Software Clustering Using Dynamic Analysis and Static Dependencies

Chiragkumar Patel[1], Abdelwahab Hamou-Lhadj[2], Juergen Rilling[1]
[1]*Department of Computer Science and Software Engineering*
[2]*Department of Electrical and Computer Engineering*
Concordia University
1455 de Maisonneuve West
Montreal, Quebec H3G 1M8, CANADA
{c_pat, abdelw, rilling}@encs.concordia.ca

Abstract

Decomposing a software system into smaller, more manageable clusters is a common approach to support the comprehension of large systems. In recent years, researchers have focused on clustering techniques to perform such architectural decomposition, with the most predominant clustering techniques relying on the static analysis of source code. We argue that these static structural relationships are not sufficient for software clustering due to the increased complexity and behavioural aspects found in software systems. In this paper, we present a novel software clustering approach that combines dynamic and static analysis to identify component clusters. We introduce a two-phase clustering technique that combines software features to build a core skeleton decomposition with structural information to further refine these clusters. A case study is presented to evaluate the applicability and effectiveness of our approach.

Keywords: Software clustering, architecture recovery, program comprehension, software maintenance

1. Introduction

Architectural knowledge, an important factor in many software maintenance activities, supports the reuse of architectural components, definition of high-level product families, impact analysis, and estimation of costs and risks associated with modification requests [12, 21]. Due to incomplete, inconsistent, and even non-existing documentation, software engineers often find themselves in a situation where the only reliable source for extracting relevant architectural information is the source code. However, the sheer volume and complexity of source code makes this extraction process inherently difficult. Software clustering techniques were introduced to ease the comprehension process by decomposing the system into smaller, more manageable clusters [4, 9, 18, 22, 26].

Existing clustering techniques [2, 9, 20] can be grouped into two main categories [1]. The first category, also the most commonly used, relies on source code to extract relations between system components. Examples of such relations include file inclusion, routine calls, type references, etc. However, many of these clustering approaches are based on static analysis of the source code and therefore are very limited/conservative in analyzing the dynamic interactions among system entities. The second category of clustering approaches is based on less formal artifacts, such as file names [2], comments [17], etc. These techniques rely on existing design conventions, which make them impractical if design conventions are not followed by software engineers. In addition, extracting knowledge from informal sources of information is known to be a difficult task due to the existence of noise in the data.

In this paper, we propose a novel two-phase clustering approach that combines both dynamic (trace based) and static dependency analysis. The first phase consists of building the core skeleton decomposition of the system. We achieve this by using software features as a clustering criterion. The core skeleton will contain the components that implement these features. The next iteration consists of clustering the non-core components by analyzing their static dependencies with the already formed clusters.

In the context of our research, we consider a system's classes as the entities to cluster, since each class typically is represented by one source file. The ultimate goal of the proposed clustering approach is to group system classes based on their behavioral characteristics (similarity measurement) rather than only relying on their mere structural relationships.

The approach presented is a continuation of our previous work [3], where we discussed how software features can be used to build a core skeleton decomposition of the system. In this paper, we extend this work to introduce a complete hybrid approach that combines static and dynamic analysis for clustering.

More precisely, the contributions of this paper are:

- We introduce software features as a clustering criterion to construct the skeleton decomposition of the system.

- We present a complete hybrid clustering approach of clustering system entities using dynamic and static dependency analysis.
- A case study that shows the applicability of this work.

The remainder of this paper is organized as follows. Section 2 presents our clustering approach. A case study and a discussion of the results are presented in Section 3. In Section 4, we compare our work with related work. Finally, we conclude the paper and present future work in Section 5.

2. Approach

The approach proposed in this paper emphasizes the use of both dynamic and static analysis techniques to support software clustering. It compasses two main steps. In the first step, we measure the similarity between system entities by identifying the number of features they implement. We achieve this by analyzing the execution traces generated from exercising these features.

We define a software feature as any specific scenario of a system that is triggered by an external user. This is similar to the concept of scenarios as defined in UML [10], except that we do not distinguish between primary and exceptional scenarios in this paper. However, it is advisable to include at least the primary scenario while executing a feature. The reason is that primary scenarios tend to correspond to the most common program execution associated with a particular feature.

The motivation behind using software features as a clustering criterion comes from the fact that after working with many feature traces, one could observe that software features constitute a natural grouping of the components that implement them. In fact, one can argue that an effective clustering of low-level system components should start by clustering more abstract concepts such as software features and then locate the components that implement these high-level clusters. The difficulty with this approach is the ability to locate the components that implement specific features - a research topic that has been the subject of many studies (e.g., [30]). Another reason for using features is that they represent *abstract* concepts and, as such, we believe that they can readily be used to recover *abstract* components from low-level implementation details.

