{K} \) \((k \neq k')\)
given by the average link metric, as expressed in
Equation (2). We mention that we use average link
as linkage metric, because we have obtained better re-

t results with this metric.

\[
dist(k, k') = \frac{1}{|k| \cdot |k'|} \sum_{e \in k, e' \in k'} d(e, e') \tag{2}
\]

Starting from the idea that we intend to obtain
high cohesive clusters and as a high cohesion between
two entities from a cluster is given by a low distance
(dissimilarity) between them, we will search for the
clustering with the lowest overall dissimilarity. For
each cluster, we define its diversity as the sum of pair-
wise entities dissimilarities, and we seek to minimize
that sum over all clusters.

Consequently, the diversity of a partition \( \mathcal{K} = \{K_1, K_2, \ldots, K_p\} \), \( \text{DIV}(\mathcal{K}) \), is defined as given in
Equation (3).

\[
\text{DIV}(\mathcal{K}) = \sum_{i=1}^{p} \text{div}(K_i), \tag{3}
\]

where \( \text{div}(K_i) \) represents the diversity of cluster \( K_i \)
and is defined as:

\[
\text{div}(K_i) = \begin{cases} 
\sum_{e \in K_i, e' \in K_i, e \neq e'} d(e, e') & \text{if } |K_i| \neq 1 \\
\infty & \text{otherwise}
\end{cases}
\]

Intuitively, the diversity of a cluster indicates the
cohesion degree between the entities from the corre-
sponding application class. This is due to the fact that
if an entity \( e \) should belong to an application class
(cluster) \( K_i \), then it is very likely that the distance
(Equation (1)) between \( e \) and the elements from \( K_i \)
is less than the distance between \( e \) and all the other
entities from the other clusters.

For these reasons, we intend to minimize the di-
versity of a partition, in order to maximize the cohe-
sion of the corresponding application classes from the
software system. We mention that we are currently
working on giving a rigorous proof for this statement.

The main steps of HARD algorithm are:

- Each entity from the software system is put in its
  own cluster (singleton).
- The following steps are repeated until the parti-
  tion remains unchanged:

  - Select the two most similar clusters from
    the current partition, i.e, the pair of clusters
    that minimizes the distance from Equation
    (2). Let us denote by \( d_{\text{min}} \) this minimum
    distance.
  - If \( d_{\text{min}} \leq d_{\text{Min}} \) (the given threshold),
    then the clusters selected at the previous
    step will be merged, otherwise the partition
    remains unchanged. Let us denote by \( \mathcal{K} \) the
    obtained partition.
  - Compute the diversity of partition \( \mathcal{K} \) (see
    Equation (3)).
  - From all the generated partitions we retain
    the one with the smallest diversity.

In our approach we have chosen the value 1 for the
threshold \( d_{\text{Min}} \), because distances greater than 1
are obtained only for unrelated entities (Equation (1)).

The main refactorings identified by HARD algo-

1. **Move Method** [7] refactoring. It moves a
   method \( m \) of a class \( C \) to another class \( C' \) that
   uses the method most; the method \( m \) of class \( C \n
   - should be turned into a simple delegation, or it
   - should be removed completely. The bad smell
     motivating this refactoring is that a method uses
     or is used by more features of another class than
     the class in which it is defined [14].

   This refactoring is identified by HARD algorithm
   by moving the method \( m \) in the cluster corre-
   sponding to the application class \( C' \).

2. **Move Attribute** [7] refactoring. It moves an
   attribute \( a \) of a class \( C \) to another class \( C' \) that uses

   -
the attribute most. The bad smell motivating this refactoring is that an attribute is used by another class more than the class in which it is defined [14].

This refactoring is identified by HARD algorithm by moving the attribute $a$ in the cluster corresponding to the application class $C'$. 

3. **Inline Class [7] refactoring.** It moves all the members of a class $C$ into another class $C'$ and deletes the old class. The bad smell motivating this refactoring is that a class is not doing very much [14].

This refactoring is identified by HARD algorithm by decreasing the number of application classes in the new structure of $S$. Classes $C$ and $C'$ with their corresponding entities (methods and attributes) will be merged in the same cluster.

4. **Extract Class [7] refactoring.** Creates a new class $C$ and move some cohesive attributes and methods into the new class. The bad smell motivateing this refactoring is that one class offers too much functionality that should be provided by at least two classes [14].

This refactoring is identified by HARD algorithm by increasing the number of application classes in the new structure of $S$. Consequently, a new cluster appears, corresponding to a new application class in the new structure of $S$.

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

## 5 Experimental Evaluation

In order to validate our clustering approach, we will consider two evaluations, which are described in Subsections 5.1 and 5.2.

### 5.1 JHotDraw Case Study

Our first evaluation is the open source software JHotDraw, version 5.1 [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.

Our focus is to test the accuracy of our approach on JHotDraw, i.e., how accurate are the results obtained after applying HARD algorithm in comparison with the current design of JHotDraw. As JHotDraw has a good class structure, HARD algorithm should generate a nearly identical class structure. We evaluate how similar is a partition the partition of JHotDraw determined after applying HARD algorithm with its actual partition (that is considered a good partition, as JHotDraw is well known for a good design).

After applying HARD algorithm on JHotDraw we have obtained the following results:

- The algorithm obtains a new class after the re-grouping step, meaning that an Extract Class refactoring is suggested. The methods which are placed in the new class are: PertFigure.handle, GroupFigure.handle, TextFigure.handle, StandardDrawing.handle.
- In the obtained partition there are no misplaced attributes.
- In the obtained partition there are four misplaced methods.

