STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 2, 2012

A STUDY ON HIERARCHICAL CLUSTERING BASED SOFTWARE RESTRUCTURING

ZSUZSANNA MARIAN

Abstract. Finding refactorings that can automatically improve the internal structure of a software system is continuously researched in the search based software engineering literature. In this paper we will investigate through a case study, whether the use of unsupervised learning methods (hierarchical clustering) can be beneficial in the process of automatic refactoring identification. We will compare the results of two algorithms (one that uses hierarchical clustering and one that does not) for a case study, and show that the algorithm that uses hierarchical clustering is capable of finding refactorings which are not found by the other algorithm.

1. Introduction

It is well-known that software system restructuring (also called refactoring) is an important part both for the development and the maintenance of software systems. During development, with the change of requirements or with the addition of new features, refactoring might become necessary. Also, refactoring is one of the steps in the Test-Driven Development cycle. During the maintenance phase of the software life-cycle, refactoring can be used to restructure the system in order to facilitate other maintenance activities, such as improving performance or implementing new features.

Since nowadays most software systems are complex, containing many classes with complicated relations between them, the field of automatic refactorings identification gained importance. The development of new algorithms and tools that are capable of finding a good restructuring of a software system is an important research domain.

Received by the editors: April 10, 2012.

2010 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - Restructuring, reverse engineering, and reengineering; D.2.8 [**Software Engineering**]: Metrics - Product metrics; I.5.3 [**Computing Methodologies**]: Pattern Recognition - Clustering.

Key words and phrases. software refactoring, hierarchical clustering, software metrics.

In this paper we aim at investigating the usefulness of using unsupervised learning for software restructuring. The case study we considered for evaluation highlights the effectiveness of unsupervised learning models to uncover hidden patterns in data.

The structure of this paper is the following: Section 2 presents a short review of the existing literature. Section 3 presents the existing theoretical background. Section 4 describes the HAC algorithm, a method that uses unsupervised learning for software restructuring. Section 5 presents a case study we used, and the results obtained for an existing algorithm (Subsection 5.1) and our unsupervised learning based algorithm (Subsection 5.2), together with a comparative analysis of the results (Subsection 5.3). Section 5 ends with the comparison of our algorithm to other existing algorithms from the literature that use unsupervised learning (Subsection 5.4). Finally, Section 6 presents our conclusions and further research directions.

2. Literature review

In this section we aim at presenting other clustering based software restructuring methods that can be found in the literature.

One of the early works that examines the idea of using clustering for reverse engineering is [1]. It describes a series of experiments on how clustering could be used for software remodularization by defining how an entity can be represented, how to decide when two entities should belong to the same clusters and how to apply a clustering algorithm to the entities. It is an important study, many other papers that use clustering reference it, for example when deciding what kind of linkage method to use.

A recent paper, [11], first identifies classes that have a low cohesion and then finds *Extract Class* refactorings to divide the class into more cohesive parts. In the first step, three software metrics are used (LCOM2, TCC and DOCMA(AR)) to find classes with low cohesion, also called God classes. In the second step, agglomerative clustering is used for the found classes, using as a distance metric the Jaccard distance on property sets defined for attributes and methods. The property set for an attribute is the attribute and the methods that use it, while for a method it contains the method itself and the methods and fields used by it.

Another paper that finds *Extract Class* refactorings is [5]. They use hierarchical clustering for attributes and methods inside a class, using as distance the Jaccard distance on entity sets. The entity set of an attribute contains the attribute and the methods that access it, while the entity set of a method contains the method itself, the methods invoked by it and the attributes used by it.

Unlike the above presented two methods, the HARS algorithm presented in [4], is capable of finding many types of refactorings (*Move Method*, *Move Attribute*, *Inline Class* and *Extract Class*). It considers as entity a method, a class or an attribute, and defined a set of relevant properties, and a distance metric between these sets. This method is similar to the one we will present below.

3. Background

We have previously introduced in [10] an approach that uses software metrics for identifying an improved structure of a software system, i.e. a structure that is likely to correspond to an improved design of it. In this section we will give a brief overview of the vector space model (Subsection 3.1) and the ARI algorithm (Subsection 3.2) introduced in [10].

