Using Software Metrics for Automatic Software Design Improvement

Zsuzsanna Marian, Gabriela Czibula, Istvan Gergely Czibula

Babeş-Bolyai University, 1, M. Kogălniceanu Street, Cluj-Napoca, 400084, Romania,
{marianzsu, gabis, istvanc}@cs.ubbcluj.ro

Abstract: In this paper we are approaching the problem of improving the quality of a software system design, an important issue during the evolution of object oriented software systems. Starting from the fact that software metrics are essential in measuring the software quality, we introduce a metric based high dimensional representation of the elements of a software system (application classes and methods from the application classes) and we define a distance semi-metric between the elements of the software system. An experimental validation of the distance semi-metric on two case studies is provided. The obtained results illustrate that the distance function introduced in this paper can be successfully used for improving the internal structure of software systems, highlighting this way the potential of our proposal.

Keywords: Software Metric, Software Design, Refactoring.

1. Introduction

It is well-known that maintenance and evolution represent important stages in the lifecycle of any software system (about 66% from the total cost of the software systems development). Even if a software product efficiently performs all its specified functions, it is very important to evaluate the software quality. The software product may be hard to understand and difficult to modify and this leads to excessive costs in software maintenance, and these costs are not trivial. Consequently, improving the quality of a software system design is the most important issue during the evolution of object oriented software systems.

Nowadays is essential to produce software with proper quality levels, low content of residual errors, flexible, portable and with high reliability figures [22]. Those targets imply the use of new techniques and models, new metrics and the contribution of software engineering tools [1]. Engineers and analysts frequently need to evaluate the performance of software or a portion of it, such as software modules in the early stages of design or just at the end of the development stage. It is a need to know how complex the design is or if one design is more complex than another [7].

Aims and relevance of our approach

Software metrics [13] are widely used to measure the software quality. In this direction, many tools have been already developed to computing metrics for quality assessment. Many researches have been conducted in order to highlight that software metrics alone are not enough to characterize software quality [19]. To solve this problem, most of advanced validated quality models aggregate software metrics, by computing averages (simple or weighted average) of metrics.

Most of the existing software metrics are computed individually, for each software component, and not for the entire software system [19]. We are studying in this paper how multiple software metrics defined for the components of the software system (in our approach application classes and methods from the application classes) can be used in order to improve the quality of a software system.

Instead of aggregating different software metrics, as in most existing approaches, we are proposing, for each element of a software system, a high dimensional representation, a vector consisting of several relevant software metrics and properties of the element. We are focusing on improving the quality of the software system design, and this should be reflected in the high dimensional representation of the elements from the software system.

In this paper we are considering an element from the software system as being an application class from the system or a method from an application class. Further extensions may increase the granularity level of our approach, also considering the attributes from the application classes, the modules or the software components.

When considering a metric based high dimensional representation of an element from the software system, we have started from the intuition that it would be useful in the following directions.

1. First, to define a distance semi-metric function between the elements of the

software system. This distance function may be used for:

- Identifying the refactorings [10] that would improve the internal structure of the software system, without altering its external behaviour. An automatic and scalable approach for refactorings identification would be very useful in assisting software developers in their daily work of maintaining software systems. The distance semi-metric function may be adapted in order to identify aspect oriented [3] refactorings, also.
- Identifying places in an existing software system where design patterns [11] should be introduced in order to increase the clarity of the system and to facilitate its further evolution.

2. The second direction that can be approached is to use data mining techniques [15] in order to discover relevant patterns and rules in the software system, i.e. high dimensional representation of its elements. Mining the system would be useful in identifying parts of it that are inappropriately defined.

All the above enumerated activities may be important steps in improving the quality of a software system's design and are directions that we want to further research. We are starting our research with investigating the first direction. More exactly, we are focusing in this paper on highlighting how the metric based multi dimensional representation of an element within the software system can be used for improving the internal structure of the software system, by suggesting the appropriate refactorings.

The contributions of this paper are: (i) definition of a metric based high dimensional representation for the elements within a software system, using several software metrics that were adapted to our goal, (ii) definition of a distance semi-metric between the elements of the software system, (iii) experimental validation of the distance semi-metric on two case studies.

