STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 2, 2010

ADAPTIVE RESTRUCTURING OF OBJECT-ORIENTED SOFTWARE SYSTEMS

ISTVAN GERGELY CZIBULA AND GABRIELA CZIBULA

Abstract. In this paper we approach the problem of adaptive refactoring, the process of adapting the class structure of a software system when new application classes are added. We have previously introduced an adaptive clustering method that deals with the evolving structure of any object oriented application. The aim of this paper is to extend the evaluation of the proposed method on the open source case study JHotDraw, emphasizing, this way, the potential of our approach.

1. Introduction

Improving the software systems design through refactoring is one of the most important issues during the evolution of object oriented software systems. Refactoring aims at changing a software system in such a way that it does not alter the external behavior of the code, but improves its internal structure. We have introduced in [1] a clustering approach, named *CARD* (*Clustering Approach for Refactorings Determination*), for identifying refactorings that would improve the class structure of a software system.

But real applications evolve in time, and new application classes are added in order to met new requirements. Consequently, restructuring of the modified system is needed to keep the software structure clean and easy to maintain. Obviously, for obtaining the restructuring that fits the new applications classes, the original restructuring scheme can be applied from scratch on the whole extended system. However, this process can be inefficient, particularly for large software systems. That is why we have proposed in [2] an adaptive method that deals with the evolving application classes set. The proposed method extends our original approach previously introduced in [1].

Received by the editors: January 10, 2009.

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

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

Key words and phrases. Software engineering, refactoring, clustering.

The rest of the paper is structured as follows. Our clustering based approach for adaptive refactorings identification is described in Section 2. For the adaptive process, a *Core Based Adaptive Refactoring* algorithm, named *CBAR*, is proposed. Section 3 indicates several existing approaches in the direction of automatic refactorings identification. An evaluation of $CBAR$ on the open source case study JHotDraw is provided in Section 4. Some conclusions of the paper and further research directions are given in Section 5.

2. Core Based Adaptive Refactoring. Background

In the following we will briefly review our clustering approach from [2] for adapting the class structure of a software system when it is extended with new applications classes.

2.1. **Initial Restructuring Phase.** We have introduced in [1] a clustering approach, named *CARD*, for identifying refactorings that would improve the class structure of a software system. First, the existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system. After data was collected, the set of entities extracted at the previous step are re-grouped in clusters using a clustering algorithm. The goal of this step is to obtain an improved structure of the existing software system. The last step is to extract the refactorings which transform the original structure into an improved one, by comparing the newly obtained software structure with the original one.

For re-grouping entities from the software system, a vector space model based *clustering* algorithm, named *kRED* (*k-means for REfactorings Determination*), was introduced in [1].

In the proposed approach, the objects to be clustered are the elements (called *entities*) from the considered software system, i.e., $\mathcal{S} = \{e_1, e_2, \dots, e_n\}$, where $e_i, 1 \leq i \leq n$ can be an application class, a method of a class or an attribute of a class. Each entity is measured with respect to a set of l features (l representing the number of application classes from $\mathcal{S}$), $\mathcal{A} = \{C_1, C_2, \dots, C_l\}$, and is therefore described by an l-dimensional vector: $e_i = (e_{i1}, e_{i2}, \dots, e_{il}), e_{ik} \in \Re, 1 \leq i \leq n, 1 \leq k \leq l$. The *distance* between two entities e_i and e_j from the software system $\mathcal{S}$ is computed as a measure of dissimilarity between their corresponding vectors, using the *Euclidian distance.*

2.2. **The Adaptive Refactoring Phase.** During the evolution and maintenance of $\mathcal{S}$, new application classes are added to it in order to met new functional requirements. Let us denote by $\mathcal{S}'$ the software system $\mathcal{S}$ after extension. Consequently, restructuring of $\mathcal{S}'$ is needed to keep its structure clean and easy to maintain. Obviously, for obtaining the restructuring that fits the new applications classes, the original restructuring scheme can be applied from scratch, i.e., *kRED* algorithm can be applied considering all entities from the modified software system $\mathcal{S}'$. However, this process can be inefficient, particularly for large software systems.

