Software Clustering Using Dynamic Analysis and Static Dependencies

Chiragkumar Patel

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2008

Your file Votre référence
ISBN: 978-0-494-45525-8
Our file Notre référence
ISBN: 978-0-494-45525-8

ABSTRACT

Software Clustering Using Dynamic Analysis and Static Dependencies

Chiragkumar Patel

Maintaining a large software system is not an easy task. The problem is that software engineers must understand various parts of the system prior to performing the maintenance task at hand.

The comprehension process of an existing system can be made easier if the system is decomposed into smaller and more manageable clusters; software engineers can focus on analyzing only the subsystems needed to solve the maintenance task at hand. There exists several software clustering techniques, among which the most predominant ones are based on the analysis of the source code. However, due to the increasing complexity of software, we argue that this structural clustering is no longer sufficient.

In this thesis, we present a novel software clustering approach that combines dynamic and static analysis. Dynamic analysis is used to build a stable core skeleton decomposition of the system by measuring the similarity between the system's components according to the number of software features they implement. Static analysis is used to enrich the skeleton decomposition by adding the components that were not clustered using dynamic analysis.

A case study involving two object-oriented systems is presented to evaluate the applicability and effectiveness of our approach.

Acknowledgements

It gives me an immense pleasure to take the opportunity to thank every one who supported me in the completion of my Masters.

I would like to express my deep regards to my supervisor Dr. Abdelwahab Hamou-Lhadj. I would like to thank him for giving me an opportunity to conduct this research. I am grateful to him for sharing his valuable experience and knowledge to me that created new ideas in me for my work. His encouragement developed enthusiasm in me to explore my research work.

I would like to thank Dr. Juergen Rilling, my co-supervisor, for assisting me in conducting my research and also for his efforts to review my work, which has provided a better insight to it.

I would like to thank to my lab mates, Computer Science and Electrical department staff and friends for always being supportive in providing the valuable information, guidance and help whenever I needed. It was a great pleasure to share ideas about academic as well as cultural interests.

Finally, I would like to thank to my family for their continuous support in my education. I would like to express my heartiest regards to my mother, who has always supported me in all of my decisions.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Software maintenance is perhaps one of the most complex software engineering activities; software engineers must understand various parts of a system before they can perform the maintenance task at hand. In this thesis, we argue that the increased complexity and behavioural aspects of today's software systems demand for better clustering algorithms to support the comprehension and analysis of existing systems. The presented approach goes beyond the mere static analysis of the source code used predominantly by most existing clustering techniques. For this purpose, we introduce a novel clustering algorithm that uses dynamic analysis, as its main mechanism, to recover the core structure decomposition of a system, and static information to enrich this decomposition.

In the remainder of this chapter we will motivate the thesis, summarize the contributions of this work, and present the thesis outline.

1.1 Problem and Motivations

Software evolution is an essential part of the software life cycle. As part of traditional software life cycle models, software documentation plays an important role in supporting the development and maintenance of a software system. However, it has also been shown that for many existing systems, the documentation associated with an existing system is often incomplete, inconsistent, or even inexistent [Wiggerts 97]. This is further complicated by the fact that key developers, knowledgeable of the system's design, commonly move to new projects or companies, taking with them valuable technical and domain knowledge about the system [Tzerpos 98].

These factors contribute to make these software systems difficult to comprehend and to evolve. As a result, software engineers need to spend a considerable amount of time understanding what the application does, and how it does it.

Software clustering techniques were introduced to facilitate the comprehension process by automatically decomposing the system into smaller, more manageable clusters, enabling software engineers to focus on analyzing only the subsystems needed to solve the maintenance task at hand [Wiggerts 97, Tzerpos 98, Anquetil 99a].

Existing software clustering techniques can be grouped into two main categories [Anquetil 99a]. The first category, also the most commonly used, relies on the source code to extract relations among a system's components. Examples of such relations include file inclusions, routine calls, type references, etc. However, many of these clustering approaches are limited to static analysis of the source code and therefore are very conservative in analyzing the dynamic interactions among the system's entities. The second category of clustering approaches is based on less formal artefacts, such as file names [Anquetil 99b], comments [Merlo 93], etc. The main drawback with these techniques is that they assume that the software development process adopts certain naming, programming and/or design conventions. However, it is not feasible to expect that such conventions will be followed in practice [Patel 07]. In addition, extracting and analyzing the knowledge from such informal information sources is often difficult due to the presence of noise and ambiguity in the data.

In this thesis, we propose a novel clustering approach that combines both dynamic (trace based) and static dependency analysis. The approach introduces a software clustering decomposition based on behavioural relations among system components compared to existing approaches that typical relying only on mere structural relations.. The approach extracts behavioural relations that exist in the source code by executing a single software feature. The proposed approach has two main phases: In the first phase, we measure the similarity between system entities based on the software features they implement. We achieve this by examining traces generated by exercising these system features. This dynamic analysis provides us with a skeleton decomposition of the system structure. In the next step, we apply static dependency analysis to further cluster the system by adding the remaining components to the skeleton decomposition.

1.2 Research Contributions

The main research contributions of this thesis are as follows:

- A novel software clustering approach based on dynamic analysis is presented. The approach uses a dynamic analysis of feature traces as a primary step to create skeleton decomposition of feature related components rather than relying on a mere static analysis of relations among a system's components as used by most existing software clustering techniques.

- In this thesis, we introduce the notion of a software feature as a new clustering criterion. To our knowledge, this is the first study that exploits software features in the context of software clustering. We argue that since software features represent high-level system functions that can guide functional partitioning from the forward engineering perspective, they are excellent candidates to be used to recover these corresponding partitions from low-level implementation details.

- We present a complete hybrid clustering approach that combines both dynamic and static dependency analysis. In this approach, static analysis is used to further refine the cluster generated by the dynamic analysis.

- The clustering approach was applied on two object-oriented software systems, to evaluate our approach and discuss its applicability.

1.3 Outline of the Thesis

The remainder of this thesis is organized as follows.

In Chapter 2, we introduce background related to essential concepts used in this thesis. The chapter covers cluster analysis, software clustering and provides a detailed survey and discussion of the most cited software clustering techniques relevant to this thesis context. The chapter ends with a discussion section.