The rest of the paper is structured as follows. Section 2 presents an overview on software metrics, also indicating approaches existing in the literature that use software metrics for improving the quality of software systems. Our approach in defining a distance semi-metric between the elements of the software system, based on a high dimensional representation of the constituting elements is introduced in Section 3. Section 4 provides an experimental validation on two case studies of the distance semi-metric function previously defined. An analysis of our approach is given in Section 5. Section 6 presents the conclusion of the paper and outlines directions to improve our approach.

2. Literature Review

In this subsection we will briefly present several approaches existing in the software engineering literature that use software metrics for improving the quality of software systems.

Development of methods for evaluating software quality appears to have first been attempted in an organized way by Rubey and Hartwick [23]. Their method was to define code "attributes" and their "metrics", the former being a prose expression of the particular quality desired of the software, and the latter a mathematical function of parameters thought to relate to or define the attribute.

A later study [4] performed by the authors included the formulation of metrics and their application in a controlled experiment to two computer programs (approximately 400 FORTRAN statements each) independently prepared to the same specification. In this study, only a limited number of attributes were considered.

Roca proposes in [21] a new method for computing software structural complexity based on the entropy evaluation of the random uniform response function associated with the so called Software Characteristic Function SCF. The behavior of the SCF with the different software structures and their relationship with the number of inherent errors is further investigated in [22]. It is also investigated how the entropy concept can be used to evaluate the complexity of a software structure considering the SCF as a canonical representation of the graph associated with the control flow diagram. The experimental evaluation for the metric introduced by Roca in [21], [22] is based on several individual cases and is carried out on five different softwares in order to point out that complexity software structure measured as their SCF entropy is proportional to the number of inherent software errors at the beginning of the debugging.

3. Methodology

In this section we introduce a distance semi-metric between the elements of a software system, based on a high dimensional representation of the constituting elements.

In the following we consider a software system S to be a set $S = \{s_1, s_2, ..., s_n\}$, where s_i, $1 \le i \le n$ can be an application class from S or a method from an application class. An element $s \in S$ is called an *entity*.

Further extensions of our approach will consider not only the application classes, methods from them, but also attributes from the application classes, modules or software components.

3.1 The vector space model

As we have already mentioned, software metrics are used as range of measurements applied to software systems with the intent of improving the systems [20]. Software measures (metrics) are indicators describing complexity of software products and processes. By their very nature, software metrics describe a number of complex and high dimensional data patterns that are able to provide different characteristics of the software systems under investigation. Such measures are very useful in investigating and quantifying key properties of the systems such as reliability, readability and maintainability.

In our approach we are focusing on characterizing each entity from the software system by a list of relevant features, features that would represent discriminative characteristics for the methods and application classes of the software system. The key features that we are using for characterizing an entity from the software system are: Relevant properties (RP), Depth in Inheritance Tree (DIT), Number of Children (NOC), Fan-In (FI) and Fan-Out (FO). Further improvements of our approach will consider other software metrics that are relevant in order to decide the quality of the internal structure of a software system.

In the following we will briefly describe the above mentioned five components of the vector that will be used to characterize an entity from the software system.

Vector Component 1. Relevant Properties (RP). For a given entity $s \in S$, the first component *rp(s)* of the vector associated to *s* represents a set of relevant properties of *s* and is defined as:

- If *s* is a method, then *rp(s)* consists of: the method itself, the application class where the method is defined, all attributes from *S* accessed by the method, all the methods from *S* used by method *s*, and all methods from *S* that overwrite method *s*.
- If *s* is an application class, then *rp(s)* consists of: the application class itself, all attributes and methods defined in the class, all interfaces implemented by class *s* and all classes extended by class *s*.

In considering the relevant properties of an entity as a component of the vector space model, we have started from the intuition that the set of relevant properties would be useful in defining the dissimilarity between two entities from *S* (by considering the properties shared by the entities).

Vector Co mponent 2. D epth in Inheritance Tree(DIT). Depth in Inheritance Tree [5], or DIT, is usually applied for a class and, as its name suggests, it simply represents the depth of a given class in the inheritance tree. In [17] DIT is presented as a value ranging from 0 to a positive integer. This meant that the DIT of the root class has the value 0. In our implementation the DIT of the root class is 1, although if we take into consideration that in Java every class has as parent the Object class then we could say that the DIT of the Object class is 0.