That is why we have extended the approach from [1] and we have proposed in [2] an adaptive method, named *CBAR* (*Core Based Adaptive Refactoring*), that deals with the evolving application classes set. Namely, the case when new application classes are added to the software system and the current restructuring scheme must be accordingly adapted, was handled. The main idea is that instead of applying *kRED* algorithm from scratch on the modified system $\mathcal{S}'$, we adapt (using *CBAR*) the partition obtained by *kRED* algorithm for the initial system $\mathcal{S}$, considering the newly added application classes.

The extension of the application classes set from $\mathcal{S}$ means that the number of entities in $\mathcal{S}'$ increases, and the vectors characterizing the entities, increase, as well. Therefore, the entities have to be re-grouped to fit the new application classes set. Let us consider that the set $C_1, C_2, \ldots, C_l$ of application classes from $\mathcal{S}$ is extended by adding s ($s \geq 1$) new application classes, $C_{l+1}, C_{l+2}, \ldots, C_{l+s}$. Consequently, the set of attributes will be extended with s new attributes, corresponding to the newly added application classes. The vector for an extended entity $e_i \in \mathcal{S}, 1 \leq i \leq n$ is extended as $e'_i = (e_{i1}, \ldots, e_{il}, e_{i,l+1}, \ldots, e_{i,l+s})$ and a set of new entities $\{e'_{n+1}, e'_{n+2}, \ldots, e'_{n+m}\}$ is added to $\mathcal{S}'$. This set corresponds to the entities from the newly added application classes, $C_{l+1}, C_{l+2}, \ldots, C_{l+s}$.

The *CBAR* method starts from the partitioning of entities from $\mathcal{S}$ into clusters established by applying *kRED* algorithm in the initial restructuring phase. Let $\mathcal{K} = \{K_1, K_2, \ldots, K_l\}$ be the initial clusters (restructured application classes) of $\mathcal{S}$, $K_i \cap K_j = \emptyset, i \neq j, \bigcup\limits_{v=1}^{l} K_v = \mathcal{S}$. *CBAR* determines then $\mathcal{K}' = \{K'_1, K'_2, \ldots, K'_{l+s}\}$ the new partitioning of entities in $\mathcal{S}'$ after application classes set extension. It starts from the idea that, when adding few application classes, the old arrangement into clusters (partition $\mathcal{K}$) can be adapted in order to obtain the restructuring scheme of the extended software system. The algorithm determines those entities within each cluster K_i ($1 \leq i \leq l$) that have a considerable chance to remain together in the same cluster. They are those entities that, after application classes extension, still remain closer to the centroid (cluster mean) of cluster K_i. These entities form what is called the core of cluster K_i, denoted by $Core_i$. We denote by $CORE$ the set of all cluster cores, $CORE = \{Core_1, Core_2, \ldots, Core_l\}$.

The cores of all clusters K_i, $1 \leq i \leq l$, will be new initial clusters from which the adaptive partitioning process begins. The extended partition $\mathcal{K}'$ should also contain initial clusters corresponding to the newly added application classes. Therefore, the centroids corresponding to clusters $K_{l+1}, K_{l+2}, \dots,$ K_{l+s} are chosen to be the newly added application classes.

Next, *CBAR* proceeds in the same manner as *kRED* [1] algorithm does. The *CBAR* algorithm can be found in [2, 3]. As experiments show, the result is reached by *CBAR* more efficiently than running *kRED* again from the scratch on the feature-extended entity set.

3. Related Work

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

Deursen et al. have approached the problem of *refactoring* in [4]. The authors illustrate the difference between refactoring test code and refactoring production code, and they describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

Xing and Stroulia present in [13] an approach for detecting refactorings by analyzing the system evolution at the design level.