In the next step, we apply static dependency analysis to further cluster the system by adding non core components to the skeleton structure. We achieve this using the orphan adoption algorithm presented by Tzerpos et al. [24], which is an algorithm that is based on analyzing static dependencies among components in order to determine the skeleton cluster to which they should belong.

2.1. Skeleton Decomposition Using Software Features

Figure 1 illustrates the steps involved in the first iteration of our clustering approach, which is the skeleton creation based on the software features.

Feature Selection: In this step, we select the software features that will be used during the formation of the skeleton decomposition. The selection of software features can be guided by domain experts or based on any available documentation such as user guides.

We want to note that our approach is affected by the coverage achieved by the features used to exercise the system and to create the traces. However, exercising all possible software features is impractical. This problem can be addressed by identifying a threshold value corresponding to the number of features one needs to execute to achieve a satisfactory level of system decomposition. The threshold itself can be based on the code coverage achieved.

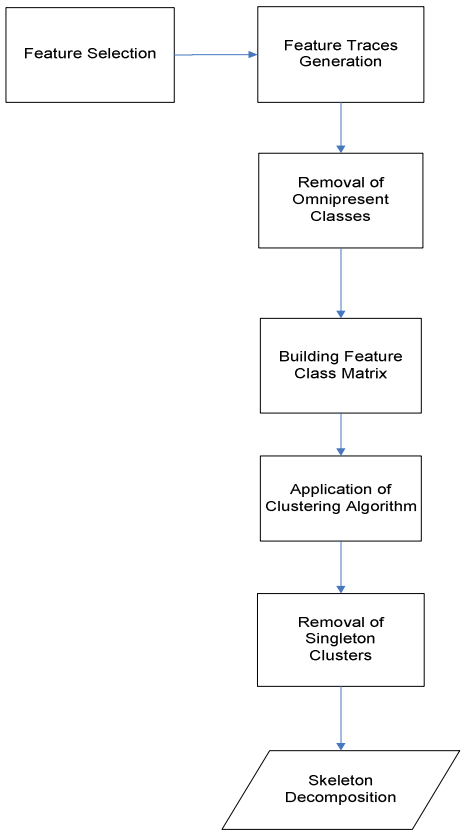

Figure 1. Phase 1: Skeleton decomposition approach

It is also important to select the features while considering a wide range of functionality supported by the system. If one only chooses features that represent similar functions then the resulting skeleton will most likely end up with a few clusters that contain most of the entities.

Therefore, it is necessary to derive a balanced set of features that cover various aspects of the system. In this paper, we use any available documentation to derive a balanced set of software features.

Feature-Trace Generation: In this step of our approach, a trace for every selected feature is generated by executing the instrumented version of a system. Source code instrumentation consists of inserting probes at the location of interest in either the source or the byte code of the system. This is usually done automatically. There are other ways of generating traces including instrumenting the execution environment such as the Java Virtual Machine. A debugger can also be set to collect event of interest. However, the use of a debugger considerably slows down the execution of the system and should be avoided for large-scale software systems [28]. We use the term *feature trace* to refer to a trace that corresponds to a particular feature. The distinct classes of the trace are extracted while the trace is being generated. These are the entities that will be clustered by our approach.

Clustering Omnipresent Classes: Software systems often contain components that act as mere utilities. They are referred to as omnipresent objects. Müller et al. showed that omnipresent components obscure the structure of a system and argued that they should be excluded from the architecture recovery process [20]. Wen and Tzerpos [25] also agreed that removing omnipresent components can significantly improve the clustering result. Therefore, we have decided to remove omnipresent classes of the system and group them into a utility cluster.

The detection of omnipresent classes has been discussed in many studies. Hamou-Lhadj and Lethbridge, for example, presented an approach for automatic detection of system-level utilities using fan-in analysis [8]. Wen and Tzerpos considered omnipresent components to be the ones that are connected to a large number of subsystems [25]. In [16], Mancoridis et al. presented a heuristic based approach for detecting these components.

Building the Feature-Class Matrix: A feature-class matrix is a two dimensional table that provides the input to the clustering algorithm. The rows represent the classes (i.e. the entities to cluster) and the columns are the feature traces. The value of each cell of the table is either 0 or 1, indicating the absence or presence of a class in the feature trace.

Applying the Clustering Algorithm: The next step consists of choosing a clustering algorithm and applying it to the feature-class matrix. Jain et al. described a taxonomy of clustering approaches, which broadly categorizes clustering algorithms into two main categories [11]: partitioning and hierarchical clustering algorithms. Partitioning algorithms start with an initial partition of a system, consisting of a given number of clusters, which is known in advance. Iteratively, the partitions or clusters are modified to optimize some predefined criteria (e.g. high cohesion), keeping the number of partitions or clusters constant.