In Chapter 3, we present the concept of software features and the motivation for using it as a clustering criterion. We start the chapter by defining what constitutes a software feature in other software engineering related areas namely, feature-driven software development, feature location, and feature interaction. Next, we discuss the motivation behind using software features as a clustering criterion, along with the potential issues that emerge.

In Chapter 4, we present in detail the software clustering approach presented in this thesis. We start, by describing the overall approach, which consists of two phases as previously mentioned. The chapter continues with the detailed description of each phase, where phase 1 is based on dynamic analysis and phase 2 is based on static analysis.

In Chapter 5, we evaluate the applicability of the proposed approach by applying it on two software systems. The target systems are introduced as well as the results of applying the clustering techniques to both systems are reported.

Finally, Chapter 6 concludes the thesis by summarizing the main contributions of the thesis, and outlining some of the potential future work to enhance the presented approach, followed by closing remarks.

Chapter 2 Background

In this chapter, we present the background and terminology relevant to the thesis context. In Section 2.1, we introduce various cluster analysis concepts. In Section 2.2, we focus on software clustering and present a detailed survey of existing clustering techniques, followed by a discussion in Section 2.3.

2.1 Cluster Analysis

The formal study of clustering algorithms and methods is known as cluster analysis [Jain 88], which aims to classify entities within a domain into disjoint groups according to some kind of similarity metric [Anderberg 73]. The resulting groups of entities are often referred to as "clusters" or "partitions". Clustering techniques have been used in many fields including life sciences (e.g., biology, zoology), medical sciences (e.g., psychiatry), behavioural and social sciences (e.g., sociology, education), earth science (e.g. geology), engineering sciences (e.g., pattern recognition, software reverse engineering), information and decision sciences (e.g., information retrieval, marketing research), etc.

The clustering process itself is based on four main elements [Jain 88]:

- Entities to cluster
- Attributes that describe the entities
- Similarity metric used to measure the distance between the entities
- Clustering algorithm that describes the steps of the clustering process

2.1.1 The Entities

The entities to cluster depend on the domain in which a cluster analysis is performed. In psychology, for example, the entities could be training methods, behavioural patterns, etc. There is no common terminology that describes clustering concepts. The objects to be clustered are normally referred to as: "data units", "cases", "entities", "patterns", "observations", and "elements" [Anderberg 73, Jain 88]. In the context of this thesis, we refer to the entities to be clustered as "entity".

2.1.2 The Attributes

Attributes define the characteristics of the entities to be clustered. They are also called "features", "variables", "characters", or "measurements" [Anderberg 73, Jain 88]. The relationship between entities and the attributes that describe them can be represented using an entity-attribute matrix, where rows represent entities and columns refer to attributes of these entities. Therefore, a cell (i, j) represents the value of an attribute A_j for an entity E_i. The values of all attributes for a specific entity form the attribute vector for this entity.

Clustering attributes can be grouped into four categories depending on the values that can be assigned to them [Anderberg 73]. The first category is the nominal scale attributes, which limit the comparison between two variable values to whether they are equal or not. A special type of nominal scale attributes are binary attributes, which can have only two possible values. The second category consists of ordinal scale attributes, which improve over the nominal scale attributes by allowing "greater than" or "less than" comparisons among entities. Interval scale attributes, which represents the third category, provide the ability to express additional information compared to ordinal scale attributes. Interval scale attributes allow to express the difference between the values (e.g., temperature measurement). Finally, the last category consists of ratio scale attributes, which allow comparing attribute values based on their ratios (e.g., 20 meters is double the distance of 10 meters).

2.1.3 Similarity Metrics

An important step used for any clustering technique is the need to select a similarity metric, which will determine how the similarity of two entities is computed. There exist various categories of similarity measures in the literature [Wiggerts 97, Anderberg 73], among which the most popular are perhaps the association coefficient metrics.

Table 2.1. Association coefficient matrix

	Entity E_i	
Entity E_j	a	b
	c	d

Where:

a = The number of attributes present for both E_i and E_j

b = The number of attributes present for E_i and **not** in E_j

c = The number of attributes present for E_j and **not** in E_i

d = The number of attributes **not** present for both E_i and E_j

Association coefficients are based on the analysis of binary attributes. The association coefficients measure the similarity between two entities based on the number of attributes present for each entity. Using the association coefficient metrics, the similarity between two entities is calculated based on a 2x2 matrix like the one shown in Table 2.1.

Existing association coefficient metrics vary depending on how a, b, c, and d are weighted. Table 2.2 shows examples of commonly used association coefficients and how they compute similarity based on a, b, c, and d [Anderberg 73].

Table 2.2. Association coefficient metrics

Name	**Value**
Simple matching coefficient	(a + d) / (a + b + c + d)
Russell and Rao	a / (a + b + c + d)
Jaccard	a / (a + b + c)
Sorenson-Dice	2a / (2a + b + c)
Kulcsynski	a / (b + c)
Rogers-Tanimato	(a + d) / (a + 2(b + c) + d)
Sokal and Sneath	a / (a + 2(b + c))

It should be noted that many coefficients (e.g., Jaccard, Sorenson-Dice, Sokal and Sneath) do not consider the value of d, the number of attributes not present in both entities. The argument is that there seems to be a consensus among researchers that absent attributes often lead to a large and meaningless clusters absorbing most entities [Anquetil 03].

There are other types of similarity metrics such as the distance measures, which measure the geometrical distance between the attribute vectors of two entities. Distance measures are usually used when the attributes are of type ordinal, interval, or ratio scale attributes. Examples of distance measures include the Euclidian distance (a special case of the Minkowski distance), the Canberra distance, the Bray-Curtis distance, etc [Anderberg 73].

2.1.4 Hierarchical Clustering Algorithms

A clustering algorithm describes the steps involved in clustering entities of a system. There are two main clustering algorithms: Hierarchical and partitioning algorithms.