A search based approach for refactoring software systems structure is proposed in [9]. 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 [5].

Based on some elementary metrics, the approach in [12] aids the user in deciding what kind of refactoring should be applied.

The paper [11] describes a software vizualization tool which offers support to the developers in judging which refactoring to apply.

Clustering techniques have already been applied for program restructuring. A clustering based approach for program restructuring at the functional level is presented in [14]. This approach focuses on automated support for identifying ill-structured or low cohesive functions. The paper [8] presents a quantitative approach based on clustering techniques for software architecture restructuring and reengineering as well as for software architecture recovery. It focuses on system decomposition into subsystems.

A clustering based approach for identifying the most appropriate refactorings in a software system is introduced in [1].

To our knowledge, there are no existing approaches in the literature in the direction of adaptive refactoring, as we have approached in [2].

4. JHotDraw Case Study

In this section we present an experimental evaluation of *CBAR* algorithm on the open source software JHotDraw, version 5.1 [7].

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. Table 1 gives an overview of the system's size.

Classes	**173**
Methods	**1375**
Attributes	**475**

Table 1. JHotDraw statistic.

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 *CBAR* algorithm described in Subsection 2.2 on JHotDraw.

4.1. Quality Measures.

In the following we will present several measures that we propose for evaluating the obtained results from the restructuring point of view and from the adaptive process point of view, as well.

4.1.1. *Quality measures for restructuring.* In order to capture the similarity of two class structures (the one obtained by a clustering algorithm and the original one) we will use three measures.

Each measure evaluates how similar is a partition of the software system S determined after applying a clustering algorithm (as *kRED* or *CBAR*) with a good partition of the software system (as the actual partition of JHotDraw is considered to be).

In the following, let us consider a software system $\mathcal{S}$ consisting of the application classes set $\{C_1, C_2, \ldots, C_l\}$. We assume that the software system S has a good design (as JHotDraw has) and $\mathcal{K} = \{K_1, \ldots K_l\}$ is a partition reported by a clustering algorithm (as *kRED* or *CBAR*).

Definition 1. [1] ***Accuracy of classes recovery - ACC.***

The accuracy of partition $\mathcal{K}$ with respect to the software system $\mathcal{S}$, denoted by $ACC(\mathcal{S}, \mathcal{K})$, is defined as:

$$ACC(\mathcal{S}, \mathcal{K}) = \frac{1}{l} \sum_{i=1}^{l} acc(C_i, \mathcal{K}), \tag{1}$$

where $acc(C_i, \mathcal{K}) = \dfrac{\sum\limits_{k \in \mathcal{M}_{C_i}} \dfrac{|C_i \cap k|}{|C_i \cup k|}}{|\mathcal{M}_{C_i}|}$. $\mathcal{M}_{C_i}$ *is the set of clusters from* $\mathcal{K}$ *that contain elements from the application class* C_i*, is the accuracy of* $\mathcal{K}$ *with respect to application class* C_i*, i.e.* $\mathcal{M}_{C_i} = \{K_j | \, 1 \leq j \leq l, \; |C_i \cap K_j| \neq 0\}$.

ACC defines the degree to which the partition $\mathcal{K}$ is similar to $\mathcal{S}$. For a given application class C_i, $acc(C_i, \mathcal{K})$ defines the degree to which application class C_i, all its methods and all its attributes were discovered in a single cluster. Based on Definition 1, it can be proved that $ACC(\mathcal{S}, \mathcal{K}) \in [0, 1]$. $ACC(\mathcal{S}, \mathcal{K}) = 1$ iff $acc(C_i, \mathcal{K}) = 1, \forall \; C_i \; 1 \leq i \leq l$, i.e., each application class was discovered in a single cluster. In all other situations, $ACC(\mathcal{S}, \mathcal{K}) < 1$.

Larger values for ACC indicate better partitions with respect to $\mathcal{S}$, meaning that ACC has to be maximized.