3.1. **The Vector Space Model.** The main idea of this approach is to represent the components of a software system as a multidimensional vector, whose elements are the values of different software metrics applied to the given component. In order to give a formal definition, we considered that a software system S is a set of components $S = \{s_1, s_2, ..., s_n\}$, where s_i, $1 \leq i \leq n$, can be either an application class, or a method from an application class, and will be called *entity* in the following.

Each entity from a software system is characterised in [10] by a list of relevant features, which are the values for the following software metrics:

(1) Relevant Properties (RP) [3]
(2) Depth in Inheritance Tree (DIT) [2]
(3) Number of Children (NOC) [2]
(4) Fan-In (FI) [8] and [9]
(5) Fan-Out (FO) [8] and [9]

Using the above presented metrics, each entity s_i, $1 \leq i \leq n$, from the software system S can be represented as a 5-dimensional vector, having as components the values of the different metrics, scaled to [0,1]: $s = (rp(s_i), dit(s_i), noc(s_i), fi(s_i), fo(s_i))$ or, more formally, $s = (s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5})$. Each element s_{ik}, $1 \leq k \leq 5$ is the value of the corresponding software metric, and the number of metrics is denoted by m (to have a more general definition).

Using this vector space model representation of the entities, a distance metric, $d(s_i, s_j)$, can be defined as an adaptation of the *Euclidian distance.* This metric, computed using Formula 1, is defined in such a way to measure the dissimilarity between two entities. This means that it will assign small values to pairs of entities that are cohesive and have to belong together, in the same application class, and high values (possibly ∞) to pairs that are not cohesive.

(1)

$$d(s_i, s_j) = \begin{cases} 0 & if \quad i = j \\ \sqrt{\frac{1}{m} \cdot \left(1 - \frac{|s_{i1} \cap s_{j1}|}{|s_{i1} \cup s_{j1}|} + \sum\limits_{k=2}^{m} (s_{ik} - s_{jk})^2\right)} & if \quad s_{i1} \cap s_{j1} \neq \emptyset \\ \infty & otherwise \end{cases},$$

It is easy to prove that Formula 1 is a semi-metric function, and that the distance between two entities that have common relevant properties is less than 1.

3.2. **The Automatic Refactorings Identification - ARI algorithm.** Considering the vector space model presented in 3.1, an algorithm capable of finding a good restructuring of a software system was introduced in [10]. The input of the algorithm is the vector space model representation for each entity s from the software system S, while the the output is a grouping of these entities into different groups - *clusters*, - according to their similarity. Each such cluster corresponds to an application class from the system. The set of all clusters is called a partition and it represents a possible structure of the software system S. The algorithm ARI tries to find a partition that corresponds to a good internal structure of S. Then, comparing this partition to the original structure of the software system, a list of refactorings can be identified.

The ARI algorithm initially places each entity that is an application class into a separate cluster. Then, the main idea is to place each method from the software system in the cluster (i.e. application class), to which it is closest, considering the distance metric d, or to place it in a new cluster if its distance to the already existing application classes is greater than 1. Finally, if there are application classes whose distance is less than 1, they will be merged.

Comparing the result of the ARI algorithm to the original partition of the software system, three types of refactorings can be identified: *Move Method*, *Extract Class*, *Inline Class*. For a detailed description of the ARI algorithm and the identified refactorings, one can consult [10].

4. Unsupervised learning based restructuring

In this section we introduce an extension of the algorithm proposed in [10], presented in Subsection 3.2, named HAC (Hierarchical Agglomerative Clustering). It uses the same vector space model and distance semi-metric as the ARI algorithm and hierarchical clustering, an unsupervised learning method, to build a good restructuring of a software system.

The HAC algorithm is based on *hierarchical agglomerative clustering* and uses a heuristic function for merging two clusters. This heuristic function expresses that two clusters are merged only if the distance between them is less than 1. This value for threshold was chosen, because distances higher than 1 are obtained only for unrelated entities. The distance between two clusters was computed using *complete link* linkage method, because this method gave the best results.