Consequently, for a given entity $s \in S$, the second component *dit(s)* of the vector associated to *s* represents the DIT value, and is a natural number defined as:

- If *s* is an application class, then *dit(s)* is the depth of the application class in the inheritance tree of the software system *S* (ignoring any classes that are not defined in *S*, for example the *java.lang.Object* class). Application classes at the top of the tree have a DIT value of 1.
- If *s* is a method, then *dit(s)* is the DIT value of the application class *s* belongs to.

Vector Component 3. Number of Children (NOC). The Number of Children, or NOC metric is one of the metrics presented in [5], [6] and it is designed specifically for object oriented systems. Based on the original definition in [5], this metric can be applied to an object or class,

and it represents the number of direct descendants of a class in the class hierarchy.

Consequently, for a given entity $s \in S$, the third component *noc(s)* of the vector associated to *s* represents the NOC value, and is a natural number defined as:

- If *s* is an application class, then *noc(s)* is the number of application classes that are direct children of *s* in the inheritance tree of the software system *S*.
- If *s* is a method, then *noc(s)* is the NOC value of the application class *s* belongs to.

Vector Components 4 and 5. Fan-In (FI) and Fan-Out (F O): Both metrics were defined as complexity measures in [16] and both of them were originally meant for modules or procedures. As object-oriented metrics started to appear, the definition of fan-in and fan-out was slightly modified, so that they could be applied to classes, instead of modules. In a simple definition fan-in of a class is the number of classes that reference that class, while fan-out is the number of classes referenced by the class.

In [18], the fan-in and the fan-out of a method is presented. The definition says that fan-in is the number of distinct method bodies that invoke a method and the fan-out is the number of times a method invokes another methods.

No matter what it is defined for, usually a large fan-in means a module/class/method that does simple things that are needed often in other places, while a large fan-out usually means a large module/class/method, that does many things. Having both a large fan-in and a large fan-out, might be the sign of a poor design.

In our approach, we considered as fan-in of a class *C*, the number of attributes of type *C* in other classes plus the number of methods in any other class except *C* that use an object of type *C*. The fan-out of a class *C* is defined as the number of distinct classes that have attributes in class *C* - including primitive types - plus the number of distinct classes that are used in any method of class *C*.

Consequently, for a given entity $s \in S$, the fourth component *fi(s)* of the vector associated to *s* represents the Fan-In value, and is a natural number defined in the following way:

- If *s* is an application class, then *fi(s)* is the number of application classes in the software system *S* that use *s*. This includes application classes that have an attribute of type *s*, or have a method that uses class *s*.
- If *s* is a method, then *fi(s)* is the number of other methods in the software system *S* that invoke method *s*. When computing *fi(s)*, only invocations from methods from different classes than the class of *s* are considered.

For the entity $s \in S$, the fifth component *fo(s)* of the vector associated to *s* represents the Fan-Out value, and is a natural number computed as follows:

- If *s* is an application class, then *fo(s)* is the number of distinct classes from software system *S* that *s* uses. Application class *s* uses application class s' if *s* has a field of type s', or if a method of *s* calls a method of s'.
- If *s* is a method, then *fo(s)* is the number of methods *s* calls.

3.2 The distance function

Considering the vector space model defined above, each entity s_i, $1 \le i \le n$ from the software system *S* will be represented as a 5-dimensional vector: $(s_{i1}, s_{i2}, s_{i3}, s_{i4}, s_{i5})$ where s_{ik} $(\forall 1 \le k \le 5)$ represents the value (scaled to [0,1]) of the software metric *k* (RP, DIT, NOC, FI, FO) applied to the entity s_i.

Consequently, as in a vector space model, the distance $d(s_i, s_j)$ between two entities s_i and s_j of the software system will be defined as the dissimilarity degree between the corresponding vectors, an adaptation of the *Euclidian distance*. This distance is expressed in Formula (1). We consider that if two entities have no common relevant properties, their distance is ∞, meaning that two entities are not cohesive enough and should not belong to the same application class. For generality purposes, we will consider that the dimension of the vector associated to an entity s_i from the software system is *m*, i.e. $s_i = (s_{i1}, s_{i2}, ..., s_{im})$ where $s_{i1} = rp(s_i)$ represents the set of relevant properties associated to s_i and s_{ik} $(\forall 2 \le k \le m)$ represents the value (scaled to [0,1]) of a software metric applied to the entity s_i. In our approach *m* = 5 and the considered software metrics are RP, DIT, NOC, FI, FO, i.e.