Definition 2. ***PRECision of Methods discovery - PRECM.***

The precision of methods discovered in $\mathcal{K}$ *with respect to the software system* $\mathcal{S}$*, denoted by* $PRECM(\mathcal{S}, \mathcal{K})$*, is defined as:*

$$PRECM(\mathcal{S}, \mathcal{K}) = \frac{1}{|nm|} \sum_{m \; is \; a \; method \; from \; \mathcal{S}} precm(m, \mathcal{K}),$$

where $precm(m, \mathcal{K})$ *is the precision of* $\mathcal{K}$ *with respect to the method* m *and* nm *is the number of methods from all the application classes from* $\mathcal{S}$*, i.e.* $precm(m, \mathcal{K}) = \begin{cases} 1 & \textit{if } m \textit{ was placed in the same class as in } \mathcal{S} \\ 0 & \textit{otherwise} \end{cases}$

$PRECM(\mathcal{S}, \mathcal{K})$ defines the percentage of methods from $\mathcal{S}$ that were correctly discovered in $\mathcal{K}$ (we say that a method is correctly discovered if it is placed in its original application class). Based on Definition 2, it can be proved that $PRECM(\mathcal{S}, \mathcal{K}) \in [0, 1]$. $PRECM(\mathcal{S}, \mathcal{K}) = 1$ iff $precm(m, \mathcal{K}) = 1$ for all m, i.e., each method was discovered in its original application class. In all other situations, $PRECM(\mathcal{S}, \mathcal{K}) < 1$.

Larger values for $PRECM$ indicate better partitions with respect to $\mathcal{S}$, meaning that $PRECM$ has to be maximized.

Definition 3. ***PRECision of Attributes discovery - PRECA.***

The precision of attributes discovery in partition $\mathcal{K}$ *with respect to the software system* $\mathcal{S}$*, denoted by* $PRECA(\mathcal{S}, \mathcal{K})$*, is defined as:*

$$PRECA(\mathcal{S}, \mathcal{K}) = \frac{1}{na} \sum_{a \; is \; an \; attribute \; from \; \mathcal{S}} preca(a, \mathcal{K}),$$

where $preca(a, \mathcal{K})$ is the precision of $\mathcal{K}$ with respect to the attribute a and na is the number of attributes from all the application classes from $\mathcal{S}$, i.e. $preca(a, \mathcal{K}) = \begin{cases} 1 & \textit{if a was placed in the same class as in } \mathcal{S} \\ 0 & \textit{otherwise} \end{cases}$

$PRECA(\mathcal{S}, \mathcal{K})$ defines the percentage of attributes from $\mathcal{S}$ that were correctly discovered in $\mathcal{K}$ (we say that an attribute is correctly discovered if it is placed in its original application class). Based on Definition 3, it can be proved that $PRECA(\mathcal{S}, \mathcal{K}) \in [0, 1]$. $PRECA(\mathcal{S}, \mathcal{K}) = 1$ iff $preca(a, \mathcal{K}) = 1$ for all a, i.e., each attribute was discovered in its original application class. In all other situations, $PRECA(\mathcal{S}, \mathcal{K}) < 1$.

Larger values for $PRECA$ indicate better partitions with respect to $\mathcal{S}$, meaning that $PRECA$ has to be maximized.

4.1.2. *Quality measures for the adaptive process.* As quality measures for $CBAR$ algorithm we will consider the number of iterations and the cohesion of the core entities. In other words, we measure how the entities in $Core_j$ $(1 \leq j \leq l)$ remain together in clusters after $CBAR$ algorithm ends.

As expected, more stable the core entities are and more they remain together with respect to the initial sets $Core_j$, better was the decision to choose them as seed for the adaptive clustering process.