Hierarchical algorithms, on the other hand, use a bottom-up approach, starting from the individual entities, gathering them two by two into small clusters, which are in turn gathered into larger clusters. The process continues until one large cluster containing all elements is created. Since software systems are commonly presented as a nested decomposition or hierarchy, the clustering decomposition of modules into sub modules is useful, especially for understanding large system [13]. For this reason we have adopted hierarchical clustering algorithms in our research. A more specialized subset of hierarchical algorithms is referred to as agglomerative hierarchical algorithms [13]. There are several types of agglomerative hierarchical algorithms that are differentiated based on how they compute the distance from a new cluster to all other ones. In this paper, we adopt a complete linkage algorithm using Jaccard coefficient as a similarity metric [2]. This is because complete linkage as well as Jaccarad distance metrics have be shown to perform better compared to the other schemes [1].

The result of the clustering algorithm can be visualized through a dendrogram (Figure 2), which is a tree-like structure representing clusters formed at different stages of the algorithm. A cut through the tree determines a set of clusters of the system.

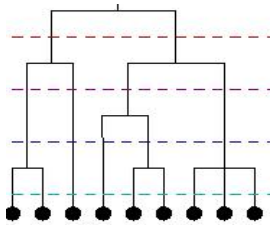

Figure 2. A dendrogram with various cut points represented as dashed lines

Removing Singleton Clusters: The final step of constructing the skeleton decomposition is to analyze the clusters resulting from the previous step and remove the ones that contain one single class (i.e., singleton clusters). The classes of these singleton clusters are added to the pool of components that will be clustered later when we apply the orphan adoption algorithm (i.e., during the second iteration of the clustering process). The rationale behind this is that singleton clusters are clusters without any similarity to other existing clusters. These clusters can often lead to a deterioration of the skeleton stability.

2.2. Full Decomposition

In this step, we cluster the remaining classes, i.e., the ones that have not previously assigned to any skeleton

cluster. For this purpose, we use the orphan adoption algorithm introduced by Tzerpos et al. [24].

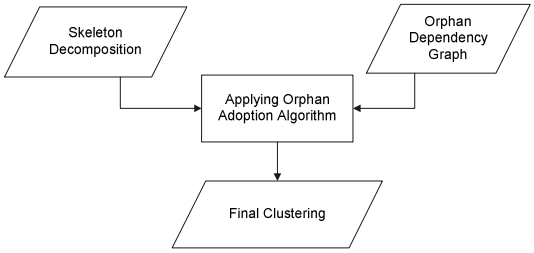

Figure 3. Phase 2: Full decomposition

The goal of the algorithm is to add orphan classes to the already identified clusters based on analyzing the static dependencies between the orphan classes and the ones that belong to the skeleton clusters (see Figure 3).

The orphan adoption algorithm works as follows. First, the algorithm attempts to identify the core cluster for each orphan based on naming criteria. In other words, the algorithm matches the name of the orphan to the name of the skeleton clusters based on naming conventions used during the development of the software system. However, this assumes that the developers have followed a specific naming convention during the development process. This assumption is not always valid in practice. In our research, we did not consider any naming conventions to avoid situations where naming conventions have either not been applied or followed correctly.

In situations when the clustering based on naming conventions fails, the algorithm uses structural relations to uncover the core cluster for each orphan. The algorithm calculates the strength of relation of an orphan with each core cluster by considering the number of relations that exist between the orphan and the entities of a cluster. It then places the orphan in the core cluster with which it has the strongest relation. If there is a tie between many core clusters then the core cluster having more entities in it wins over the other clusters. If there is no core cluster selected for an orphan, then the algorithm creates a new cluster called "orphan container" and add all such orphans to it. Orphan container represents all orphans that do not have relations to core clusters.

It should be noted that the orphan adoption algorithm will have only a limited effect on the skeleton decomposition. Changes to the skeleton structure will be mainly limited to the addition of new clusters or populating the existing ones. Components assigned to the clusters during the dynamic analysis will typically not be relocated during the static dependency analysis to any other clusters. Furthermore, new clusters will be created if and only if a component has no static dependency with any existing component in the skeleton. By limiting the modifications to skeleton additions, we are able to preserve the skeleton derived by our dynamic analysis.

3. Case Study

In this section, we present a case study to demonstrate the applicability of our approach. We first briefly introduce the target system used for the case study. Then, we describe in detail the application of our clustering approach. Finally, we compare the resulting decomposition to the one provided by the designers of the target system.