$s_{i2} = dit(s_i)$, $s_{i3} = noc(s_i)$, $s_{i4} = fi(s_i)$, $s_{i5} = fo(s_i)$.

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

Considering the dissimilarity measure as expressed in Equation (1) it is very likely that the distance *d* will assign low distances to cohesive entities that have to belong to the same application class.

4. Experimental Validation

In order to experimentally validate the distance semi-metric function introduced in Subsection 3.2, we will focus in this section on showing how it can be used for automatically identifying a list of refactorings that would be useful for improving the internal structure of a software system.

4.1. Automatic refactorings identification

Considering that we can visualize a software system *S* as a set of entities (Section 3), a possible structure of the software system *S* can be viewed as a ***partition*** of it. A cluster (group of entities) from this partition corresponds to an application class from the software system.

Our idea is to identify from the possible partitions of the set *S* the partition that reflects a good internal structure of the software system. This partition will be identified based on the distance semi-metric introduced in Section 3 using an algorithm that will be further explained. More exactly, a *partition* $K = \{K_1, K_2, \ldots, K_p\}$ of *S* is provided, each cluster K_i from the partition K representing an application class in the new structure of the software system. The goal of this re-partitioning is to obtain an improved structure of the existing software system. If the newly obtained software structure is compared with the original structure of *S*, a list of refactorings, which transform the original structure into an improved one, can be provided.

In order to identify the partition K of the software system that would correspond to an improved structure of it, we proceed as indicated below.

First, the software system *S* is analyzed in order to extract from it information needed to compute the relevant properties of the entities, as well as the software metrics indicated in Subsection 3.1. Using the collected data, the high dimensional representation of the entities from *S* is constructed.

The second step is to build the partition *K*. After the partition is initialized with the empty partition, the construction of *K* is made as follows. For each method $m \in S$, the following three steps are performed:

1. We search for the application class $c \in S$ that is closest to m (considering the distance function d given in Formula (1)), i.e. $d(m,c) < d(m,c') \forall c' \in S, c'$ application class, $c \neq c'$.
2. At this step possible *Move Method* refactorings may be identified. If the method *m* and the class *c* found at the previous step are cohesive (have common relevant properties), i.e. $d(m,c) < 1$, there are two possibilities:

– If the class *c* found at the previous step appears in a cluster *k* from the partition *K* (a cluster corresponding to the application class *c* was already created), the method *m* is added to the cluster *k*.

– If there is no cluster in the partition that contains the class *c*, a new cluster consisting of the class *c* and the method *m* is added to the partition *K*. The goal of this step is to dispose *m* into the application class that is closest to it.

3. If the distance between the method *m* and all the application classes from *S* is ∞, it means that *m* should not belong to any of the existing application classes from *S*, and should be extracted or added in a new cluster (this cluster will represent a new application class that has to be created in *S* - an *Extract Class* refactoring). More exactly, from the set K' of clusters that do not contain application classes existing in *S* (i.e. represent new application classes that have to be added in *S*) we are searching for the method m' that is closest to *m*. There are two situations:

– If $K' = \phi$ or $d(m,m') \geq 1$ (is ∞), a new cluster (application class) containing the

method *m* has to be created and added to the partition *K*.

– If the method m' that is closest to *m* was found in K' and is cohesive with *m* (i.e. $d(m,m')<1$), then method *m* is added into the cluster that contains m'. This means that *m* will be placed in the newly extracted application class in which its closest method belongs.

After the partition *K* was created, we are searching this partition in order to identify possible *Inline Class* refactorings that would be appropriate. More exactly, $\forall C, C' \in S$ application classes from *S*, for which $d(C,C')<1$ (i.e. *C* and C' are cohesive), we merge in the partition *K* the two clusters that contain *C* and C'. This merging operation means in fact that application classes *C* and C' have to be merged in a single application class.

We mention that the refactorings that the algorithm that was introduced above is able to identify are: *Move Method*, *Inline Class* and *Extract Class* [10].

4.2. Results

In order to experimentally validate our proposal for refactorings identification based on the distance semi-metric defined between the entities from the software system, we will consider two case studies, which are described in the following subsections.

Each of the systems evaluated in Subsections 4.2.1 and 4.2.2 are written in Java. In order to extract from the systems the data needed to compute the high dimensional representation of the entities, we use ASM 3.0 [2]. ASM is a Java bytecode manipulation framework. We use this framework in order to extract the structure of the systems (attributes, methods, classes and relationships between all these entities).

4.2.1 First Case Study: The first case study, described in this subsection, contains a simple example to provide the reader with an easy to follow example of refactorings identification.

Let us consider the Java code example shown below. The example is taken from [25] and is used in order to illustrate the *Move Method* refactoring.

```
public class Class_A {
  public static int attributeA1;
  public static int attributeA2;
  public static void mA1(){
     attributeA1 = 0;
     mA2();
  }
  public static void mA2(){
        attributeA2 = 0;
        attributeA1 = 0;
    }
  public static void mA3(){
        attributeA2 = 0;
        attributeA1 = 0;
        mA1();
        mA2();
  }
}

public class Class_B {
  private static int attributeB1;
  private static int attributeB2;
  public static void mB1(){
        Class_A.attributeA1=0;
        Class_A.attributeA2=0;
        Class_A.mA1();
    }
  public static void mB2(){
        attributeB1=0;
        attributeB2=0;
    }
  public static void mB3(){
        attributeB1=0;
        mB1();
        mB2();
  }
}
```

As we have indicated in Section 3, the software system described above is represented as a set S = {**Class_A**, **mA1**(), **mA2**(), **mA3**(), **Class_B**, **mB1**(), **mB2**(), **mB3**()}. The actual structure of the system illustrated in the code example below can be viewed as the partition K^1 of *S*, $K^1=\{K_1^1, K_2^1\}$, where the cluster K_1^1 contains the entities from the application class **Class_A**, and the cluster K_2^1 contains the entities from the application class **Class_B**:

– K_1^1={Class_A, mA1(), mA2(), mA3()}

– K_2^1={Class_B, mB1(), mB2(), mB3()}

Analyzing the code presented above, it is obvious that the method **mB1**() has to belong to **Class_A**, because it uses features of **Class_A** only. Thus, the refactoring *Move Method* [10] should be applied to this method, as indicated in [25].

Table 1. The distances between the entities.

	Class_A	Class_B	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()
Class_A	0	∞	0.329	0.397	0.528	0.471	∞	∞
Class_B	∞	0	∞	∞	∞	0.482	0.555	0.239
mA1() 0.32	**9**	∞	0	0.361	0.453	0.406	∞	∞
mA2() 0.39	**7**	∞	0.361	0	0.596	0.464	∞	∞
mA3() 0.52	**8**	∞	0.453	0.596	0	0.444	∞	∞
mB1() 0.47	**1**	0.482	0.406	0.464	0.444	0	0.474	0.471
mB2()	5.49	**0.555**	∞	∞	∞	0.474	0	0.567
mB3()	5.5	**0.239**	∞	∞	∞	0.471	0.567	0

We show, in the following, how using the algorithm described in Subsection 4.1, the *Move Method* refactoring mentioned above is successfully identified.

As indicated in Subsection 4.1, the first step of our approach is to collect data from the system and compute the multi dimensional representation of the entities from the software system. The high dimensional representation of the entities from the system will be further used to compute the distance *d* (Formula (1)) between the entities. These distances (with three decimals) are given in Table 1. In this table, for each method we highlight the class that it is closest to.

From Table 1 the following can be observed:

- The methods **mA1**(), **mA2** (), **mA3**() are closer to **Class_A** than to **Class_B**. These means that the methods are correctly placed in their initial application class **Class_A**.
- The methods **mB2**(), **mB3**() are closer to **Class_B** than to **Class_A**. This means that the methods are correctly placed in their initial application class **Class_B**.
- The method **mB1**() is closest to **Class_A** than to **Class_B**. These means that the method should be placed in **Class_B**, instead of **Class_B**, and this will lead to a *Move Method* refactoring.
- The distance between **Class_A** and **Class_B** is ∞, meaning that no *Inline Class* refactoring will be reported.
- All methods have to be placed in an existing application class, meaning that no *Extract Class* refactoring will be reported.

Consequently, after applying the algorithm introduced in Subsection 4.1, the newly obtained partition of the system is $K_2^2 = \{K_1^2, K_2^2\}$, where the cluster K_1^2 contains the entities from the restructured application class **Class_A** (also containing the method **mB1**()), and the cluster K_2^2 contains the entities from the restructured application class **Class_B** (without the method **mB1**()):

- K_1^2={**Class_A**, **mA1**(), **mA2**(), **mA3**(), **mB1**()}
- K_2^2={**Class_B**, **mB2**(), **mB3**()}

This way, the *Move Method* refactoring **mB1**() to **Class_A** is correctly identified (as indicated in [25]).

4.2.2 Secon d Case Study: Our second evaluation is the open source software JHotDraw, version 5.1 [12]. 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 how accurate are the results obtained after applying the algorithm introduced in Subsection 4.1 in comparison to the current design of JHotDraw. As JHotDraw has a good class structure, our algorithm should generate a nearly identical class structure.

After applying our algorithm for JHotDraw case study, only *Move Methods* refactorings were identified. In the obtained partition, 20 *Move Method* refactorings were obtained: 9 refactorings were incorrectly identified, and other 11 refactorings may be justified, in our view (these 11 methods suggested to be moved provide functionalities similar to the ones provided by the target class [12], so, in our view, all these refactorings are acceptable, without altering the internal structure of JHotDraw). The names of the methods that were incorrectly

Table 2. The incorrectly reported *Move Method* refactorings for JHotDraw

	Method Target	Class
1	DrawApplet.toolDone	ToolButton
2	DrawApplication.createColorMenu	ColorMap
3	DrawApplet.setupAttributes	ColorMap
4	DrawApplet.createColorChoice	ColorMap
5	DrawApplication.createEditMenu	CommandMenu
6	DrawApplication.createAlignmentMenu	CommandMenu
7	DrawApplication.selectionChanged	CommandMenu
8	PertFigure.asInt	NumberTextFigure
9	PertFigure.setInt	NumberTextFigure

proposed to be moved is shown in the first column of Table 2. The suggested target class is shown in the second column.

5. Discussion

In this section we aim at providing an analysis of the approach presented in this paper. The effectiveness of the distance semi-metric function introduced in Section 3 in the process of automatic refactorings identification will be shown below by comparing it with similar approaches for which the obtained results are publicly available.

Seng et al. apply in [24] a weighted multi-objective search [14], in which metrics are combined into a single objective function. An heterogeneous weighed approach is applied here, since the weight of software entities in the overall system and refactorings cost are studied. This approach partially gives the results obtained on JHotDraw. The advantages of our approach in comparison with the approach presented in [24] are as follows. In the technique from [24] there are **10** misplaced methods, while in our approach there are only **9** misplaced methods. Our technique is deterministic, in comparison with the approach from [24]. The evolutionary algorithm from [24] is executed **10** times, in order to judge how stable are the results. The overall running time for the technique from [24] is about **300** minutes (30 minutes for one run), while our approach provides the results in about **3** minutes (the execution was made on similar computers).

The paper [25] describes a software visualization tool which offers support to the developers in judging which refactoring to apply. The authors have considered for evaluation a simple example that was presented in Subsection 4.2.1. On this example, the refactoring proposed by the algorithm introduced in this paper coincides with the one given in [25].

In [8] an approach that uses clustering for improving the class structure of a software system is introduced. In this direction, a partitional clustering algorithm, *kRED* was developed. Unlike the *kRED* algorithm, our algorithm also identifies the Extract Class refactoring.

Considering the results presented above, we can conclude that our approach in defining a distance semi-metric based on a high dimensional representation of the entities from a software system has potential and further improvements can lead to valuable results.

6. Conclusions and Future Work

We have introduced in this paper a distance semi-metric function between the elements (application classes and methods from the application classes) of a software system, distance defined using a metric based high dimensional representation of the constitutive elements. An experimental evaluation of this distance semi-metric is provided, illustrating that it can be successfully used for improving the internal structure of software systems by identifying appropriate refactorings.

Further work will be done in order to investigate other software metrics that would be relevant to decide the quality of the internal structure of a software system and to consider other principles related to an improved design, like: *Single Responsibility Principle, Open-Closed Principle, Interface Segregation Principle, Common Closure Principle* [9]. We will also focus on increasing the granularity level of the proposed approach, also considering attributes from the application

classes, modules or software components. Further experiments on real software systems will be considered for validating the introduced distance function.

REFERENCES

1. ASAM, R., N. DRENKARD, H. MAIER, **Qualitatsprufung von Software-produkten**, Siemens AG Verlag, Berlin, 1986.
2. **ASM**: (ObjectWeb: Open Source Middleware) http://asm.objectweb.org/.
3. BABAMIR, S. M., **Active Program Analysis using Rule-based Modification and Aspecta tion**, Studies in Informatics and Control, vol.20 (4), 2011, pp. 381-392.
4. BROWN, J. R., M. LIPOW, **The Quantitative Measurement of Software Safety and Reliability** , Technical Report TRW Report No. SDP-1776, TRW Software Series, 1973.
5. CHIDAMBER, S. R., C. F. KEMERER, **Towards a Metrics S uite for Object-oriented Design** , Conference Proceedings on Object Oriented Programming Systems, Languages, and Applications, 1991, pp. 197-211.
6. CHIDAMBER, S. R., C. F. KEMERER, **A Metrics Su ite for Object Orie nted Design**, IEEE Transactions on Software Engineering, no. 20, 1994, pp. 476-493.
7. CURTIS, B., S. B. SHEPPARD, P. MILLIMAN, M. A. BORST, T. LOVE, **Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics** , IEEE Transactions on Software. Engineering, no. 5, 1979, pp. 96-104.
8. CZIBULA, I., G. SERBAN, **Improving Systems D esign using a Cluste ring Approach**, International Journal of Computer Science and Network Security, no. 6, 2006, pp. 40-49
9. DEMARCO, T., **Structured Analysis and System Spec ification**, Software pioneers: contributions to software engineering, 2002, pp. 529-560
10. FOWLER, M., **Refactoring: Im proving the De sign of Existing Code** , Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
11. GAMMA, E., R. HELM, R. JOHNSON, J. VLISSIDES, **De sign Patterns: Abstraction and Reu se of Object-oriented Design**, 2002, pp. 701-717.
12. GAMMA, E., **(JHotDraw P roject)** http://sourceforge.net/projects/jhotdraw.
13. GRADY, R. B., **Practical Software Metrics for Project M anagement and Process I mprovement**, Prentice Hall Press, 1992.
14. GZARA, M., A. ESSABRI, **Balanced Explore-Exploit Clustering based Distributed Evolutionary Algorithm for Multi-objective Optimi sation**, Studies in Informatics and Control, no. 20, 2011, pp. 97-106.
15. HAN, J., **Data Mining: Concepts and Techniques**, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
16. HENRY, S., D. KAFURA, **Software Structure M etrics based on Information Flow**, IEEE Transactions on Software Engineering, no. 7, 1981, pp. 510-518.
17. LI, W., S. HENRY, **Object Ori ented Metrics whi ch Predi ct Maintainability**, Journal of Systems and Software, no. 23, 1993, pp. 111-122.
18. MAISIKELI, S.G., **Aspect Mining using Self-Organizing Maps with Method Level Dynamic Softwa re Metri cs as Input Vectors** , PhD thesis, Nova Southeastern University, 2009.
19. MORDAL-MANET, K., J. LAVAL, S. DUCASSE, N. ANQUETIL, F. BALMAS, F. BELLINGARD, L. BOUHIER, P. VAILLERGUES, T. J. MCCABE, **An Empirical Model for Continuous and Weighted Metric Aggregation** , Proceedings of CSMR '11, Washington, DC, USA, IEEE Computer Society, 2011, pp. 141-150.
20. PRESSMAN, S., **Software Engineeri ng: A Practitioner's Approach**, McGraw-Hill Education, 2005.
21. ROCA, J. L., **An Entropy-based Method for Compu ting Software Struct ural Complexity**, Microelectronics and Reliability, no. 36, 1996, pp. 609-620.

22. ROCA, J. L., **A New Entropy based Method for Compu ting Software Structural Complexity**, Technical Report ARN-PI-2002-8, Autoridad Regulatoria Nuclear, Buenos Aires, 2002.

23. RUBEY, R. J., R. D. HARTWICK, **Quantitative Measurem ent of Program Quality**, Proceedings of ACM '68, New York, NY, USA, ACM, 1968, pp. 671-677.

24. SENG, O., J. STAMMEL, D. BURKHART, **Search-based Determination of Re factorings for Improving the Clas s Structu re of Object-oriented Syste ms**: Proceedings of GECCO '06, New York, ACM Press, 2006, pp. 1909-1916.

25. SIMON, F., F. STEINBRUCKNER, C. LEWERENTZ: **Metrics ba sed Refactoring**: Proceedings of CSMR '01, Washington, DC, USA, IEEE Computer Society, 2001, pp. 30-38.