We express the *cohesion* of the set of cores $CORE = \{Core_1, \ldots, Core_l\}$ as:

$$(2) \qquad Coh(CORE) = \frac{\sum\limits_{j=1}^{l} \frac{1}{\textit{no of clusters where the entities in } Core_j \textit{ ended}}}{l}$$

The worst case is when each entity from each $Core_j \in CORE$ ends in a different final cluster. The best case is when each $Core_j$ remains compact and it is found in a single final cluster. So, the limits between which $Coh(CORE)$ varies are given below, where the higher the value of $Coh(CORE)$ is, better was the cores choice.

Based on the definition of $Coh(CORE)$, it can be proved that

$$(3) \qquad \frac{1}{l} \leq Coh(CORE) \leq 1.$$

4.2. **Experimental results.** Let us consider JHotDraw system from which we have removed 2 application classes: **StorableInput** and **ColorMap**. We denote the resulting system by $\mathcal{S}$. Therefore, $\mathcal{S}$ consists of 171 application classes ($l = 171$). After applying *kRED* algorithm on $\mathcal{S}$ we have obtained a partition $\mathcal{K}$ in which there were 3 misplaced methods. The names of the methods that were proposed to be moved is shown in the first column of Table 2. The suggested target class is shown in the second column.

Element	Type	Target class
PertFigure.writeTasks	Method	**StorableOutput**
PolygonFigure.distanceFromLine	Method	**Geom**
StandardDrawingView. drawingInvalidated	Method	**DrawingChangeEvent**

TABLE 2. The misplaced elements.

Let now extend $\mathcal{S}$ with the 2 application classes that were initially removed from JHotDraw, **StorableInput** and **ColorMap**. We denote by $\mathcal{S}'$ the extended software system, which, in fact, is the entire JHotDraw system. Consequently, the number of application classes from $\mathcal{S}'$ is 173 ($s = 2$).

There are two possibilities to obtain the restructured partition $\mathcal{K}'$ of the extended system $\mathcal{S}'$.

A. To apply *kRED* algorithm from scratch on the entire JHotDraw system.
B. To adapt, using *CBAR* algorithm, the partition $\mathcal{K}$ obtained after applying *kRED* algorithm before the system's extension.

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

A. After applying *kRED* algorithm for JHotDraw case study ($\mathcal{S}'$), we have obtained a partition $\mathcal{K}'$ characterized by the following quality measures:

- $ACC = 0.9829$.
- $PRECM = 0.997$.
- $PRECA = 0.9957$.

In the partition $\mathcal{K}'$ there were 4 methods and 2 attributes that were misplaced in the partition obtained after applying *kRED* algorithm. The names of the elements (methods, attributes) that were proposed to be moved is shown in the first column of Table 3. The suggested target class is shown in the second column.

B. We have adapted, using *CBAR* algorithm, the partition $\mathcal{K}$ obtained after applying *kRED* algorithm before the system's extension. The partition $\mathcal{K}'$ obtained this way is characterized by the following quality measures:

- $ACC = 0.9721$.
- $PRECM = 0.9949$.
- $PRECA = 0.9957$.

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

Table 3. The misplaced elements.

In the partition $\mathcal{K}'$ there were 7 methods and 2 attributes that were misplaced in the obtained partition. The names of the elements (methods, attributes) that were proposed to be moved is shown in the first column of Table 4. The suggested target class is shown in the second column.

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

Table 4. The misplaced elements.

From Table 4 we can observe that $CBAR$ algorithm determines 3 more refactorings than $kRED$ algorithm:

(i) The *Move Method* refactoring *PertFigure.write* to class *StorableOutput.*
(ii) The *Move Method* refactoring *CompositeFigure.write* to class *StorableOutput.*
(iii) The *Move Method* refactoring *PertFigure.read* to class *StorableInput.*

From our perspective, all these refactorings can be justified. We give below the justification for these refactorings.