The main steps the HAC algorithm performs for identifying the partition $\mathcal{K}$ that is likely to correspond to an improved structure of the software system S are the following:

(1) Initialise $\mathcal{K}$ as an empty partition.
(2) For each entity s_i from S create a new cluster that contains only s_i, and add it to $\mathcal{K}$. Now $\mathcal{K} = \{K_1, K_2, ..., K_n\}$ is the initial partition, containing as many clusters as the number of entities in S.
(3) As long as changes can be done, the following steps are repeated:
    i. Compute the distance $d_{i,j}$ as $d(K_i, K_j)$, where $1 \leq i \leq n$ and $i \leq j \leq n$.
    ii. Select the minimum distance d_{min} from the distances computed at the previous step. Let i^* and j^* be the indexes of the clusters whose distance is d_{min}.
    iii. If $d_{min} \leq 1$ then we will create a new cluster, $K_{new} = K_{i^*} \cup K_{j^*}$, and modify $\mathcal{K}$ in the following way: $\mathcal{K} = \mathcal{K} \setminus \{K_{i^*}, K_{j^*}\} \cup K_{new}$.

Like the ARI algorithm, this algorithm will also return a partition $\mathcal{K}$ of the software system, where each cluster corresponds to an application class. If this partition is equal to the original partition of S, $\mathcal{K}'$, then we consider that S has a good internal structure and no changes need to be done. If there are differences between $\mathcal{K}$ and $\mathcal{K}'$, S needs to be restructured. Different types of differences correspond to different refactorings. The refactorings identified by the HAC algorithm are the following:

- **Move Method refactoring** [6] - It means moving method m from class c to class c'. Such a refactoring is justified when a method uses or is used more by a different class than its own class. This type of refactoring is identified, when a method m will be placed by HAC into a cluster with a different application class, than its original application class.
- **Extract Class refactoring** [6] - Extract class refactoring means creating a new application class c and moving some methods into it. This refactoring is justified in case of a large class, with many functions that should be split into at least two classes. This type of refactoring appears, when at the end of HAC there are clusters, that contain only

methods, and no application classes. For these clusters a new application class will be created that will contain the methods from the corresponding cluster. Also, in such cases, the number of clusters in $\mathcal{K}$ increases compared to the number of clusters in $\mathcal{K}'$.

- **Inline Class refactoring** [6] - It means moving one class (together with methods and attributes), inside of another class, and it usually happens for small classes that does not do many things. This type of refactoring is identified, when $\mathcal{K}$ contains clusters which have two application classes inside. In this case, one of the classes will become an inline class for the other one, which means that it will be moved - together with methods and attributes - inside the other class. The presence of such refactoring is shown also by the fact that $\mathcal{K}$ contains fewer clusters than $\mathcal{K}'$.

5. Computational experiments

In this section we aim at experimentally evaluating ARI and HAC algorithms, providing a comparative analysis of the obtained results. Two case studies are considered for evaluation, for which we applied both algorithms. The first case study was a small, artificial example (taken from [12]) with two classes, one of them containing a method that should belong to the other class. For this simple example both algorithms identified the correct restructuring.

For our second case study we used JHotDraw (version 5.1), an open source software [7]. It is a Java GUI framework for technical and structured graphics, developed by Erich Gamma and Thomas Eggenschwiler. We chose JHotDraw because it is a well-known example for the use of design patterns and for good design, so we expected our algorithms to find only a few possible refactorings. Another reason for choosing JHotDraw was the fact that, unlike our first case study, this is a complex project, consisting of 173 classes, 1375 methods and 475 attributes.

5.1. **ARI results.** Applying the ARI algorithm to the JHotDraw framework, 20 *Move Method* refactorings were identified, which are presented in Table 1. The first column shows the name of the method to be moved, while the second contains the name of the class where the method should be moved to. The last column shows whether we considered the given refactoring justifiable or not. The justifications for the values in the last column (both Justified and Misplaced) are given below. When deciding on the justifications, we considered three criteria. The first criterion was, whether the refactoring is justified conceptually. The second criterion was, how complicated would it be to perform the actual refactoring. In case of the Move Method refactorings, we considered how many attributes does the method use and where is it called inside the class. The third criterion, used mainly for the Extract Class refactoring identified by the HAC algorithm, was whether the methods to be moved to a new class could be replaced by a single method, sufficiently general that it would implement the same responsibilities as the methods replaced with it. For this criterion the justification was the"Rule of Three" presented in Martin Fowler's book on refactoring: when you write a similar piece of code the third time, it is time to refactor [6].

	Method	Target class	Remark
1.	**DrawApplet.createAttributeChoices**	**CommandChoice**	Justifiable
2.	**DrawApplet.createFontChoice**	**CommandChoice**	Justifiable
3.	**DrawApplication.saveAsStorableOutput**	**StorableOutput**	Justifiable
4.	**DrawApplication.paletteUserOver**	**ToolButton**	Justifiable
5.	**DrawApplet.paletteUserSelected**	**ToolButton**	Justifiable
6.	**DrawApplet.paletteUserOver**	**ToolButton**	Justifiable
7.	**DrawApplet.toolDone**	**ToolButton**	Misplaced
8.	**DrawApplication.createColorMenu**	**ColorMap**	Misplaced
9.	**DrawApplet.setupAttributes**	**ColorMap**	Misplaced
10.	**DrawApplet.createColorChoice**	**ColorMap**	Misplaced
11.	**DrawApplication.createEditMenu**	**CommandMenu**	Misplaced
12.	**DrawApplication.createAlignmentMenu**	**CommandMenu**	Misplaced
13.	**DrawApplication.createArrowMenu**	**CommandMenu**	Justifiable
14.	**DrawApplication.createFontMenu**	**CommandMenu**	Justifiable
15.	**DrawApplication.createFontSizeMenu**	**CommandMenu**	Justifiable
16.	**DrawApplication.createFontStyleMenu**	**CommandMenu**	Justifiable
17.	**DrawApplication.selectionChanged**	**CommandMenu**	Misplaced
18.	**DrawApplet.readFromStorableInput**	**StorableInput**	Justifiable
19.	**PertFigure.asInt**	**NumberTextFigure**	Misplaced
20.	**PertFigure.setInt**	**NumberTextFigure**	Misplaced

TABLE 1. *Move Method* refactorings suggested by the ARI algorithm

We give below the justification for the extracted refactorings.

DrawApplet.createFontChoice, *DrawApplet.createAttributeChoices* and *DrawApplet.createColorChoice.* These are the only three methods from the DrawApplet class that have a name of the form *"createSomethingChoice"* and all three of them were identified as possible *Move Method* refactorings, but *createColorChoice* was suggested to be moved to the *ColorMap* class (this is why it is marked "Misplaced"). All three of them use the method *addItem* from class *CommandChoice* many times, which can be a sign of a need for refactoring.

DrawApplication.saveAsStorableOutput. This method creates an object of type *StorableOutput* and calls its *writeStorable* method, passing to it as parameter an object of type *Drawing* (which, in turn, extends interface *Storable*). Analysing the method *writeStorable*, we can see that the most important part is calling the *write* method on the *Storable* object. Since the main part of writing is already done in class *StorableOutput*, and *saveAsStorableOutput* does nothing strictly related to the class *DrawApplication*, it would make sense moving the whole method to the *StorableOutput* class.

DrawApplication.paletteUserOver, *DrawApplet.paletteUserSelected*, *DrawApplect.paletteUserOver*. Methods *paletteUserOver* and *paletteUserSelected* are defined in interface *PaletteListener*. Since both *DrawApplication* and *DrawApplet* implement this interface, the methods are implemented in both classes (so there is a *paletteUserSelected* method in class *DrawApplication*, too). The implementation of the methods is the same in both classes, which is an argument for moving them to a single class. It would be possible to move the implementation of the *PaletteListener* interface, together with the methods, to the *ToolButton* class as suggested by our algorithm.

DrawApplet.toolDone. This method comes from the *DrawingEditor* interface which is implemented by the *DrawApplet* class. Since this interface contains six methods, moving only one of them to a separate class is impossible.

DrawApplication.createColorMenu, *DrawApplet.setupAttributes*, *DrawApplet.createColorChoice*. All three methods use many functions from the *ColorMap* class, but they cannot be moved there. As mentioned above *createColorChoice* could be moved, but not to the *ColorMap* class. The same is true for the *createColorMenu* method, and it will be detailed in the next part. *SetupAttributes* uses five attributes of the *DrawApplet* class, so it cannot be moved either.

DrawApplication.createColorMenu, *DrawApplication.createEditMenu*, *DrawApplication.createAllignmentMenu*, *DrawApplication.createArrowMenu*, *DrawApplication.createFontMenu*, *DrawApplication.createFontSizeMenu*, *DrawApplication.createFontStyleMenu*. First of all, *createColorMenu* was marked "Misplaced", because it was suggested to be moved to class *ColorMap* instead of *CommandMenu* like the rest of the methods. The rest of the methods are quite independent from class *DrawApplication*, they use only one of its attributes, which could easily be passed as a parameter if the methods were moved. The reason why methods *createEditMenu* and *createAllignmentMenu* were marked as "Misplaced" is that they are called in a method *createMenus*, together with three other methods that create menus and it would be strange to move two methods to a different class, but leave three in *DrawApplication*.

DrawApplet.readFromStorableInput. This case is very similar to the one with method *DrawApplication.saveAsStorableOutput*, the main part is done in the *read* method from the *Storable* interface and the *readStorable* method from class *StorableInput*, so the whole method could be moved to the *StorableInput* class.

PertFigure.asInt, *PertFigure.setInt*. The source code of the JHotDraw framework contains a package *samples* with four examples of how to use the framework. These two methods belong to the example *pert*, so they cannot be moved into a class from the framework.

5.2. **HAC results.** Applying the HAC algorithm to the same JHotDraw framework, we got slightly different results: out of the 158 clusters identified by the algorithm only 3 did not correspond to the existing structure of the framework. These three clusters, represent possible *Extract Class* refactorings, meaning that new classes should be created, and the selected methods moved to the classes. The methods belonging to these three clusters are presented in Table 2. Just like in Table 1, we marked each method either "Justifiable" or "Misplaced". The reasons for assigning the labels are presented below.

	Method	Remark
1	**ChangeConnectionHandle.findConnectableFigure**	Justifiable
2	**ConnectionHandle.findConnectableFigure**	Justifiable
1	**GroupFigure.handles**	Justifiable
2	**TextFigure.handles**	Justifiable
3	**StandardDrawing.handles**	Justifiable
4	**NodeFigure.handles**	Misplaced
5	**PertFigure.handles**	Misplaced
1	**GroupCommand.execute**	Misplaced
2	**UngroupCommand.execute**	Misplaced
3	**AlignCommand.execute**	Misplaced
4	**SendToBackCommand.execute**	Justified
5	**BringToFrontCommand.execute**	Justified

TABLE 2. *Extract Class* refactorings suggested by the HAC algorithm

We give below the justifications for the found refactorings.

ChangeConnectionHandle.findConnectableFigure, *ConnectionHandle.findConnectableFigure*. The reason why these two methods belong to the same cluster is that they look exactly the same. They both are private methods which take as parameter two integers and a *Drawing*, and they are both called from the *invokeStep* method of their class, so they could be moved together in the same class.

GroupFigure.handles, *TextFigure.handles*, *StandardDrawing.handles*, *NodeFigure.handles*, *PertFigure.handles*. All these methods are similar, but the last two of them are marked "Misplaced", because they belong to the examples provided in the JHotDraw framework. All of them build a vector containing four objects from classes that implement the *Handle* interface. They also use the *RelativeLocator* class to define the position of these handles. This class contains nine different "positions": center, southWest, southEast, south, northWest, northEast, west, north, east, but in the above methods only northWest, northEast, southWest and southEast is used.

GroupCommand.execute, *UngroupCommand.execute*, *AlignCommand.execute*, *SendToBackCommand.execute*, *BringToFrontCommand.execute*. Even if all the methods use only one or two attributes from their class, only the methods *SendToBackCommand* and *BringToFrontCommand* can be moved together, because they are very similar, with the difference that one calls the *sendToBack* method and the other the *bringToFront* method on a *Drawing* object. But the rest of the three methods are so different from these two and from each other, that they cannot be moved.