In our view, the refactoring identified at (i) can be justified. All these methods provide similar functionalities, that is why, in our view, these methods can be extracted in a new class in order to avoid duplicated code, applying Extract Class refactoring.

### 5.2 A Real Software System

This subsection describes the second case study used for evaluating HARD algorithm. It is a DICOM (Digital Imaging and Communications in Medicine) [5] and HL7 (Health Level 7) [9] compliant PACS (Picture Archiving and Communications System) system, facilitating medical images management, offering quick access to radiological images, and making the diagnosis process easier.

The analyzed application is a distributed system, currently used by hospitals in locations such as Romania, United Kingdom, South Africa, Bulgaria and the Republic of Moldova. It is a large system that consists of several subsystems in form of stand-alone and web-based applications. We have applied HARD algorithm on one of the subsystems from this application.

For confidentiality reasons, we will refer the analyzed application as $A$. $A$ is a stand-alone Java application used for archiving radiological images for long time storage (on CD, DVD or tape). $A$ consists of 675 classes, 5759 methods and 2970 attributes.

After applying HARD algorithm, a total of 56 refactorings have been suggested: 4 Move Attribute
refactorings, 51 Move Method refactorings, and 1 Extract Class refactoring.

The obtained results have been analyzed by the developers of A and the following conclusions were made: 17 refactorings identified by HARD were accepted by the developers as useful in order to improve the system; 13 refactorings were acceptable for the developers, but they concluded that these refactorings are not necessary in the current stage of the project; 26 refactorings were strongly rejected by the developers.

Analyzing the obtained results, based on the feedback provided by the developers, we have concluded the following:

HARD successfully identified classes with low cohesion (classes with more than one responsibility), misplaced constants (constants used only on a subtree of a class hierarchy, but defined in some base class). These kind of weaknesses can be discovered only if the developer manually inspects all the classes, or if a bug arises. That is why automatic detection by HARD of these kind of weaknesses can prevent system failure or other kind of bugs and also save a lot of manual work.

A large number of miss-identified refactorings are due to technical issues: the use of Java anonymous inner classes, introspection, the use of dynamic proxies. These kind of technical aspects appear frequently in projects developed in JAVA. In order to correctly deal with these aspects, we have to improve only the Data collection step of our approach, without modifying HARD algorithm.

Another cause of miss-identified refactorings is due to the fact that the distance (Equation (1)) used for discriminating entities in the clustering process consider only two aspects of a good design: low coupling and high cohesion. It would be also important to consider other principles related to an improved design, like: Single Responsibility Principle, Open-Closed Principle, Interface Segregation Principle, Common Closure Principle [4], etc. Future improvements of our approach will deal with these aspects, also.

6 Comparative analysis with existing approaches

The only approach on the topic studied in this paper, that partially gives the results obtained on a relevant case study (like JHotDraw) is [11]. The authors use an evolutionary algorithm in order to judge how stable are the results, while HARD algorithm from our approach is executed just once.

The technique from [11] reports 10 misplaced methods, while in our approach there are only 4 misplaced methods.

The overall running time for the technique from [11] is about 300 minutes (30 minutes for one run), while HARD algorithm in our approach provide the results in about 4.73 minutes. We mention that the execution was made on similar computers.

Because the results are provided in a reasonable time, our approach can be used for assisting developers in their daily work for improving software systems.

The authors from [14] present a short example illustrating the Move method refactoring. We have applied HARD algorithm on the code example from [14] and the suggested refactoring was obtained by our algorithm, also.

A clustering approach for identifying refactorings in order to improve the structure of software systems is developed in [3]. For this purpose, a clustering algorithm (named kRED) is introduced and an evaluation of kRED algorithm on JHotDraw case study is provided.

A comparison between HARD algorithm introduced in this paper and kRED algorithm is illustrated in Table 1. The comparison is made considering the following characteristics: the number of misplaced methods, the number of misplaced attributes, the running time and if the algorithm identifies or not the Extract Class refactoring. We mention that the execution was made on similar computers.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>No. of misplaced methods</th>
<th>No. of misplaced attributes</th>
<th>Running time (min)</th>
<th>Extract Class refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARD</td>
<td>4</td>
<td>0</td>
<td>4.73</td>
<td>Yes</td>
</tr>
<tr>
<td>kRED</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>No</td>
</tr>
</tbody>
</table>

From Table 1 we can conclude that:

- The results obtained by HARD are better than the results provided by kRED algorithm, as the
numbers of methods and attributes misplaced by HARD are less than those misplaced by the other two algorithms.

- The running time of HARD is less than the running time of kRED.

We cannot make a complete comparison with other refactoring approaches existing in the literature, because, for most of them, the obtained results for relevant case studies are not available. Most approaches give only short examples indicating the obtained refactorings. Other techniques address particular refactorings: the one in [16] focuses on automated support only for identifying ill-structured or low cohesive functions and the technique in [10] focuses on system decomposition into subsystems. The comparisons presented above show that CARD clustering approach using HARD algorithm provides better results than similar approaches existing in the literature.

7 Conclusions and Further Work

Starting from the approach introduced in [3], we have presented in this paper a new hierarchical clustering algorithm (HARD) that can be used for improving software systems design.

We have demonstrated the potential of our algorithm by applying it to the open source case study JHotDraw and a real software system and we have also presented the advantages of our approach in comparison with existing approaches.

Further work can be done in the following directions: to use other search based approaches and machine learning techniques in order to determine refactorings that improve the design of a software system; to improve the distance semi-metric used for for discriminating the entities from the software system; to apply our approach in order to transform non object-oriented software into object-oriented systems.

Acknowledgements: This work was supported by the research project TD No. 411/2008, sponsored by the Romanian National University Research Council.

References: