Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 233 (2009) 43–57

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Clustering for Monitoring Software Systems Maintainability Evolution

P. Antonellis[a] D. Antoniou[a] Y. Kanellopoulos[b] C. Makris[a] E. Theodoridis[a] C. Tjortjis[b] N. Tsirakis[a]

[a] *{adonel,antonid,makri,theodori,tsirakis}@ceid.upatras.gr*
Department of Computer Engineering and Informatics, University Of Patras, Greece

[b] *Yiannis.Kanellopoulos@postgrad.manchester.ac.uk, christos.tjortjis@manchester.ac.uk*
School Of Computer Science, The University Of Manchester, UK

Abstract

This paper presents ongoing work on using data mining clustering to support the evaluation of software systems' maintainability. As input for our analysis we employ software measurement data extracted from Java source code. We propose a two-steps clustering process which facilitates the assessment of a system's maintainability at first, and subsequently an in-cluster analysis in order to study the evolution of each cluster as the system's versions pass by. The process is evaluated on Apache Geronimo, a J2EE 1.4 open source Application Server. The evaluation involves analyzing several versions of this software system in order to assess its evolution and maintainability over time. The paper concludes with directions for future work.

Keywords: evaluation,software, maintainability,data mining

1 Introduction

Software maintenance is considered as the most difficult stage in software lifecycle. According to the National Institute of Standards and Technology (NIST), it costs the U.S. economy $60 billion per year [24]. Given this high cost, maintenance processes can be considered as an area of competitive advantage. There are several studies for evaluating a system's maintainability and controlling the effort required to carry out maintenance activities [10], [11], and [28].

The scope of this work is to facilitate maintenance engineers to comprehend a software system and evaluate its evolution and maintainability. Questions that can be answered are which classes are fault prone and more difficult to understand and maintain; how a system evolves from version to version, what are the dynamics of a system's classes through time and others.

For this reason we present a methodology which employs the clustering mining technique for the analysis of software measurement data. The k-Attractors algo-

doi:10.1016/j.entcs.2009.02.060

rithm which is tailored for software measurement data was used for this purpose [17]. The proposed methodology consists of two steps. At the first, each version of a software system is analyzed separately in order to evaluate its maintainability. The second step comprises a macro-clustering analysis which investigates the derived clusters from all the versions of a software system. The aim of this step is to study a system's evolution by observing how clusters from each version grow up or shrink and how their centroids are moving in space from version to version. We attempt to evaluate the usefulness of our work using Apache Geronimo Application Server, an open source server used in real life industrial applications. The remaining of this paper is organized as follows. Section 2 reviews existing work in the area of data mining for program comprehension and evaluation. Section 3 outlines the steps of the whole process and the rationale for each of them. Section 4 assesses the accuracy of the output of the proposed methodology, analyses its results and outlines deductions from its application. Finally, conclusions and directions for future work are presented in section 5.

2 Related Work

Data mining [18], is the process which extracts implicit, previously unknown, and potentially useful information from data, by searching large volumes of them for patterns and by employing techniques such as classification, association rules mining, and clustering. It is a quite complex topic and has links with multiple core fields such as computer science, statistics, information retrieval, machine learning and pattern recognition. Its ability to deal with vast amounts of data has been considered a suitable solution in assisting software maintenance, often resulting in remarkable results [9], [18], [19], [22], and [29]. As previous studies have shown, data mining is capable to obtain useful knowledge about the structure of large systems.

More specifically, data mining has been previously used for identification of subsystems based on associations (ISA methodology) [11]. Sartipi et al. used it for architectural design recovery [27]. They proposed a model for the evaluation of the architectural design of a system based on associations among system components and used system modularity measurement as an indication of design quality and its decomposition into subsystems. Besides association rules, the clustering data mining technique has been used to support software maintenance and software systems knowledge discovery [30], [26]. The work in [26] proposes a methodology for grouping Java code elements together, according to their similarity and focuses on achieving a high level system understanding.

Understanding low/medium level concepts and relationships among components at the function, paragraph or even line of code level by mining C and COBOL legacy systems source code was addressed in [25]. For C programs, functions were used as entities, and attributes defined according to the use and types of parameters and variables, and the types of returned values. Then clustering was applied to identify sub-sets of source code that were grouped together according to custom-made similarity metrics [25]. An approach for the evaluation of clustering in dynamic dependencies is presented in [31]. The scope of this solution is to evaluate the usefulness of providing dynamic dependencies as input to software clustering algorithms. Additionally, Clustering over a Module Dependency Graph (MDG) [20] uses a collection of algorithms which facilitate the automatic recovery of the modular structure of a software system from its source code. This method creates a hierarchical view of system architecture into subsystems, based on the components and the relationships between components that can be detected in source code.

Moreover, [19] presented an approach that examines the evolution of code stored in source control repositories. This technique identifies Change Clusters, which can help managers to classify different code change activities as either maintenance or new development. On the other hand, [29] analyzes whether some change coupling between source code entities is significant or only minor textual adjustments have been checked in; in order to reflect the changes to the source code entities. An approach for analyzing and classifying change types based on code revisions has been developed. In addition, Beyer and Noack [13] presented a method based on clustering Software artifacts, in order to organize software systems into subsystems and by this way make changes less expensive and less error prone. Towards the same goal of comprehending large software systems by creating abstractions of the software system's structure, Mitchell and Mancoridis [16]presented the Bunch clustering system. In this work, clustering is implemented by search techniques and is performed on graphs that represent the system's structure. The subsystems are generated by partitioning a graph of entities and relations. Another approach in the context of software clustering is the Limbo algorithm, introduced by Tzerpos and Andritsos [8]. This scalable hierarchical algorithm focuses on minimizing the information loss when clustering a system, by applying weighting schemes that reflect the importance of each component.

Clustering algorithms are also used by Mancoridis et al. [21] in order to support the automatic recovery of the modular structure of a software system from its source code. The algorithms selected in this case are traditional hill-climbing and genetic algorithms. Towards program comprehension, a crucial step is detecting important classes of the system, since they implement the most basic and high level actions. Zaidman et al [32] introduced four static web-mining and coupling metrics in order to identify such classes and generally analyze a software system.

The work presented in this paper differs from the literature discussed above in means of performing clustering on the software measurement data, aiming at comprehending a software system and assessing its maintainability. More specifically, instead of applying clustering algorithms on graphs or directly on the source code, we employ the k-Attractors clustering algorithm on metrics that reflect the most important design aspects of a software system concerning its quality and maintainability. We employ a two-steps clustering analysis in order to provide a quick and rough grasp of a software system and depict its evolution by from version to version.

3 Clustering Analysis

3.1 Objectives

The primary objective of the proposed clustering methodology is to provide a general but illuminating view of a software system that may lead engineers to useful conclusions concerning its maintainability. This data mining technique is useful for Similarity/Dissimilarity analysis; in other words it analyzes what data points are close to each other in a given dataset. This way, mutually exclusive groups of classes are created, according to their similarities and hence the system comprehension and evaluation is facilitated. Thus, maintenance engineers are provided a panoramic view of a system's evolution, which helps them in revising the system's maintainability, studying the classes' behavior from version to version and discovering programming patterns and "unusual" or outlier cases which may require further attention.

In order to extract useful information for the maintenance engineers through the clustering analysis, it is very interesting to observe the form of each cluster over time. How each cluster grows ups or shrinks and how its median is moving in space. In order to achieve that, a first task to be performed is the identification of each cluster in each version. An approach is to combine all the data sets (the data points corresponding to classes) into a large data set. Each point is marked with a different color in order to disentangle them later on. If we apply a clustering algorithm in this data set (k-Attractors in our case) we can make the assumption that a cluster will encompass data items of the same cluster through the versions. In each of these clusters will exist the same data items with different color and thus from different version. We can verify this by inventing an inner metric: the percentage of data points that exist in the cluster with all the possible colors (or a percentage respectively of them, for example 3 out of 5 of the versions). There are several ways to exploit this clustering by automated methods: we can trace the data items that have escaped the cluster and examine if they have gone to a better or a worse cluster, by examining in each cluster the sub-clusters, each one with a different color, and how their centroid is moving and the portion of their spatial overlap. By these panoramic observations, the sequence of centroids and proportion of the overlap, we can see if the data items of the corresponding cluster evolve to better or a worse state.

In order to quantify the cluster changes we define a metric m(i) of each cluster i which expresses how many variations, data items (from version to version) exist in the same cluster at the same time, and thus in the same quality space. This metric is expressed by the following formula:

$$m(i) = \frac{\sum_{\forall x \in C_i} \sum_{j=1}^{n} ocj(x)}{\sum_{j=1}^{j} p_j}$$

Where n is the number of the formed clusters, $occ(x_i)$ is the number of occurrences of each data item x in cluster i, and pi is the cardinality (population) of cluster i.

3.2 k-Attractors Algorithm

For this purpose the k-Attractors algorithm was employed which is tailored for numerical data like measurements from source code [17]. The main characteristics of k- Attractors are:

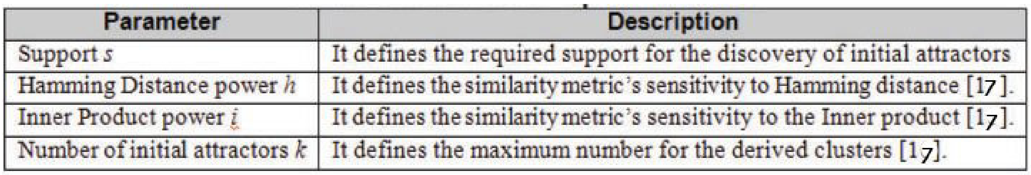
- It defines the desired number of clusters (i.e. the number of k), without user intervention.
- It locates the initial attractors of cluster centers with great precision.
- It measures similarity based on a composite metric that combines the Hamming distance and the inner product of transactions and clusters' attractors.

The k-Attractors algorithm employs the maximal frequent itemset discovery and partitioning in order to define the number of desired clusters and the initial attractors of the centers of these clusters. The intuition is that a frequent itemset in the case of software metrics is a set of measurements that occur together in a minimum part of a software system's classes. Classes with similar measurements are expected to be on the same cluster. The term attractor is used instead of centroid, as it is not determined randomly, but by its frequency in the whole population of a software system's classes. The main characteristic of k-Attractors is that it proposes a similarity measure which is adapted to the way initial attractors are determined by the preprocessing method. Hence, it is primarily based on the comparison of frequent itemsets. More specifically, a composite metric based on the Hamming distance and the dot (inner) product between each transaction and the attractors of each cluster is utilized. The two basic steps of the k-Attractors algorithm are:

- Initialization phase:
- The first step of this phase is to generate frequent itemsets using the APriori algorithm. The derived frequent itemsets are used to construct the itemset graph, and a graph partitioning algorithm is used to find the number of the desired clusters and assign each frequent itemset into the appropriate cluster.
- As soon as the number of the desired clusters (k) is determined, we select the maximal frequent itemsets of every cluster, forming a set of k frequent itemsets as the initial attractors.
- Main Phase:
- As soon as the attractors have been found, we assign each transaction to the cluster that has the minimum Score(Ci← tj) against its attractor.
- When all transactions have been assigned to clusters we recalculate the attractors for each cluster in the same way as during the initialization phase.

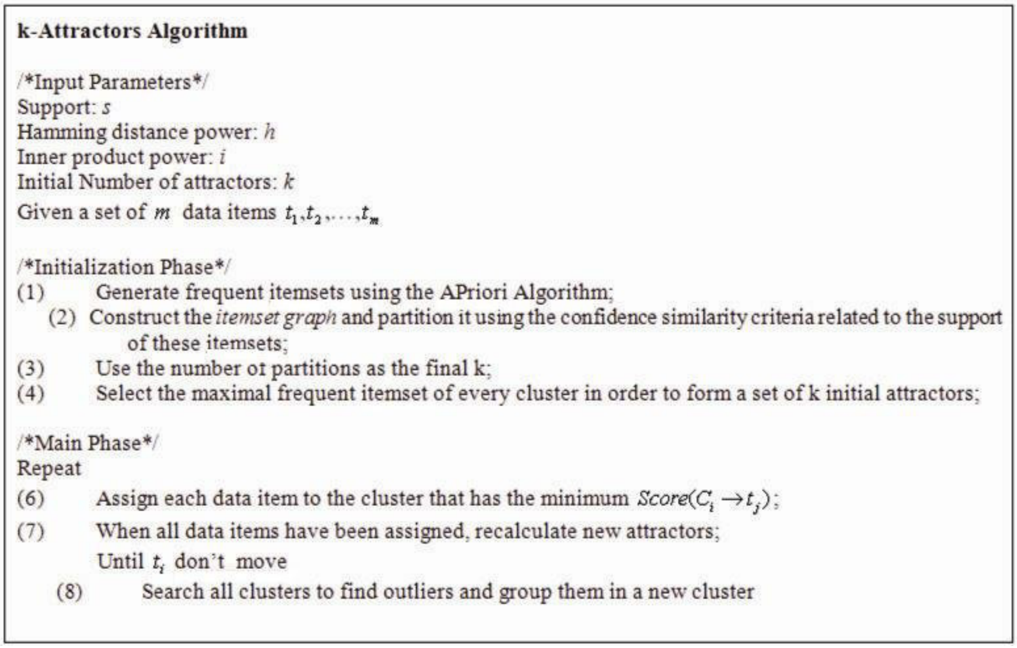
The k-Attractors algorithm utilizes a hybrid similarity metric based on vector representation of both the data items and the cluster's attractors. The similarity of these vectors is measured employing the following composite metric:

$$Score(C_i \leftarrow t_j) = h * H(a_i, t_j) + i * (a_1 * t_1 + \ldots a_n * t_n$$



Parameter	Description
Support s	It defines the required support for the discovery of initial attractors
Hamming Distance power h	It defines the similarity metric's sensitivity to Hamming distance [17].
Inner Product power i	It defines the similarity metric's sensitivity to the Inner product [17].
Number of initial attractors k	It defines the maximum number for the derived clusters [17].

Fig. 1. k-Attractors Input Parameters



```
k-Attractors Algorithm

/*Input Parameters*/
Support: $s$
Hamming distance power: $h$
Inner product power: $i$
Initial Number of attractors: $k$
Given a set of $m$ data items $t_1, t_2, \ldots, t_m$

/*Initialization Phase*/
(1)    Generate frequent itemsets using the APriori Algorithm;
  (2)  Construct the itemset graph and partition it using the confidence similarity criteria related to the support
          of these itemsets;
(3)    Use the number of partitions as the final k;
(4)    Select the maximal frequent itemset of every cluster in order to form a set of k initial attractors;

/*Main Phase*/
Repeat
(6)    Assign each data item to the cluster that has the minimum $Score(C_i \to t_j)$;
(7)    When all data items have been assigned, recalculate new attractors;
       Until $t_i$ don't move
   (8)    Search all clusters to find outliers and group them in a new cluster
```