(i),(ii) *PertFigure.write* method is responsible with the persistence of an *PertFigure* instance. *PertFigure* class has the following attributes: an instance of *java.awt.Rectangle* and two lists of *Storable* objects. These attributes need to be persisted when a *PertFigure* instance is written. The class *StorableOutput* is responsible for persisting primitive data types (int, boolean) but also contains method for storing *java.awt.Color* and *Storable* objects. Creating a method in class *StorableOutput* that stores a *java.awt.Rectangle* object or a list of *Storable* objects is justified by the fact that the *StorableOutput* class already contains similar methods. Moving these functionalities to the *StorableOutput* class can help to avoid code duplication.

(iii) *StorableInput* class provides generic functionality for reading primitive and user defined data. The instances stored using *StorableOutput* class can be retrieved using *StorableInput* class. The identified refactoring suggests the need for an additional method in *StorableInput* class to handle the retrieval of *java.awt.Rectangle*, list of *Storable* instances. As in the case of *StorableOutput* class, adding this functionality into the *StorableInput* will avoid code duplication.

In our opinion, applying the *Move Method* refactorings suggested at (i), (ii) and (iii) does not extend the responsibilities of the modified classes, but in some situations we may obtain classes with multiple responsibilities. Further improvements will deal with these possible problems.

We comparatively present in Table 5 the results obtained after applying $kRED$ and $CBAR$ algorithms for restructuring the extended system $\mathcal{S}'$.

Table 5. The results

Quality measure	$kRED$ for 173 classes	$CBAR$ for 173 classes
No. of iterations	6	4
ACC	0.9829	0.9721
PRECM	0.997	0.9949
PRECA	0.9957	0.9957
Coh	-	0.8912

From Table 5 we observe the following:

- $CBAR$ algorithm finds the solution in a smaller number of iterations than $kRED$ algorithm. This confirms that the time needed by $CBAR$ to obtain the results is reduced, and this leads to an increased efficiency of the adaptive process.
- The accuracy of the results provided by $CBAR$ are preserved (the additional refactorings identified by $CBAR$ were justified below).
- The choice of the cluster cores in the adaptive process was good enough (the cohesion is close to 1).

5. Conclusions and Future Work

We have extended in this paper the evaluation of $CBAR$ algorithm previously introduced in [2] on the open source case study JHotDraw. $CBAR$ is a clustering based method for adapting the class structure of a software system when new application classes are added to the system. The considered experiment shows the potential of our approach.

Further work will be done in order to isolate conditions to decide when it is more effective to adapt (using *CBAR*) the partitioning of the extended software system than to recalculate it from scratch using *kRED* algorithm. We also plan to improve the method for choosing the cluster cores in the adaptive process and to apply the adaptive algorithm *CBAR* on other open source case studies and real software systems.

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

References

[1] I.G. Czibula and G. Serban. Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[2] Gabriela Czibula, Istvan Gergely Czibula. Adaptive Refactoring Using a Core-Based Clustering Approach. In *Proceedings of SEPADS'2010*, 2010, to be published.

[3] Gabriela Czibula, Istvan Gergely Czibula. Clustering Based Adaptive Refactoring. *Wseas Transactions on Computers*, 2010, to be published.

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

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

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

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

[8] Chung-Horng Lung. Software architecture recovery and restructuring through clustering techniques. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 101–104, New York, NY, USA, 1998. ACM Press.

[9] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM Press.

[10] Frank Simon, Silvio Loffler, and Claus Lewerentz. Distance based cohesion measuring. In *proceedings of the 2nd European Software Measurement Conference (FESMA)*, pages 69–83, Technologisch Instituut Amsterdam, 1999.

[11] 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.

[12] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 183–192, Washington, DC, USA, 2003. IEEE Computer Society.

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

[14] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan. Program restructuring through clustering techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*. pages 75–84. Washington, DC, USA, 2004. IEEE Computer Society.

Babeş-Bolyai University, Department of Computer Science, 1, M. Kogalniceanu Street, Cluj-Napoca, Romania

E-mail address: {`istvanc, gabis`}`@cs.ubbcluj.ro`