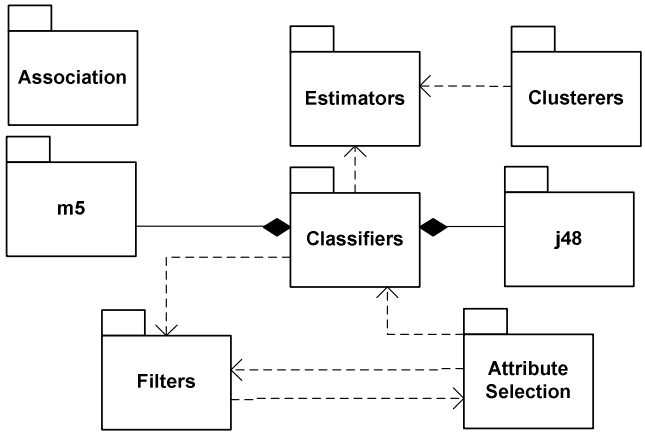

Figure 4. Weka architecture

3.1. Target System

We have applied our approach on an object-oriented system called Weka[1] (ver. 3.0), which is a medium-sized Java based open source application. Weka is a machine learning tool that provides several algorithms for classification, regression techniques, clustering, and association rules. It contains 10 packages, 147 class files with a total of 95 KLOC of source code.

We selected Weka because it is well documented, including its architectural properties, which we used to validate our approach and to identify the omnipresent classes. Additional references with respect to the Weka architecture can be found in [27].

Figure 4 shows the architecture of Weka. The tool contains the following packages: `associations`, `clusterers`, `core`, `estimators`, `filters`, `attribute selection`, and `classifiers`. The `classifiers` package contains two additional package namely `j48` and `m5`. The packages `core` and `filters` are not shown in Figure 4 because they are utility

[1] http://www.cs.waikato.ac.nz/ml/weka/

packages as we will discuss in the next subsection. Note that the `associations` package is isolated in Figure 4 because it only depends on `core` and `filters`.

Table 1. Weka features used in this study

Trace	Feature
T1	Cobweb clustering algorithm
T2	EM clustering algorithm
T3	IBk classification algorithm
T4	OneR classification algorithm
T5	Decision table classification algorithm
T6	J48 (C4.5) classification algorithm
T7	SMO classification algorithm
T8	Naïve Bayes classification algorithm
T9	ZeroR classification algorithm
T10	Decision stump classification algorithm
T11	Linear regression classification algorithm
T12	M5Prime classification algorithm
T13	Apriori association algorithm

3.2. Constructing the Skeleton Decomposition

In this section, we apply the steps of constructing the skeleton decomposition to Weka.

Feature Selection and Trace Generation:

Table 1 shows the software features selected for this study. We carefully chose features covering most Weka machine learning algorithms so as to ensure a good coverage of the source code and a balanced set of features.

We generated an execution trace for each of the features of Table 1. For this purpose, we instrumented the Weka class files using the BIT framework [15] by inserting probes at entry and exit of every method including constructors. Since we are only interested in analyzing the presence or absence of a particular class in a trace, only distinct classes of each trace have been saved.

Removal of Omnipresent Classes:

Most of the removal of omnipresent classes can be performed automatically based on the omnipresent detection approach introduced in [8]. However, the analysis of the Weka documentation revealed that the package `core` is used by all other packages of the system, resulting in a high fan-in for this package. We further observed that the `filters` package is also providing utility functions for different machine learning algorithms, like the filtering and processing of data. Therefore, we decided to categorize these packages as utility packages, cluster them into a utility cluster, and eliminate all classes within these packages from further analysis.

Building the Feature-Class Matrix:

Building the feature class matrix is straightforward, since we have already analyzed the feature traces and identified and collected the distinct set of classes involved in different features.

Table 2. Extracted clusters from the first phase

Cluster	Classes
C1	weka.classifiers.Evaluation weka.classifiers.Classifier weka.classifiers.DistributionClassifier weka.classifiers.DecisionStump
C2	weka.clusterers.ClusterEvaluation weka.clusterers.Clusterer weka.clusterers.Cobweb
C3	weka.estimators.KernelEstimator weka.classifiers.LinearRegression
C4	weka.classifiers.m5.Option weka.classifiers.m5.M5Prime weka.classifiers.m5.M5Utils weka.classifiers.m5.Node weka.classifiers.m5.Function weka.classifiers.m5.SplitInfo weka.classifiers.m5.Impurity weka.classifiers.m5.Values weka.classifiers.m5.Errors weka.classifiers.m5.Ivector weka.classifiers.m5.Dvector weka.classifiers.m5.Matrix
C5	Weka.associations.ItemSet weka.associations.Apriori
C6	Weka.clusterers.DistributionCluster weka.clusterers.EM weka.estimators.DiscreteEstimator
C7	Weka.classifiers.IBK
C8	Weka.classfiers.OneR
C9	Weka.classifiers.SMO
C10	Weka.estimators.NormalEstimator weka.classifiers.NaiveBayes
C11	Weka.classifiers.ZeroR
C12	Weka.classifiers.j48.C45ModelSelectio weka.classifiers.j48.J48 weka.classifiers.j48.ModelSelection weka.classifiers.j48.C45PruneableClassifierTree weka.classifiers.j48.ClassifierTree weka.classifiers.j48.Distribution weka.classifiers.j48.NoSplit weka.classifiers.j48.ClassifierSplitModel weka.classifiers.j48.C45Split weka.classifiers.j48.EntropyBasedSplitCrit weka.classifiers.j48.InfoGainSplitCriteria