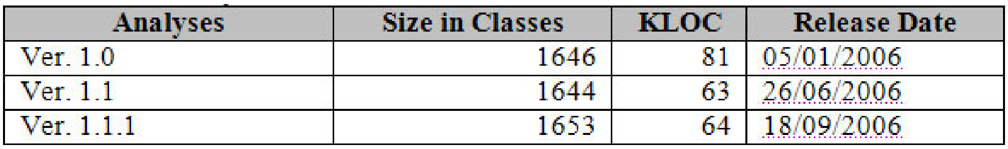
Fig. 2. k-Attractors Overview

In this formula, the first term is the Hamming distance between the attractor and the data item . It is given by the number of positions that pair of strings is different and is defined as follows:

$$H(a_i, t_j) = n - \#(a_i \cap t_j)$$

As the algorithm is primarily based on itemsets' similarity, we want to measure the number of substitutions required to change one into the other. The second term is the dot (inner) product between this data item and the attractor . It is used in order to compensate for the position of both vectors in the Euclidean space. Because of the semantics of software measurement data, the usually utilized internal metrics (such as lines of code, coupling between objects, number of comments etc) have large positive integer values. Thus in order for the inner product distance to be more accurate, we firstly normalize all the values in the interval [-1, 1] and then apply the k-Attractors algorithm.

The multipliers in equation (2) define the metric's sensitivity to Hamming distance and inner product respectively. For example, the case indicates the composite metric is insensitive to the inner product between the data item and the cluster's centroid. Both and i are taken as input parameters in our algorithm during its



Analyses	Size in Classes	KLOC	Release Date
Ver. 1.0	1646	81	05/01/2006
Ver. 1.1	1644	63	26/06/2006
Ver. 1.1.1	1653	64	18/09/2006

Fig. 3. Apache Geronimo size in classes and in lines of code

execution. Thus, k-Attractors provides the flexibility of changing the sensitivity of the composite distance metric to both Hamming distance and inner product, in correspondence with the each clustering scenario's semantics.

4 Results Assessment

The evaluation of the proposed methodology involved the study of Apache Geronimo Application Server. It is a fully certified J2EE 1.4 platform for developing and deploying Enterprise Java applications, Web applications and portals. Three publicly available versions of Apache Geronimo [33] were evaluated employing a set of software evaluation metrics and their analysis using the k-Attractors clustering algorithm. Figure 3 presents its size measured in classes and in lines of code.

4.1 Data Extraction and Preparation

The objective of data extraction and preparation was two-fold:

- At first to collect appropriate elements that describe the software architecture and its characteristics. These elements include native source code attributes and metrics.
- Then to analyze the collected elements, choose a refinement subset of them and store them in a relational database system for further analysis.

Native attributes include classes, methods, fields etc. Metrics, on the other hand, provide additional system information and describe more effectively the system's characteristics and behaviour.

All the collected data (i.e. attributes and metrics) are stored into appropriate structured XML files. We have chosen XML because of its interoperability and its wide acceptance as a de facto standard for data representation and exchange. Storing the metrics in XML files enables further processing and analysis with a variety of tools. The basic unit of every XML file is the java-source-program. Every java-source-program is associated with a java-class-file and can include a package declaration (package-decl), a list of imported modules (import), a list of interfaces (interface) and a list of classes (class). For every such subunit, the XML file stores detailed information. For example, for every class we store its superclass, a list of its member fields (field), a list of its methods (methods) and its constructors (constructor) and finally the associated metrics.

For simplicity, we chose to analyse a refinement subset of the most important collected elements. This subset should be small enough in order to be easily analyzed and large enough to contain all the necessary system information. Based on this requirement, we stored and further analyzed only the metrics and their associated native attributes.

4.2 Experimental Datasets

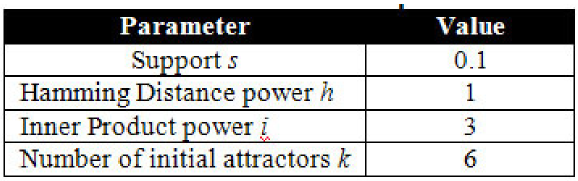
For our experiments we combined a size metric (i.e. Lines of Code) and two sets of metrics proposed by [14] and [12]. The derived set can be applied to OO programs and can be used as a predictor and evaluator of a system's maintenance effort [16]. The following metrics were included and calculated for the systems' classes and were used as their clustering attributes:

- Lines of Code (LOC), which measures a class's number of lines of code including empty lines and comments.
- Weighted Methods per Class (WMC), which is simply the sum of the complexities of its methods [14].
- Coupling between Objects - Efferent Coupling (CBO), which represents the number of classes a given class, is coupled to [14].
- Lack of Cohesion in Methods (LCOM), which measures if a class has all its methods working together in order to achieve a single, well-defined purpose [14].
- Number of Children (NOC), which measures the number of immediate descendants of the class [14].
- Depth of Inheritance Tree (DIT), which provides for each class a measure of the inheritance levels from the object hierarchy top [14].
- Data Access Metric (DAM), which reflects how well the property of encapsulation is applied to a class [12].
- Measure of Aggregation (MOA), which measures the extent of the part-whole relationship realized by using attributes [12].
- Number of Polymorphic Methods (NOP) that is a measure of the overridden (or virtual) methods of an object oriented software system [12].
- Number of Messages (NOM), which is a measure of the services that a class provides [12].