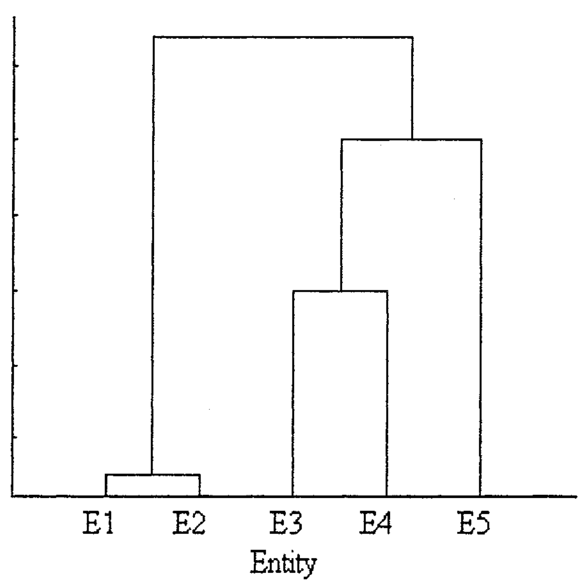

Figure 2.1. Example of a dendrogram

Hierarchical clustering algorithms work by clustering the entities of the system in an iterative manner. There are two types of hierarchical algorithms: agglomerative algorithms and divisive algorithms. Agglomerative algorithms use a bottom-up approach, starting from individual entities, with each of them represented in a single cluster. After each iteration, the algorithm merges two clusters, therefore reducing the initial number of clusters by one. Before the last iteration, all entities are clustered into two clusters. In the last final iteration, the algorithm merges these two clusters into one single cluster, which includes all the entities into it. The result of the clustering process is depicted in a tree-like structure known as a dendrogram (see Figure 2.1).

Four variations of agglomerative algorithms can be distinguished depending on the way they measure the similarity between the newly formed cluster and all pre-existing clusters. These variations, also referred to as updating rules, are: Complete linkage, Single linkage, Weighted average linkage and Unweighted average linkage.

Table 2.3. Updating rules for agglomerative hierarchical algorithms

Updating rule	**Similarity formula**
Single Linkage	$f(E_m, E_{ij}) = \min(f(E_m, E_i), f(E_m, E_j))$
Complete Linkage	$f(E_m, E_{ij}) = \max(f(E_m, E_i), f(E_m, E_j))$
Weighted Average Linkage	$f(E_m, E_{ij}) = (f(E_m, E_i), f(E_m, E_j))/ 2$
Unweighted Average Linkage	$f(E_m, E_{ij}) = (f(E_m, E_i)*\text{size}(E_i)+f(E_m, E_j)*\text{size}(E_j)) / (\text{size}(E_i)+ \text{size}(E_j))$

Table 2.3 summarizes these updating rules used in agglomerative clustering algorithms, with $f(E_m, E_{ij})$ referring to the similarity metric used to measure the similarity (proximity) between two clusters E_i and E_j. In this table, the proximity formula for different updating rule is presented, which finds a new proximity value between an

existing cluster E_m and a newly formed cluster E_{ij}. Cluster E_{ij} is created by merging the two clusters E_i and E_j. The proximity value between E_m and E_{ij} is calculated based on the proximity of E_m with E_i and E_j. For example, in single linkage updating rule, $f(E_m, E_{ij})$, which measures the proximity between an existing cluster E_m and a newly formed cluster E_{ij}, is calculated as the minimum value between $f(E_m, E_i)$ and $f(E_m, E_j)$.

It also has to be noted that the selection of an updating rule may have a significant impact on the resulting clusters. It has been shown that single linkage tends to form large and isolated clusters including many entities in them compared to complete linkage, which usually results in less isolated clusters [Anquetil 99a]. The results obtained by applying weighted and unweighted average linkage can be found between these two extremes, complete and single linkage.

Unlike agglomerative algorithms, divisive algorithms, which represent the second category of hierarchical algorithms, proceed in a top-down fashion [Wiggerts 97]. They start by grouping the entities to cluster into a single cluster. This cluster is then split, in the next iteration, into two clusters, which in turn are divided into more clusters, and so on, until a cluster is created for each entity.

Agglomerative algorithms have been shown to be timely more efficient compared to divisive algorithms. Divisive algorithms tend to take longer time to create two clusters in the very first iteration, because there are 2^{N-1}-1 (N is the number of entities to be clustered) possibilities [Wiggerts 97].

2.1.5 Partitioning Algorithms

Partitioning algorithms start by grouping all entities into a predefined number of clusters [Jain 88]. The entities are rearranged according to some clustering criteria provided by the user. The clustering criterion is expressed in the form of an objective function. Examples for such criteria include high cohesion, low coupling, shared data bindings, etc. It might take several iterations before the value of the objective function can be reached.

A partitioning algorithm begins by identifying the seed points from the entity set. These seed points act as the nucleus of a cluster that attracts other elements to them [Anderberg 73]. The number of seed points is equal to the number of clusters the algorithm produces. The set of seed points could be selected by the user or chosen randomly from the entity set. The idea is to group the entities according to their minimum distance with the seed points while at the same time improving the objective function. If seed points are selected from the entity set, then each partition will have at least one entity in it as its seed point.

Jain et al. discuss two issues related to using partitioning algorithms: the selection of the clustering criteria and the number of all possible iterations [Jain 88]. The selection of clustering criteria depends highly on the problem at hand and cannot be generalized to other problem domains. In addition, the criteria have to be expressed into a mathematical formula to form the objective function. The second challenge is somewhat more complex to resolve, as the number of iterations one has to perform in order to fulfill the criteria across all partitions can be enormous, even for small sets of entities. One proposed solution to this combinatorial explosion problem is to start with an initial partition of entities that a user can provide in advance. However, the selection of an initial partition requires great care since it has been shown that the result of applying the same partitioning algorithm to two different initial partitions can be significantly different.

2.2 Software Clustering

Software clustering is defined as recovering architectural knowledge by applying a clustering technique to software entities in order to group them into different clusters. The entities grouped in one cluster are more similar to one another and at the same time more dissimilar to the entities in other clusters [Tzerpos 98]. These clusters are also referred to as "sub-systems" or "decompositions" of a software system. There exists several software clustering techniques that vary depending on the attributes used to describe the entities to be clustered, the clustering process, as well as the clustering algorithm itself. These techniques can be grouped into two categories that we present here and discuss in more detail in the subsequent subsections:

- Graph-based clustering techniques
- Similarity matrix based clustering techniques

2.2.1 Graph-Based Clustering Techniques

Graph-based clustering techniques operate on a graph representation, where the nodes are the entities to be clustered and the links are the relationships among them.

Hutchens and Basili studied the applicability of clustering techniques to software architecture recovery [Hutchens 85]. In particular, they used data binding as a clustering criterion to cluster routines of FORTRAN programs based on the degree of data shared between these routines. For this purpose, a routine dependency graph was extracted from the system under study where the vertices represent the routines of the system and the edges represent the relationships among these routines based on the data they share. Examples of data binding between two routines p and q include data sharing through global variables. For example by having p assigning a value to a variable that is later referenced by q, etc.

The authors defined several levels of data binding, grouped into four categories, namely, potential data binding, used data binding, actual data binding and control flow data binding, with potential data binding being the simplest and least expensive to extract, whereas control flow data binding being the most complex and highly expensive to represent.

The authors showed the applicability of their approach by applying it to two small-sized FORTRAN systems. They used different hierarchical clustering algorithms on the routine dependency graphs extracted from these systems. They validated their approach by comparing the resulting decompositions to the ones provided by the designers of the systems. They concluded that the usage of data binding could be useful in software clustering after noticing a large degree of correspondence between their decompositions and the ones produced by the initial developers of both systems.

Mancoridis et al. proposed a tool, called Bunch, to group a system's modules (e.g., files, classes, routines, or any other component of the system) into clusters [Mancoridis 99]. The modules are represented in a module dependency graph (MDG) where the nodes represent a system's modules and the edges are the structural relations (e.g., procedural invocation, variable access) connecting the modules with each other. The clustering process was performed by partitioning the MDG into disjoint clusters using a partitioning clustering algorithm (see Section 2.1.5). The authors used high cohesion (dependency between the modules of the same partition) and low coupling (dependency between the modules of different partitions) as the main partitioning selection criterion, which they expressed in the form of an objective function referred to as the Modularization Quality (MQ) function. MQ ranges from -1 to 1, where the extremes represent no internal cohesion (-1) and no external coupling (1).

Another important contribution of the authors' approach is the ability to preprocess the entities of the system prior to performing the clustering. For this purpose, they allow the users of their tool to filter out utility modules (also called omnipresent components) from the clustering process. The rationale behind this is that omnipresent components tend to encumber the module dependency graph and may affect the effectiveness of the clustering process. In our research we apply similar approach of removing omnipresent components and propose to group them into a separate cluster that we call a utility cluster. This will be discussed in more detail in Section 4.2.3.

In addition, the tool enables users to guide the clustering process by allowing them to specify modules that must be clustered together. This ability for users to override the decisions made by the automatic partitioning algorithm adds a significant additional flexibility to the tool, allowing user knowledge to be integrated into the clustering process.

The authors conducted several experiments by applying their approach to clustering C programs. The main observation from these experiments was that a better result was obtained when omnipresent components were removed from the clustering process. In addition, they showed that a user-directed clustering approach combined with the automatic partitioning algorithm provided better results than relying on a pure automatic algorithm alone.

In [Tzerpos 00], Tzerpos and Holt presented an algorithm called ACDC (Algorithm for Comprehension-Driven Clustering) where the authors introduced the concept of incremental clustering. Their clustering process consists of two phases. During the first phase, they built a skeleton decomposition of the system, which contains core entities of the system. In the second phase, they cluster non-core entities by adding them to the already formed clusters. An interesting aspect of their work is that they did not use the source code to build the skeleton decomposition. Instead, they built an algorithm that simulates the way software engineers group entities into subsystems. The authors showed that their approach performed better than most other clustering techniques that cluster all entities at once using the source code to measure similarity between entities. We attribute this to the fact that they used an abstract concept (clustering patterns) to build the skeleton decomposition. In our case, we use software features as discussed in Section 3.

Tzerpos et al. [Xiao 05] are perhaps the first authors in the area of software clustering who used a combination of static and dynamic analysis for clustering. Their clustering technique uses a static component dependency graph to represent the entities to cluster (e.g., a system's files, classes, etc.) and the relationships among these entities (e.g., file inclusion, object interaction, etc.). Run-time information was used to weigh the edges of the static dependency graph based on the number of times a component (i.e., represented as a node in the graph) invokes another component according to a given scenario. This information was obtained by applying dynamic analysis techniques. More precisely, in their approach they instrumented the source code of the system, and an execution trace was generated by exercising the given scenario using the instrumented version of the system. The authors applied their technique to a large C-based software system and showed promising results.

Unlike the previous authors, Bauer and Trifu favoured higher level semantic information such as architectural patterns over static dependencies among a system's entities as the clustering criteria for their clustering approach [Bauer 04]. They argued that the result of clustering using static dependencies was not always significant to the end users. They clustered the classes of an object-oriented system using a five-phase approach. First, they extracted a class dependency graph from the system, where the nodes represent the classes of the system and the edges represent various relations among classes including access to global variables, method invocations, inheritance relations, etc. The second phase was performed manually and consisted of gathering architectural clues from the facts (i.e., classes and relations among them) extracted in the first phase. They defined an architectural clue as small structural pattern, which is part of an architectural pattern. In the third phase, they built a multi-edge system graph, which has classes as the nodes and the edges represent six types of coupling between classes: inheritance coupling, aggregation coupling, association coupling, access coupling, call coupling, and indirect coupling. The architectural clues were used to guide the identification of the type of coupling between any two given classes and create an edge between classes. There could be multiple edges possible between any two classes according to the type of coupling between them. During this phase, a weight is also assigned to the edges. The weight of an edge is calculated according to the metric specified for the corresponding type of coupling represented by that edge. The fourth stage consisted of compacting the multiple edges between two classes into a single edge where the new weight consists of the summation of the weights assigned to the multiple edges. The final stage consisted of applying a clustering algorithm to the resulting graph.

They applied their approach on two software systems, and compared their results with the ones obtained by applying the same clustering algorithm to class dependency graphs that were not refined using architectural clues. Their approach performed better than the traditional approach both in accuracy and efficiency.

2.2.2 Similarity Matrix Based Clustering Techniques

The similarity matrix based clustering techniques represent the proximity between entities based on the attributes that describe them (discussed earlier in Section 2.1). Several approaches for similarity matrix based clustering techniques have been proposed in the literature, with the main difference among being the type of attributes used to create the clusters.

Schwanke introduced the concept of information sharing heuristics to define the attributes based on procedures in the system that are used to group entities in clusters [Schwanke 91]. The information sharing heuristic is based on the idea that two procedures should be grouped together based on the degree to which they share information through non-local names. Non-local names are the names of procedures, macros, type definitions, or variables that appear in more than one procedure's scope. The non-local names were distinguished by unique identifiers due to the fact that they might have multiple declarations in different scopes. One important contribution of their work is the ability to identify non-local names and use them as attributes to cluster the entities. Non-local names are cross references between the entities in the static dependency graph. The cross-reference between two entities is used as attribute for both the entities (e.g., if procedure A calls procedure B then, A receives the attribute as "B" and B receives the attribute as "Called by A"). The authors applied their approach to a software system written in C. The result was evaluated by three architects who were familiar with the system. The obtained decomposition recall reached only 60% of the original decomposition of the system.

Dugerdil used a dynamic analysis driven approach to recover the system architecture from execution traces generated from the system under study [Dugerdil 07]. The authors collected traces by exercising various scenarios of a Visual Basic system they worked with. The traces were sliced into portions of equal size, called trace samples. The clustering attributes consisted of these trace samples. In other words, two entities were considered identical if they appeared in the same samples and were not invoked in any other sample. However, before the clustering process started, omnipresent components were removed from the traces. These were the components that appeared in most samples (the threshold was 75% in their case study). The author applied his technique to a commercial application written in Visual Basic, and was able to cluster about 11% of the total entities of the system. This low number of entities was attributable to the fact that these were the only components invoked in the traces. The author stated that the results were promising but no validation was performed.

Anquetil and Lethbridge [Anquetil 99b] used file names as a clustering criterion. They made this choice after an experiment they conducted, where they asked software engineers to manually group the files of the system under study into clusters. The experiment showed that software engineers had used file names as the main clustering criterion. The authors run several other studies on the same system using various clustering criteria besides file names and concluded that file names were the best suitable clustering criterion for their system.

2.3 Discussion

Software clustering is a broad domain, which involves many different algorithms, similarity measures, and attributes. The challenge lies in identifying a suitable clustering algorithm along with the other parameters such as attributes of the entities and similarity measures. Different clustering algorithm types also have different advantages and limitations. For example, a hierarchical algorithm makes arbitrary decisions that may affect early on the end result of the clustering process. On the other hand, the computing cost of partitioning algorithms is often too high to be applicable to large systems. In addition, one has to have some knowledge of the system prior to applying the algorithm in order to propose an initial partition used to initialize the algorithms.

Also, it is very common that a clustering algorithm imposes a structure instead of recovering the one created by the initial designers of the system. It might happen that a clustering algorithm extracts clusters out of data that has no natural grouping.

Software clustering techniques uses various clustering algorithms discussed in the previous section. Also many researchers developed their own algorithm, which is build specifically for software clustering and focuses onto some predefined criteria [Tzerpos 00, Mancoridis 99]. The comparison with various clustering elements that affect the software clustering like clustering algorithm, similarity metric and omnipresent component elimination has also provided in the literature [Anquetil 99a, Mancoridis 99, Xiao 05, Tzerpos 00]. The comparison is made with different software systems that favours one clustering element over the other, which does not give clear benefits for the usage of one specific clustering algorithm or similarity metric. However, the improvement of clustering results due to omnipresent component elimination is confirmed through many experimental analyses [Mancoridis 99, Müller 93, Wen 05].

Chapter 3 Software Features

One of the novel contributions of this thesis is the use of software features as a new clustering attribute. In other words, we cluster a large number of a system's entities based on the degree to which they collaborate with each other to implement software features. Therefore, we have embarked on to study the concept of software features to understand their characteristics and how they are related to other system artefacts such as system architecture.

The remaining parts of this chapter are organized as follows: In the next section, we review the literature on the use of software features to understand various definitions provided in it. The section ends with summery of feature definitions. The motivations behind using features as a clustering attributes is discussed in Section 3.2, followed with the challenges that are associated with this choice.

3.1 What is a Software Feature?

In this section, we review literature on what constitutes a software feature, and how software features have been related to other system artefacts. We focus our survey in particular on three areas in which features are used extensively, namely, feature-driven software development, feature location, and feature interaction.

3.1.1 Feature-Oriented Software Development

Feature-oriented software development is a research area that explores the relationship between a system's features and functional and non-functional system requirements [Davis 82]. Perhaps, the most widely used definition of a software feature is the one given in [American Heritage 85], which describes a feature as "a prominent or distinctive user-visible aspect, quality, or characteristic of a system or systems." Although this definition is too broad to be used in our research, it refers to the fact that a software feature should represent an important aspect of a system that can be observed by an external user. This aspect could be in the form of a function of the system or one of its quality attributes (e.g., speed by which the system responds to a user query). Kang et al. [Kang 90] refine this definition by adding that software features are a powerful tool to uncover the common aspects of a particular domain. They suggest that software features capture abstract concepts of a domain that can be reused when developing applications of the same domain. Similarly, Kang et al. [Kang 98], propose using software features to capture the commonalities and differences that exist among software applications in terms of the features they provide. The authors argue that this feature-based analysis can lead to creating domain architectures and components that can be easily reused. In addition, Lee et al. stress the importance of using feature models in the development of software product lines since product lines typically require reusing existing components and that software features tend to be ideal to capture abstract domain concepts [Lee 02]. The authors note that there is a tendency to consider a software feature and a software concept as the same. They also assume that a software concept represents an abstract construct that are identified from the internal viewpoint such as functions, objects and software aspects, whereas features are externally visible characteristics of the system.

Griss et al. argue that software features are closely related to the concept of use cases defined in UML [Griss 98]. According to them, a use case model is something that is designed by a system engineer with a user perspective in mind, whereas a feature model is designed by a domain engineer for the purpose of reusing the application. They add that the main difference between a use case model and a software feature model is that a use case model provides "what" the system is capable of doing, whereas a feature model puts an emphasis on the knowledge about "which" system functionality to be selected when engineering a new system in the same domain.

Finally, Liu and Mei propose that a "feature is a higher-level abstraction of a set of relevant detailed software requirements, and is perceivable by users (or customers)" [Liu 03]. The authors support the idea that features are first class entities in requirement modeling. They add that features are at a higher level of abstraction than requirements. Once the requirement specification has been organized by features, the authors propose a way to map the feature model to an architectural model. According to them, functional features can be mapped directly to a subsystem or an entity of the system.

3.1.2 Feature Location

Feature location is a reverse engineering technique that focuses on locating specific parts of a software system that implement a particular feature [Rohatgi 08], also known as software location in source code. Feature location has been shown to be useful in helping software engineers, working on solving maintenance problems, understand how software features are implemented [Eisenbarth 03].

Perhaps, one of the earliest proposed feature location techniques is the Software Reconnaissance technique introduced by Wilde and Skully [Wilde 95]. Their approach consists of generating execution traces by exercising various features of the system. The traces are then compared and the components that implement particular features are identified. Although the authors have not provided an explicit definition of what they consider as a feature, it is clear from the experiment they conducted that they have considered features as any function of the system that can be triggered by an external user.

Eisenbarth et al. proposed a feature location approach that combines static and dynamic analysis techniques. According to them a feature is "a realized functional requirement of a system". They argue that a feature is an "observable behaviour of the system that can be triggered by the user".

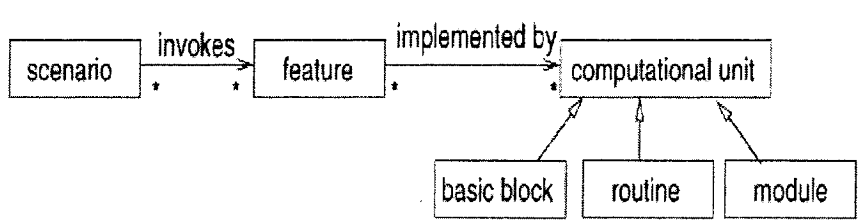

Figure 3.1. A feature model (taken from [Eisenbarth 03])

As shown in Figure 3.1, the authors show the relationship between a software feature, a scenario, and a computation unit. A scenario is defined here as the way the system is expected to be used by the user. Based on this, a scenario can invoke many features of a system. For example, editing picture using a drawing tool might involve loading the picture to memory, editing it, and saving it to disk. As such, the same software feature can be involved in multiple usage scenarios. Computational unit refers to the source code units that are executed by exercising the feature on the system. The authors add that a feature might require many execution traces in order to fully understand how it is implemented.

Antoniol and Gael-Gueheneuc define a feature as "a requirement of a program that a user can exercise and which produces an observable behaviour" [Antoniol 06]. Similarly, Einsberg et al. state that features are "defined as behaviours that are observable to users at their particular level of interaction with the system" [Eisenberg 05]. Unlike Einsbarth et al., they consider one to one mapping between features and execution traces.

Poshyvanyk et al. define features as "the concepts that represent a functionality of a system accessible and visible to the users, usually captured by the requirements explicitly" [Poshyvanyk 07]. They further state that "a feature links program architecture with its dynamic behaviour". In other words, a software feature exercised on a system can exhibit the behaviour of the system from its architectural point of view.

3.1.3 Feature Interaction

Feature interaction is the study of how features influence or conflict with other features of the same system. Feature interaction techniques have traditionally been discussed in the context of telecommunication systems due to the fact that the features of such systems represent a modification or an enhancement of the same baseline feature set.

Research in the feature interaction adopts a common definition that considers features as incrementally added functionalities to the basic functionality of a system [Calder 03]. Zave, for example, defines a software feature as "an increment of functionality, usually with a coherent purpose. If a system description is organized by features, then it probably takes the form B + F1 + F2 + F3..., where B is a base description, each Fi is a feature module, and + denotes some feature-composition operation" [Zave 99]. This definition proposes that system evolution is usually feature driven through continuous enhancement of existing features or addition of new ones based on the existing ones. This suggests that software features can be highly coupled (i.e., the execution of one feature depends on another one).

3.1.4 Summary

From the above discussion, there is a strong agreement among researchers that a software feature should represent a behavioural aspect of the system that is observable by an external user. In this thesis, we also define a software feature as an observable behaviour of a system that represents a particular functionality. At a high-level, there is a strong relationship between software features and the concept of use cases defined in UML [Jacobson 94], since use cases also define the system functionality from the user's perspective. The main difference is that software features capture high-level domain concepts whereas use cases focus on mapping these concepts to system functionality as noted by Griss et al. [Griss 98]. Similar to the concept of use cases, a software feature can cover many functional requirements of the system, and that a particular requirement can be involved in many features.

In addition, software features play an important role during functional decomposition of the system into subsystems by having a subsystem encompass the components that implement related features (i.e., the ones that represent common functionality of the system).

Moreover, there are various relations between software features such as dependency relationships. Exercising a software feature might involve exercising other features on which it depends.

3.2 Software Features as a Clustering Attribute

From the above discussion, we demonstrated that software features represent abstract concepts of a system. This has led us to believe that they are more suitable for recovering high-level and more abstract views of a system from low-level implementation details than mere source code constructs used by many other clustering approaches.

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., [Wilde 95]).

At first glance, one might make several objections to using software features:

- Execution traces represent a partial model of the system, with the components being invoked constituting a subset of the entire set of the system components. Therefore, feature-based clustering does not cover the system in its entirety.

- The number of software features can affect the result of the clustering process. The more features we have in a system, the better will be the result of the clustering approach. But since it would be impractical to exercise all possible features of a system, one has to determine an acceptable threshold that can lead to a satisfactory decomposition of the overall system.

- The choice of software features is crucial to the success of the clustering process. For example, if we only choose features that represent similar functions of the system then we will most likely end up with a few clusters that contain most of the elements. Therefore it is necessary to derive a balanced set of features that cover various aspects of the system.

We address the first issue by adopting a two-phase clustering process rather than traditional clustering methods. Traditional one-phase algorithms start their clustering process only once all system entities and their relations are available. In comparison to these approaches, our two-phase approach starts in the first phase, which is also the most important one, by determining a skeleton decomposition. The clusters belonging to the skeleton form the core clusters of the system. The remaining components are then incrementally added to the skeleton during the second phase. The remaining components are added by measuring their similarity with the ones already grouped in the skeleton decomposition. It should also be noted that during this phase, new clusters can also be created, if any remaining component does not hold relationship with none of the already clustered component.

For the second issue, which deals with the number of features, a threshold-based approach can be derived from experimental analysis in order to determine the impact of code coverage achieved on the resulting decomposition.

Finally, to address the third issue, there is a need to cover different system functions to be able to identify the basic feature set of a system. The selection of software features can and should be guided by domain experts or based on any other form of domain knowledge, e.g., available documentation or product usage guide.