Applying the Clustering Algorithm:

The next step is to apply the complete linkage hierarchical algorithm using Jaccard as a similarity measure.

The result of the hierarchical clustering is shown in the dendrogram of Figure 5.

Cutting the dendrogram at a certain cut point will result in distinct trees that have been formed underneath the cut point. These distinct trees correspond to different clusters of classes identified by our clustering algorithm. One of the challenges in our approach is to determine this cut point. As discussed in [1], there exist more than one potential cut point value. The common practice is to cut a dendrogram at an arbitrary height and then re-adjust it if the result provides either too many clusters with too few classes or only a few clusters containing a large number of classes.

Table 3. The final decomposition of Weka

Cluster	Classes
C1	weka.classifiers.DecisionStump weka.classifiers.Evaluation weka.classifiers.AdaBoostM1 weka.classifiers.Bagging weka.attributeSelection.WrapperSubsetEval weka.attributeSelection.OneRAttributeEval weka.classifiers.CheckClassifier ….
C2	weka.clusterers.ClusterEvaluation weka.clusterers.Clusterer weka.clusterers.Cobweb
C3	weka.classifiers.LinearRegression weka.estimators.KernelEstimator weka.estimators.DKConditionalEstimator weka.estimators.KDConditionalEstimator weka.estimators.KKConditionalEstimator weka.estimators.Estimator weka.estimators.ConditionalEstimator weka.estimators.MahalanobisEstimator weka.estimators.NNConditionalEstimator weka.estimators.PoissonEstimator
C4	weka.classifiers.m5.M5Prime weka.classifiers.m5.Options weka.classifiers.m5.M5Utils weka.classifiers.m5.Node …
C5	weka.associations.ItemSet weka.associations.Apriori
C6	weka.clusterers.EM weka.clusterers.DistributionClusterer weka.estimators.DiscreteEstimator weka.estimators.DDConditionalEstimator weka.estimators.DNConditionalEstimator
C10	weka.classifiers.NaiveBayes weka.estimators.NormalEstimator weka.estimators.NDConditionalEstimator
C12	weka.classifiers.j48.J48 weka.classifiers.j48.5ModelSelection weka.classifiers.j48.ModelSelection weka.classifiers.j48.5PruneableClassifierTree weka.classifiers.j48.ClassifierTree weka.classifiers.j48.Distribution weka.classifiers.j48.NoSplit …

After manually analyzing the content of the dendrogram, we decided to cut the tree at point `P1` as shown in Figure 5 so as to balance the number of clusters and the number of components in each cluster. The resulting clusters are shown in Table 2.

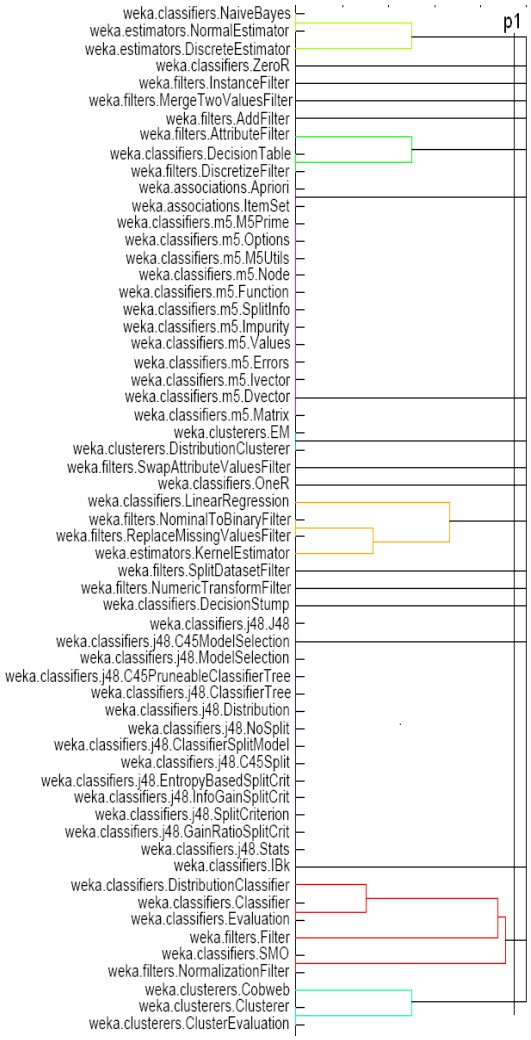

Figure 5. Dendogram generated by applying the clustering algorithm to Weka classes

Removal of Singleton Clusters:

From the list of clusters of Table 2, we filtered out four clusters that are considered as singleton clusters (`C7, C8, C9` and `C11`). The contained classes were moved to the pool of orphans to be further processed later as part of the static analysis. The remaining core skeleton of the system consisted of eight clusters, `C1-C6, C10,` and `C12`.

3.3. Applying the Orphan Adoption Algorithm

In this second step of the clustering process, we clustered all classes not previous assigned by adding them to an

existing or a new cluster using the orphan adoption algorithm.

For this purpose, we used a tool called Javex[2] to extract dependencies among orphans and the already clustered classes. Javex is a fact extractor for Java. The tool takes into account many dependencies that may exist between two classes such as method calls, inheritance relationships, etc. The facts were saved in RSF (Rigi Standard Format) [19] and fed to the orphan adoption algorithm.

Table 3 shows the result of adding orphans to the skeleton decomposition. Due to limited space, not all classes are shown in the table. The orphan adoption algorithm did not change the structure of the skeleton decomposition. It populated the already formed clusters by adding the non core classes. Our clustering approach identified a final of eight clusters: `C1-C6, C10,` and `C12.` The evaluation of the resulting decomposition is presented in the next subsection.

3.4. Evaluation of the results

To validate the result of our clustering approach, we compared the recovered decomposition for the Weka system to the decomposition provided by the software designers of this system, which is shown in Figure 4.

For the evaluation, we measured the extent to which two given decompositions are similar to each other using the MoJoFM distance [25], which is an improved version of a MoJo metric introduced by the same authors [23]. The metric takes two partitions as input and calculates the number of "move" and "join" operations, needed to transform one partition into another. The "move" operation moves an entity from one cluster to another existing cluster or a newly created cluster. The "join" operation joins two clusters into one cluster and reduces the number of clusters by one. The *MoJoFM* distance assigns the same weight to both operations.

More precisely, given two partitions P and Q, *MoJoFM* is calculated as follows [25]:

$$MoJoFM = \left(1 - \frac{mno(P,Q)}{\max(mno(\forall P,Q))}\right) \times 100\% \cdot$$

Where:

- $mno(P,Q)$ is the number of "move" and "join" operations needed to go from P to Q.
- $\max(mno(\forall P,Q))$ is the maximum number of possible "move" and "join" operations to transform any partition P into Q.

[2] http://www.swag.uwaterloo.ca/javex/index.html

MoJoFM ranges from 0% to 100%. It converges to 0% if the partitions are very different from each other. It reaches 100% if the two partitions are exactly the same.

Table 4 shows the result of comparing Weka extracted decompositions with the expert decomposition using MoJoFM.

Table 4. Comparing the extracted decomposition to the expert decomposition using the MoJoFM metric

Target System	MoJoFM
Weka	87.83%

As shown in this table, the recovered decomposition of Weka is very similar to the one provided by Weka designers (MoJoFM = 87.83%). In order to further investigate the reason for the variation in the performance of our algorithm, we superimposed the resulting clusters onto the Weka decomposition provided by the designers. The result is shown in Figure 6. For example, the cluster C1 (Table 3), which was recovered by our approach contains classes that belong to the Weka packages: `classifiers` and `attribute selection`. In the following subsection, we discuss the observed discrepancies between the decompositions extracted using our approach and the ones created by the domain expert.

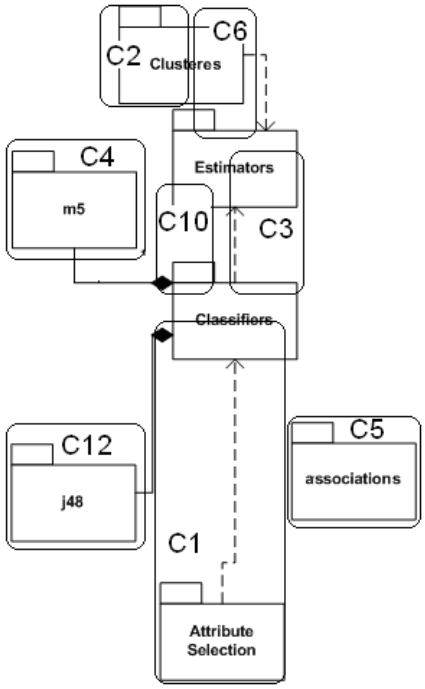

Figure 6. Resulting clusters compared to Weka architecture

From Figure 6, we can observe that the `m5`, `j48` and `associations` packages of Weka were completely

recovered by our clustering technique. The corresponding clusters are `C4`, `C12`, and `C5` respectively.

The cluster `C2` contains some classes of the Weka `clusterers` package. To understand the reason behind this, we had to examine the class dependency graph of Weka using a static analysis tool called Structural Analysis for Java (SA4J)[3]. The tool has a feature called "explorer" that allowed us to visually explore the relationships between the system components. Further analysis of the content of `C2` using SA4J revealed that `C2` components are tightly coupled and have no other dependencies with any other classes other than the classes of the utility packages `core` and `filters`. Similar results were obtained by analyzing the cluster `C6`, which contains the remaining classes of the `clusterers` package.

The analysis of the cluster `C1` showed that the static dependency analysis could not sufficiently cluster the packages `attribute selection` and `classifiers`. `C1` includes all classes of `attribute selection` which is quite satisfactory because they are strongly interrelated classes, but also some classes of the `classifiers` package that strongly depend on the `attribute selection` package.

Clusters `C3` and `C10` share the content of the package `estimators`. It appears that this package implements general purpose functions used by the classification and the clustering algorithms of Weka. Further analysis of the `C3` and `C10` package using SA4J revealed that the classes of these packages do not have any dependencies with each other despite the fact that Weka designers chose to put them in the same package.

From the above analysis, we were able to infer that Weka designers have packaged the system classes based on the fact that they implement similar functionality (e.g., classification algorithms), although these classes might be completely decoupled. Our clustering approach, on the other hand, groups related system components according to the degree of their interaction.

4. Related Work

There exists a significant body of work in software clustering [2, 14, 18, 22, 28], often with a particular emphasis on static dependency analysis, measuring the similarity between source code objects. Less work has been performed using dynamic analysis techniques. Due to space limitations we focus our review on these approaches that measure similarity among artifacts other than static source code.

[3] SA4J: http://www.alphaworks.ibm.com/tech/sa4j

In [28], Tzerpos et al. introduced a clustering approach based on a static component dependency graph. They have employed dynamic analysis through weighing the static dependencies between components by number of times a component calls another component. However, this approach does not use dynamic analysis to establish dependencies between components and still uses static analysis as the main mechanism for clustering.

Eisenbarth et al. [6, 7] proposed a solution for the feature location problem using a combination of static and dynamic analysis. They used concept analysis to build a concept lattice that relates components to features. We believe that the resulting concept lattice can further be processed to group components that implement common features into clusters. The authors, however, did not perform the clustering. In addition, our approach tends to be simpler in practice since it does not involve building and processing a concept lattice.

In [24], Tzerpos et al. introduced the concept of incremental clustering. They presented an algorithm called ACDC (Algorithm for Comprehension Driven Clustering). An interesting aspect of their work is that they did not use the source code to build the skeleton composition. Instead, they built an algorithm that simulates the patterns that software engineers use to group elements into subsystems. The authors showed that their approach performs better that source code based clustering. We attribute this to the fact that they used an abstract concept (patterns) to build the skeleton; we used software features.

Anquetil and Lethbridge [2] used file names as a clustering criterion. The authors run several showed that files names in the best criterion for the large system that experimented with. Once again, we attribute the success of this approach to the fact that they used file names, another abstract concept, for clustering.

Dugerdil [5] et al. presented a reverse-architecting process based on the rational unified process and on dynamic analysis of program executions. They used a sampling technique on uncompressed traces. Components that are invoked in each sample are grouped together into a cluster. This approach is, however, very sensitive to the number of samples in the trace and the criteria used for sampling.

5. Conclusion and Future Work

In this paper, we introduced an incremental clustering approach, which is based on using software features as clustering criteria to create the skeleton decomposition, and static dependencies to complete the full decomposition of the system.

We argued that software features are a good clustering criterion because of the fact that (1) they represent an

intrinsic grouping of the components that implement them, (b) they are abstract concepts and, as such, they can readily be used to extract abstract components of the system.

We applied our clustering technique to a system called Weka. The results are promising. We succeed to construct a decomposition where the components that interact with each other are grouped into clusters.

The main feature direction is to continue experimenting with the proposed approach and assess its effectiveness when applied to large software systems with poor architecture.

In addition, there is a need to determine the number of features needed for better results and how this number correlates with code coverage by experimenting with many various systems. We also need to compare our technique with existing techniques.

6. References

[1]. N. Anquetil, C. Fourrier, T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization Method", *In Proc. of the 6th WCRE*, 1999, pp. 235-255.

[2]. N. Anquetil and T. Lethbridge, "Recovering software architecture from the names of source files", *Journal of Software Maintenance: Research and Practice,* 1999, pp. 201-221.

[3]. C. Patel, A. Hamou-Lhadj, J. Rilling, "Software clustering based on behavioural features", *In Proc. of the 11th IASTED International Conference on Software Engineering*, 2007, pp. 591-185.

[4]. J. Davey and E. Burd, "Evaluating the Suitability of Data Clustering for Software Remodularization," *In Proc. Seventh Working Conf. Reverse Engineering*, 2000, pp. 268-277.

[5]. P. Dugerdil, "Using trace sampling techniques to identify dynamic clusters of classes", *CASCON 2007*, 2007, pp. 306-314.

[6]. T. Eisenbarth, R. Koschke, D. Simon, "Aiding Program Comprehension by Static and Dynamic Feature Analysis", *In Proc. of the IEEE International Conference on Software Maintenance*, 2001, pp. 602-612.

[7]. T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, 2003, pp. 210 - 224.

[8]. A. Hamou-Lhadj, T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 181-190.

[9]. D. H. Hutchens and V.R. Basili, "System Structure Analysis: Clustering with Data Bindings," *IEEE Trans. Software Engineering,* vol. 11, no. 8, Aug. 1985, pp. 749-757.

[10]. J. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1994.

[11]. A. K. Jain, M.N. Murty, and P.J. Flynn, "Data Clustering:AReview," *ACM Computing Surveys*, vol. 13, no. 3, Sept. 1999, pp. 264-323.

[12]. R. Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution," PhD dissertation, Univ. of Stuttgart, 2000.

[13]. R. Koschke and D. Simon, "Hierarchical Reflection Models," *In Proc. of the 10th Working Conf. Reverse Engineering*, 2003, pp. 36-45.

[14]. J. Kothari,T. Denton,S. Mancoridis, and A. Shokoufandeh, "On Computing the Canonical Features of Software Systems", *In Proc. 13th IEEE Working Conference on Reverse Engineering (WCRE'06)*. 2006, pp. 93 - 102.

[15]. H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes", *In Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, 1997, pp. 73-82.

[16]. S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *In Proc. Int'l Conf. Software Maintenance*, 1999, pp. 50-62.

[17]. E. Merlo, I. McAdam, and R. D. Mori, "Source Code Informal Information Analysis Using Connectionist Models", *In Proc. of the Int. Joint Conference on AI*, 1993, pp. 1339-1344.

[18]. B.S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, Mar. 2006, pp. 193-208.

[19]. H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse engineering approach to subsystem structure identification", *Journal of Software Maintenance: Research and Practice*, 1993, pp. 181–204.

[20]. H.A. Müller and J.S. Uhl, "Composing Subsystem Structures using (k, 2)-Partite Graphs", *In Proc. of the International Conference on Software Maintenance*, 1990, pp. 12-19.

[21]. C. Riva, "Reverse Architecting: An Industrial Experience Report", *In Proc. of the 7th Working Conf. Reverse Eng*. 2000, pp. 42-51.

[22]. M. Saeed, O. Maqbool, H.A. Babri, S.M. Sarwar, S.Z. Hassan "Software Clustering Techniques and the Use of the Combined Algorithm", *In Proc. of the In International Conference on Software Maintenance and Re-engineering*, 2003.

[23]. M. Shtern and V. Tzerpos, "A Framework for the Comparison of Nested Software Decompositions", *In Proc. of the 11th IEEE Working Conf. Reverse Engineering*, 2004, pp. 284-292.

[24]. V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering", *In Proc. of the 7th Working Conference on Reverse Engineering*, 2000, pp. 258–267.

[25]. Z. Wen and V. Tzerpos, "Software Clustering based on Omnipresent Object Detection", *In Proc. of the 13th IEEE Int. Workshop on Program Comprehension*, 2005, pp. 269-278.

[26]. T. A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization", *In Proc. of the 6th Working Conference on Reverse Engineering*, 1997, pp. 33-43.

[27]. H. Witten and E. Frank, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2005.

[28]. C. Xiao and V. Tzerpos, "Software Clustering based on Dynamic Dependencies", *In Proc. of the 9th European Conference on Software Maintenance and Reengineering*, Manchester, 2005, pp. 124-133.

[29]. S. S. Yau and J. S. Collofello, "Some stability measures for software maintenance", *IEEE Transactions on Software Engineering*, 1980, pp. 545-552.

[30]. N. Wilde, and M. Scully, "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, vol. 7, 1995, pp. 49-62.