4.3 k-Attractors tuning

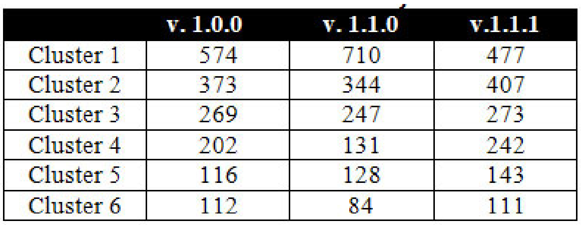
We utilized k-Attractors in order to form 6 clusters for every version of Apache Geronimo. The input parameters that were used are presented in Figure 8. We chose to use the same parameters in all the three clustering operations in order for the formed clusters to be consistent in all versions.

Additionally we have chosen to give more weight to the inner product in relation to Hamming distance, because of the dissimilar values of every module metric which result in a high Hamming distance for the most data items. Thus we set i=3 and h=1. The support for the Apriori algorithm was set to 0.1, as we wanted to consider only itemsets with at least 10% frequency in the original dataset. Finally, the number of initial attractors k was set to 6, as we wanted to form 6 clusters for every



Parameter	Value
Support *s*	0.1
Hamming Distance power *h*	1
Inner Product power *i*	3
Number of initial attractors *k*	6

Fig. 4. k-Attractors input values



	v. 1.0.0	v. 1.1.0	v.1.1.1
Cluster 1	574	710	477
Cluster 2	373	344	407
Cluster 3	269	247	273
Cluster 4	202	131	242
Cluster 5	116	128	143
Cluster 6	112	84	111

Fig. 5. Clusters cardinality

version of the Apache Geronimo Application Server.

4.4 Geronimo Application Server Evaluation

We utilized k-Attractors with the previously described input parameters in order to form 6 clusters for every version of the Apache Geronimo Application Server. Figure 5 presents the cardinality for every formed cluster:

The main characteristics of Apache Geronimo's formed clusters are:

- Clusters 1, 2 and 3 have the biggest population (more than 70% of the whole population) in every version and contain classes that their measurement values are low and therefore it is easier to understand and maintain them. These clusters can be labeled as "good" clusters.
- Clusters 4 and 5 contain classes with increased values which indicate that they need further inspection and effort in order to remain maintainable in future versions of Apache Geronimo. We can characterize these clusters as "under inspection".
- Cluster 6 has the lowest population and it contains classes that exhibit exceptional measurement values. These classes are considered as outliers of the Apache Geronimo. The cluster that contains them can be labeled as "bad" cluster.

Figure 6 presents the distribution of complexity (WMC) among different clusters in all the three versions of Apache Geronimo Application Server. Due to space limitations we don't include histograms for the rest of the utilized metrics. In the following sections we give a brief description of the special characteristics of each group of formed clusters, derived from observing the distribution of all the metrics on those clusters through the 3 versions of Geronimo.

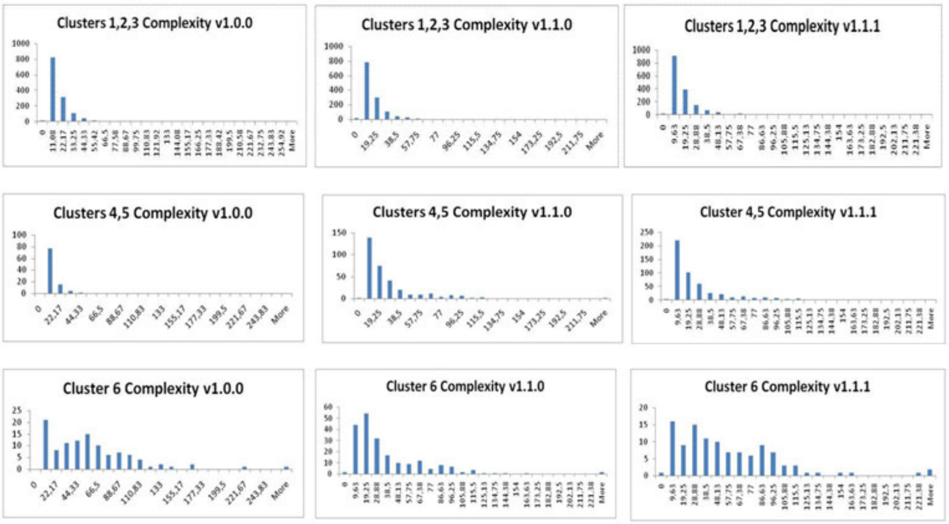


Fig. 6. Complexity of formed clusters through different versions

4.4.1 Analysis of Clusters 1, 2 and 3 (”Good Clusters”)

Clusters 1, 2 and 3 are considered to be the ”good” clusters in the evaluation of Apache Geronimo’s maintainability. In version 1.1.0 the classes of clusters 1, 2 and 3 seem to be more maintainable than those in version 1.0.0 and version 1.1.1 as the respective metrics are more close to 0 (on X-axis). For example in Figure 6, it is obvious that the complexity distribution is closer to 0 (on X-axis) for version 1.1.0 than for the other versions. Another observation is that all these clusters in version 1.1.1 are starting to move away from 0 (on X-axis) and this indicates that the classes are becoming less maintainable