Chapter 4 Clustering Approach

In this chapter, we present the main contribution of the thesis, which consists of a new clustering approach that uses software features as its main clustering criterion. Unlike existing techniques, our approach groups a system's entities based on the degree to which they collaborate with each other rather than mere structural relationships extracted from the source code.

The remainder of this chapter is organized as follows. In Section 4.1, we discuss the overall approach. In Section 4.2, we present the first phase of our approach which consists of building a skeleton decomposition of the system using software features. The second phase of the clustering process is presented in Section 4.3. Finally, we conclude the chapter in Section 4.4.

4.1 Overall Approach

The approach proposed in this thesis emphasizes the use of both dynamic and static analysis techniques to support software clustering. It compasses two main phases. The first phase, which is also the most important, consists of determining a skeleton decomposition of a system. The clusters belonging to the skeleton form the core clusters of a system. We achieve this by measuring the similarity between a system's entities by identifying the number of features they implement.

In the next phase, we apply static dependency analysis to further cluster the non core entities of the system, also known as "orphans". Orphans are the entities that were not clustered in the skeleton decomposition. We achieve this using the orphan adoption algorithm presented by Tzerpos et al. [Tzerpos 00], which is an algorithm that is based on analyzing static dependencies among entities in order to determine the skeleton clusters to which they should belong.

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

4.2 Skeleton Decomposition

Figure 4.1 shows the steps involved in the first phase of our clustering approach, the skeleton creation based on the software features. The detailed steps are presented in the subsequent sections.

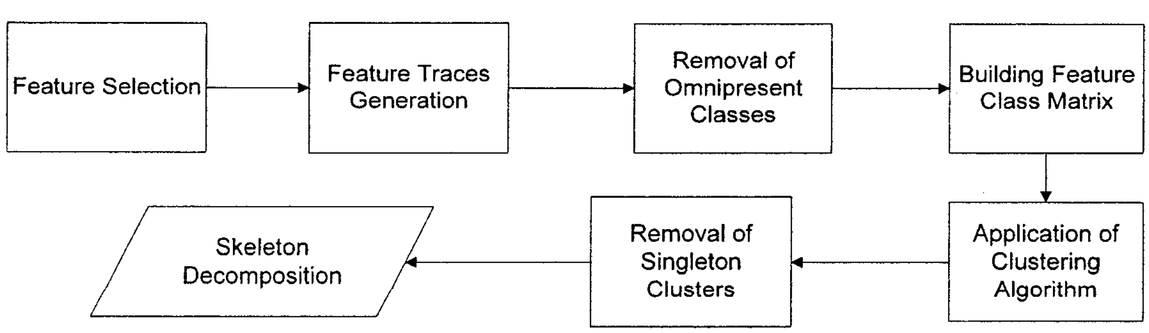

Figure 4.1. Phase 1: Skeleton building

4.2.1 Feature Selection

In this step, we select the software features that will be used during the formation of the skeleton decomposition. As discussed previously, it is important to select features that execute a wide range of different system functionalities. If the selected features perform only a narrow subset of the overall functions found in the system 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 thesis, we use any available documentation to derive a balanced set of software features.

4.2.2 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 events of interests. However, the use of a debugger slows down considerably the execution of the system and should be avoided for large-scale software systems [Hamou-Lhadj 06, Xiao 05]. 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.

4.2.3 Clustering of Omnipresent Classes

Software systems often contain entities that act as mere utilities to support feature specific components. They are referred to as omnipresent components [Müller 93]. 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 [Müller 93]. Wen and Tzerpos [Wen 05] also agreed that removing omnipresent components can significantly improve the clustering result. Therefore, we have decided to remove omnipresent system classes 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 [Hamou-Lhadj 06]. Wen and Tzerpos considered omnipresent components to be the ones that are connected to a large number of subsystems [Wen 05]. In [Mancoridis 99], Mancoridis et al. presented a heuristic based approach for detecting omnipresent components.

Table 4.1. Class-feature matrix

	F1	**F2**	**...**	**Fm**
C1	1	0	...	1
C2	1	1	...	0
...	...	...	...	...
Cn	0	1	...	0

4.2.4 Building the Class-Feature Matrix

A class-feature 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. For example, Table 4.1 shows a matrix where the rows represent the distinct classes invoked in all traces, where as the columns refer to the feature-traces generated in the previous step.

4.2.5 Applying the Clustering Algorithm

The next step in this phase consists of selecting a clustering algorithm and applying it to the class-feature matrix. As discussed in Chapter 2, clustering algorithms can be grouped into two main categories: Partitioning and hierarchical clustering algorithms. In this thesis, we apply an agglomerative hierarchical algorithm, with complete linkage as an updating rule, and the Jaccard coefficient as a similarity metric. We selected this algorithm and similarity metrics because complete linkage as well as Jaccard distance metrics have been shown to perform better compared to the other schemes [Anquetil 99a].

The result of the clustering algorithm can be visualized through a dendrogram (Figure 4.2), 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.

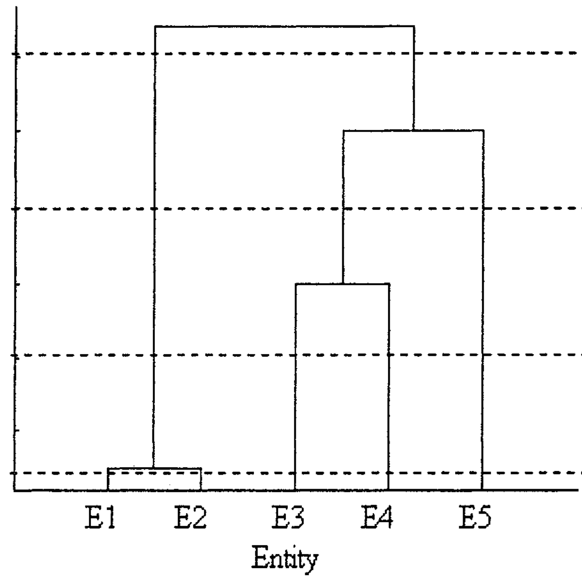

Figure 4.2. A dendrogram with various cut points shown as dashed lines

4.2.6 Removing Singleton Clusters

The final step of constructing the skeleton decomposition is the need to analyze the clusters resulting from the previous step and remove the ones that contain one single class (i.e. singleton clusters). The rationale behind this is that singleton clusters are clusters without any similarity to the formed clusters. These clusters can often lead to a deterioration of the skeleton stability as shown by Tzerpos et al. in [Tzerpos 00]. 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 phase of the clustering process).