5.3. **Comparative Analysis.** Considering the results obtained after applying ARI and HAC algorithms and illustrated in Tables 1 and 2 we can conclude the following:

- Both algorithms identify most of the existing classes in the JHotDraw framework as correct, internally well-structured classes, which was expected, because JHotDraw is considered an example of good design. Still, both algorithms find possible refactorings, some of them justifiable, some of them not. The HAC algorithm finds only 5 misplaced methods, while ARI finds 9 misplaced methods. For both algorithms, two of the misplaced methods come from the package which contains sample applications, so they do not really count as errors.
- The HAC algorithm correctly identifies 3 possible *Extract Class* refactorings, which are not identified by the ARI algorithm.

The obtained results show that applying an unsupervised learning model (like hierarchical clustering in this paper) would be beneficial, as machine learning models are able to capture hidden patterns in data.

5.4. **Comparison to Related Work.** In this section we aim to compare our method to other methods from literature, presented in Section 2, that use hierarchical clustering to find possible refactorings.

All the methods presented in Section 2, compute the distance between elements using some kind of sets, defined for attributes, methods and, in case of [4], classes. These sets are similar to the relevant properties metric in our approach, but we use four other metrics for the distance, making it more complex.

Compared to [11] and [5], the HAC (and also the ARI) algorithm is capable of identifying three types of refactorings (not just the *Extract Class*). Direct comparison of the results is impossible, because they use different software projects as case study, which might not be available (in [11], a banking application is used, developed by some students), and they do not report the exact results (refactorings), only percentages of correctly identified refactorings.

The closest to our method is the HARS algorithm, in [4]. The difference is, that it uses only the set of relevant properties for computing the distance between two entities, and considered attributes as entities, too. [4] uses JHotDraw as a case study too, and it finds only one out of the three *Extract Class* refactorings that HAC finds (but it finds also two *Move Attribute* refactorings, which neither ARI nor HAC can find).

6. Conclusions and further work

In this paper we presented a new algorithm, HAC, that uses unsupervised learning to find a good restructuring of a software system. Using the JHotDraw framework as a case study, we compared the new algorithm, to an existing one, called ARI, which identifies refactorings, but without using unsupervised learning.

The results of the case study, presented on Tables 1 and 2 show that the HAC algorithm identifies *Extract Class* refactorings, that are not identified by the ARI algorithm. This demonstrates that using unsupervised learning methods (hierarchical clustering in our case) for automatic refactorings identification is beneficial, because it can uncover hidden patterns in data.

Although the introduced algorithm is capable of finding three types of refactorings, in the future we want to increase this number. This can be done by considering the attributes as entities too, thus adding the *Move Attribute* refactoring. Further investigations will be made to extend the vector space model characterising the entities from the software system by identifying other software metrics that are relevant for software restructuring.

References

[1] Nicolas Anquetil and Timothy Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of 6th Working Conference on Reverse Engineering*, pages 235–255, Atlanta, USA, October 1999.

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

[3] Istvan Gergely Czibula and Gabriela Serban. Improving systems design using a clustering approach. *International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[4] Istvan Gergely Czibula and Gabriela Serban. Hierarchical clustering for software systems restructuring. *INFOCOMP Journal of Computer Science*, 6(4):43–51, 2007.

[5] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiu, and Jorg Sander. Decomposing object-orietend class modules using an agglomerative clustering technique. In *Proceedings of International Conference on Software Maintenance*, pages 93–101, Edmonton, Canada, 2009.

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

[7] Erich Gamma. JHotDraw Project. http://sourceforge.net/projects/jhotdraw.

[8] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[9] Sayyed Garba Maisikeli. *Aspect Mining Using Self-Organizing Maps With Method Level Dynamic Software Metrics as Input Vectors*. PhD thesis, Graduate School of Computer and Information Sciences Nova Southeastern University, 2009.

[10] Zsuzsanna Marian, Gabriela Czibula, and Istvan Gergely Czibula. Using software metrics for automatic software design improvement. *Studies in Informatics and Control*, 2012. Submitted for review.

[11] Akepogu Ananda Rao and Kalam Narendar Reddy. Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique. *International Journal of Computer Science Issues*, 8(2):185–194, 2011.

[12] 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.

Department of Computer Science,, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania.

E-mail address: `marianzsu@cs.ubbcluj.ro`