4.4.2 Analysis of Clusters 4 and 5 (”Under Inspection”)

Clusters 4 and 5 contain classes with values that indicate they need further attention and inspection, in order to remain maintainable. From version 1.0.0 to version 1.1.0 the classes of clusters 4 and 5 are becoming significantly less maintainable as the measurements concerning their complexity, coupling and lack of cohesion increase. The same behavior (low maintainability) stands also for the version 1.1.1. For example, if we consider the Figure for clusters 4 and 5, we observe that in version 1.0.0 the distribution of complexity is closer to 0 (on X-axis) comparing to version 1.1.0 and version 1.1.1.

4.4.3 Analysis of Cluster 6 (Outliers - ”Bad Clusters”)

Cluster 6 contains those classes that exhibit exceptional values in their measurements. Consequently these classes are the most difficult to understand and maintain and may require possible refactoring to improve their design. If we take a look at Figure 6, it is obvious that in all 3 versions of Geronimo, the average distribution of complexity for cluster 6 is far from 0 (on X-axis), thus indicates a low maintainability for the corresponding classes. A good example is classes CdrOutputStream

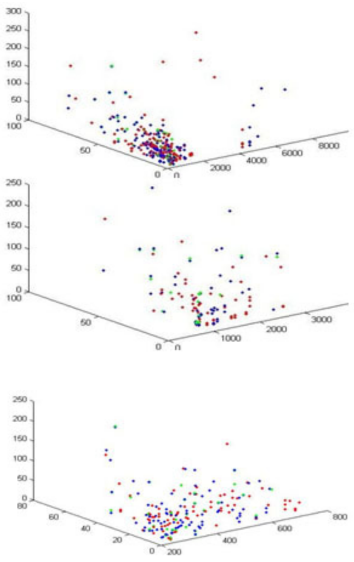


Fig. 7. Cluster 1, 2, 3 through versions (red=v 1.0, green=v 1.1,blue=v 1.1.1)

and CdrInputStream which are used for streaming objects in Corba Common Data Representation format. These classes are used fairly widely within the application server, for, among others, serializing non-primitive data structures, hence the high complexity values. They should be of interest to the maintenance engineers, since they are at Geronimo's core and widely used, so for maintainability and runtime performance they will be important classes. Classes KernelManagementHelper and MockGBean can also be interesting from a maintenance engineer's perspective.

4.4.4 Classes' Changes

The performed clustering analysis provided us also the capability to trace those classes that have moved the cluster that were assigned in version 1.0 and examine if they have gone to a better or worse one in the succeeding versions. A very good example of a class moving to a "good" cluster is RefContext which in version 1.0 was in cluster 4 (LOC=318, NPM=26 DIT=0, NOC=0, NOM=101, LCOM = 100, RFC=101, CBO = 23, MOA=56 and WMC = 56) but in the following versions moved to clusters 3 (LOC=91, NPM=10, DIT=0, NOC=0, NOM=27, LCOM = 45, RFC=27, CBO = 16, MOA=0 and WMC = 21) and 2 (LOC=91, NPM=10, DIT=0, NOC=0, NOM=0, LCOM = 45, RFC=27, CBO = 16, MOA=0 and WMC = 19) respectively. On the other hand now, an example of a class moving to a worse cluster is class AbstractWebModuleBuilder which moved from cluster 1 (LCOM = 0, CBO = 3 and WMC = 1) to cluster 5 (LCOM = 394, CBO = 41 and WMC = 74), an "under inspection" cluster, in both 1.1 and 1.1.1 versions.

As Figures 7 and 8 depict, each cluster consists of three sub-clusters, one for each version (red = version 1.0, green = version 1.1, blue = version 1.1.1). We can observe that many data items are shifted in space from version to version, while a concrete core remains in the same space. For the clusters of figures 7, 8 the m(i) values are 0.5395, 0.5439, 0.4444, 0.8108 and 0.4769 of each cluster respectively.

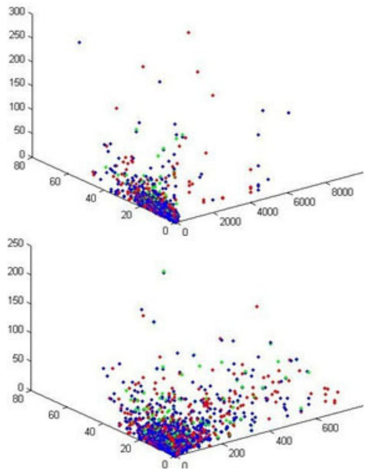


Fig. 8. Cluster 4 , 5 through versions (red=v 1.0, green=v 1.1, blue=v 1.1.1)

Hence, the forth cluster is the most concrete. It lies at the beginning of the axes and seems to stay the same through the versions. All the other clusters either tend to exchange a quite large number of items or a number of their data items have been omitted. By observing how each cluster's centroids are shifted in space from version to version we can have an overview of their evolution. For example for the cluster 5 (which is depicted in figure 8) the distance of the centroid of the version 1.0 to the 1.1 is d1 = 28.2452 while the distance of the 1.1 to the 1.1.1 is d2 = 25.3733. This core cluster is quite solid. The corresponding distances of cluster 2 of figure 6 are d1 = 403.5479 and d2 = 406.9371. In the first cluster the evolution is discretional while in the second more significant from the first version to the second and small from the second to the third.

5 Conclusions and Future Work

In this research work, the development of a methodology based on the clustering data mining technique was presented. It consists of two steps: i. a separate clustering step for every version of a system to assist software system's evaluation in means of maintainability. ii. a macro-clustering analysis in order to study the system's dynamics from version to version. The scope of the proposed methodology is to facilitate maintenance engineers to identify classes which are fault prone and more difficult to understand and maintain as well as to study the evolution of a system from version to version, and its classes' dynamics. We chose to employ the k-Attractors clustering algorithm as it is tailored for the analysis software measurement data [17]. Our work is different than [21], which employs clustering in order to produce a high-level organization of the source code. Additionally, instead of applying clustering algorithms on [23] or directly on the source code [21], we clustered software metrics that reflect the most important aspects of a system concerning its quality and maintainability. Moreover the study of the classes' evolution through versions differentiates this work from [32] which only detects the most important classes on a single version of the system.

The proposed methodology was tested on Apache Geronimo, a J2EE 1.4 open source Application Server. In the first step of the analysis we created overviews for each version of Apache in order to have an indication for their maintainability status. Then by studying the formed clusters for each version, we discovered classes which were fault prone. Those classes were members of the outlier clusters and examples are CdrOutputStream and CdrInputStream. In the second step, the macro-clustering analysis we traced classes that their quality was either degraded or upgraded. Such classes are RefContext and AbstractWebModuleBuilder. Our findings indicate that the proposed methodology has considerable merit in facilitating maintenance engineers to monitor how a system's maintainability evolves. On the other hand though, it lacks the ability to predict the maintainability of an upcoming version of a system. Another data mining technique with prediction capabilities (such as classification) could be additionally employed in order to enhance our methodology. Moreover and apart from this, we consider the following various alternatives in order to further develop the proposed methodology:

Systems' components clustering based on their dynamic dependencies It would be of great interest to attempt to evaluate the usefulness of analysing the dynamic dependencies of a software system's artefacts.

Employ an alternative approach for monitoring cluster changes from version to version Another approach for monitoring cluster changes is to perform the clustering procedure for each one of the versions. We use all these clusters (each one with a different color according to the version that belongs) in a second clustering phase, using the corresponding centroids, in order to produce clusters of clusters in a hierarchical way. We can assume that each one of the level two clusters consists of the same cluster of data item through versions. *Enhance the Extraction Method* The proposed method processes information derived only from Java source code files (*.java). It is of great interest to extract data from other languages like C++, C and COBOL which were used for the development of the majority of legacy systems, a category of software systems which is very interesting in terms of program comprehension and maintainability.

Acknowledgement

This research work has been partially supported by the Greek General Secretariat for Research and Technology (GSRT) and Dynacomp S.A. within the program "P.E.P. of Western Greece Act 3.4".

References

[1] Civin, P., and B. Yood, *Involutions on Banach algebras*, Pacific J. Math. **9** (1959), 415–436.

[2] Clifford, A. H., and G. B. Preston, "The Algebraic Theory of Semigroups," Math. Surveys **7**, Amer. Math. Soc., Providence, R.I., 1961.

[3] Freyd, Peter, Peter O'Hearn, John Power, Robert Tennent and Makoto Takeyama, *Bireflectivity*, Electronic Notes in Theoretical Computer Science **1** (1995), URL: http://www.elsevier.nl/locate/entcs/volume1.html.

[4] Easdown, D., and W. D. Munn, *Trace functions on inverse semigroup algebras*, U. of Glasgow, Dept. of Math., preprint 93/52.

[5] Roscoe, A. W., "The Theory and Practice of Concurrency," Prentice Hall Series in Computer Science, Prentice Hall Publishers, London, New York (1198), 565pp. With associated web site http://www.comlab.ox.ac.uk/oucl/publications/books/concurrency/.

[6] Shehadah, A. A., "Embedding theorems for semigroups with involution, " Ph.D. thesis, Purdue University, Indiana, 1982.

[7] Weyl, H., "The Classical Groups," 2nd Ed., Princeton U. Press, Princeton, N.J., 1946.

[8] Andritsos, P. and Tzerpos, V. "Information-Theoretic Software Clustering". IEEE Trans. Software Eng. vol. 31(2), 2005, pp. 150-165

[9] Anquetil, N. and Lethbridge, T. C. "Experiments with Clustering as a Software Remodularization method", Proc. 6th Working Conf. Reverse Engineering (WCRE 99), IEEE Comp. Soc. Press, 1999, pp. 235-255.

[10] Arisholm, E., Briand, L. C. and Foyen, A. "Dynamic Coupling Measurement for Object-Oriented Software", IEEE Transactions on Software Engineering, vol. 30, No. 8, August 2004, pp. 491-506.

[11] Bandi, R. K., Vaishnavi, V. K. and Turk, D. E. "Predicting Maintenance Performance Using Object Oriented Design Complexity Metrics", IEEE Transactions on Software Engineering, vol. 29(1), January 2003, pp. 77-87.

[12] J. Bansiya, C.G Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment", IEEE Transactions on Software Engineering, 28: pp. 4-19, 2002.

[13] Beyer, D. and Noack, A. "Clustering software artifacts based on frequent common changes". In Proc. IWPC, IEEE, 2005, pp. 259-268.

[14] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):pp. 476-493, 1994

[15] Dunham, M. H. Data Mining: Introductory and Advanced Topics. Prentice Hall PTR, 2002.

[16] Kan, S. H. Metrics and Models in Software Quality Engineering. Addison-Wesley. Second Edition. 2002.

[17] Y. Kanellopoulos, P. Antonellis, C. Tjortjis, C. Makris, "k-Attractors, A Clustering Algorithm for Software Measurement Data Analysis", In Proceedings of IEEE 19th International Conference on Tools for Artificial Intelligence (ICTAI 2007), IEEE Computer Society Press 2007

[18] Kunz, T. and Black, J. P. "Using Automatic Process Clustering for Design Recovery and Distributed Debugging", IEEE Transactions on Software Engineering, vol. 21(6), 1995, pp. 515-527,

[19] Lawrie, D. J., Feild, H. and Binkley, D. "Leveraged Quality Assessment using Information Retrieval Techniques," 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 149-158.

[20] Mancoridis, S., Mitchell, B.S., Chen, Y. and Gansner, E.R. "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures", Proc. Int'l Conf. Software Maintenance (ICSM 99), 1999, pp.50-59.

[21] Mancoridis, S., Mitchell, B. S., Rorres, C. "Using Automatic Clustering to Produce High-Level System Organizations of Source Code", (1998) IEEE Proceedings of the 1998 Int. Workshop on Program Understanding (IWPC'98), 1998

[22] Maqbool, O., Babri, H.A., Karim, A. and Sarwar, M. "Metarule-guided association rule mining for program understanding, Software", IEEE Proceedings, vol. 152(6) , 2005, pp. 281- 296.

[23] Mitchell, B. S. and Mancoridis, S. "On the Automatic Modularization of Software Systems Using the Bunch Tool". IEEE Trans. Software Eng., vol. 32(3), 2006, pp. 193-208

[24] National Institute of Standards and Technology (NIST), "The Economic Impacts of Inadequate Infrastructure for Software Testing", Washington D.C. 2002.

[25] Oca, C. M. de and Carver, D. L. "Identification of Data Cohesive Subsystems Using Data Mining Techniques", Proc. Int'l Conf. Software Maintenance (ICSM 98), IEEE Comp. Soc. Press, (1998) 16-23.

[26] Rousidis, D. and Tjortjis, C. ”Clustering Data Retrieved from Java Source Code to Support Software Maintenance: A Case Study”, Proc IEEE 9th European Conf. Software Maintenance and Reengineering (CSMR 05), IEEE Comp. Soc. Press, (2005) 276-279.

[27] Sartipi, K., Kontogiannis, K. and Mavaddat, F. ”Architectural Design Recovery Using Data Mining Techniques”, Proc. 2nd European Working Conf. Software Maintenance Reengineering (CSMR 00), 2000, pp. 129-140.

[28] Tan, Y., Mookerjee, V. S. ”Comparing Uniform and Flexible Policies for Software Maintenance and Replacement”, IEEE Transactions on Software Engineering, vol. 31(3), March 2005, pp. 238-255.

[29] Tjortjis C., Sinos, L. and Layzell, P. J. ”Facilitating Program Comprehension by Mining Association Rules from Source Code”, Proc. IEEE 11th Int’l Workshop Program Comprehension (IWPC 03), 2003, pp. 125-132.

[30] Tzerpos, V. and Holt, R. ”Software Botryology: Automatic Clustering of Software Systems”, Proc. 9th Int’l Workshop Database Expert Systems Applications (DEXA 98), 1998, pp. 811-818.

[31] Xiao, C. and Tzerpos, V. ”Software Clustering on Dynamic Dependencies”, Proc. IEEE 9th European Conf. Software Maintenance and Reengineering (CSMR 05), 2005, pp. 124-133.

[32] Zaidman, A., Du Bois, B. and Demeyer, S. ”How Webmining and Coupling Metrics Improve Early Program Comprehension.” ICPC, 2006, pp. 74-78

[33] http://geronimo.apache.org/downloads.htm