4.3 Full Decomposition - The Orphan Adoption Algorithm

The second phase of our approach is based on an orphan adoption algorithm [Tzerpos 00]. The algorithm takes as input the skeleton clusters derived in the first phase of the algorithm and a component dependency graph of the orphans as shown in Figure 4.3. In what follows we describe this orphan adoption algorithm in more detail.

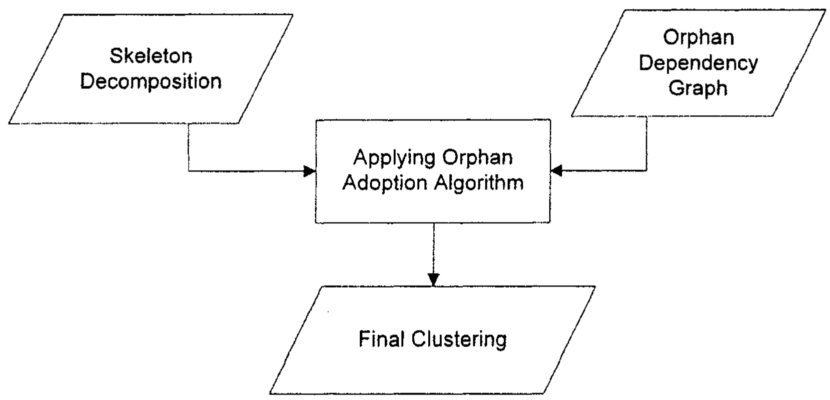

Figure 4.3. Phase 2: Orphan adoption

4.3.1 Building a Component Dependency Graph

After identifying the orphans, the next activity is to derive the components dependency graph of the orphans. The dependencies used in the orphan adoption algorithm are structural relations between the components. The structural relations can be of various types such as referencing of variables, inheritance, instantiation of other components, etc. The dependency graph includes the links representing the structural relations among the classes and the classes are the nodes of the graph. The graph is a directed graph with at most two edges between two classes and all edges have same weight.

Extraction of structural relationship is an automatic task which is supported by numerous tools (e.g., SA4J[1]). The extraction tools differ by their support of various technologies

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

and programming languages.

4.3.2 Clustering of Orphan Components

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.

Chapter 5 Evaluation

In this chapter, we present two case studies to demonstrate the applicability of our approach. We first introduce the target systems used for the case studies. Then, we describe in detail the application of our clustering approach. Finally, we compare the results of our clustering decomposition with the results obtained by manual analysis through domain experts (the designers of the target system).

5.1 Target Systems

We applied our approach to two Java-based object-oriented software systems: Weka (ver. 3.0)[2], and JHotDraw (ver 5.1)[3]. Weka is a machine learning tool that provides several algorithms for classification, regression techniques, clustering, and association rules. JHotDraw is a system that provides a graphical user interface (GUI) framework support for graphical activities. The framework can be extended to support new graphical capabilities by adding extra functionality to it. The tool has been designed using many well known design patterns, which enhances its reusability. Table 5.1 shows the characteristics of both systems.

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

[3] JHotDraw : http://sourceforge.net/projects/jhotdraw

We selected Weka and JHotDraw because both systems are well documented. Their documentation also includes architectural properties that are later used to validate our approach and to identify the omnipresent classes.

Table 5.1. Target system details

	Number of Packages	Number of Classes	KLOC
Weka	09	142	95
JHotDraw	11	155	17

Figure 5.1 shows Weka architecture, which consists of the following packages: `associations`, `clusterers`, `core`, `estimators`, `filters`, `attributeSelection` and `classifiers`. The `classifiers` package contains two additional packages namely `j48` and `m5`.

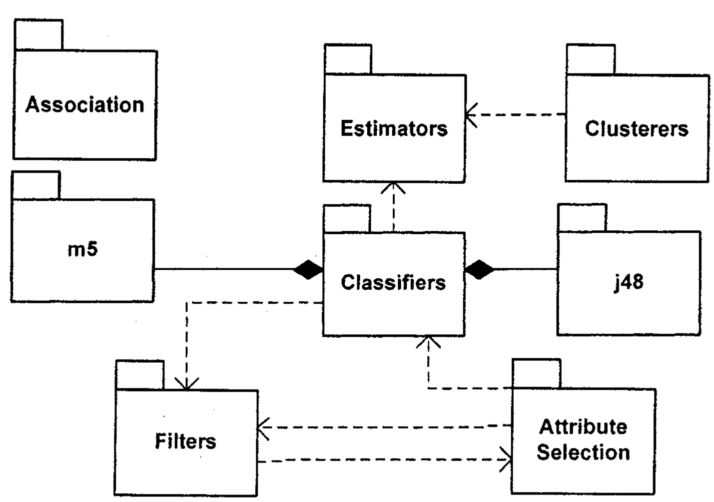

Figure 5.1. Weka architecture

The role of each package is as follows:

- `associations`: This package contains two classes that implement the Apriori machine learning algorithm [Agrawal 94].

- `attributeSelection`: The classes of this package implement techniques from reducing the dimensionality of data set.

- `classifiers`: This package contains several classes that implement the following classification algorithms: SMO [Platt 99], Naïve Bayes [John 95], ZeroR [Holte 93], Decision Stump, Linear regression, and OneR [Witten 05]. It also contains two other sub packages `J48` and `M5`. `J48` contains classes that implement the J48 (also known as the C45) classification algorithm [Quinlan 93], while `M5` classes are used to implement the M5Prime classification algorithm [Wang 97].

- `clusterers`: This package contains classes that implement two clustering algorithms supported by Weka, namely, Cobweb [Fisher 87] and EM [Dempster 77].

- `filters`: This package contains classes that allow preprocessing the data that is used as input for the machine learning algorithms.

- `estimators`: This package contains classes that compute various types of probabilities needed for the clustering and classification algorithms.

- `core`: The `core` package contains general-purpose utilities. It is not shown in Figure 5.1 to avoiding cluttering the figure, since all other packages depend on it. It should also be noted that the `associations` package is isolated in Figure 5.1 because it only depends on the `core` package.

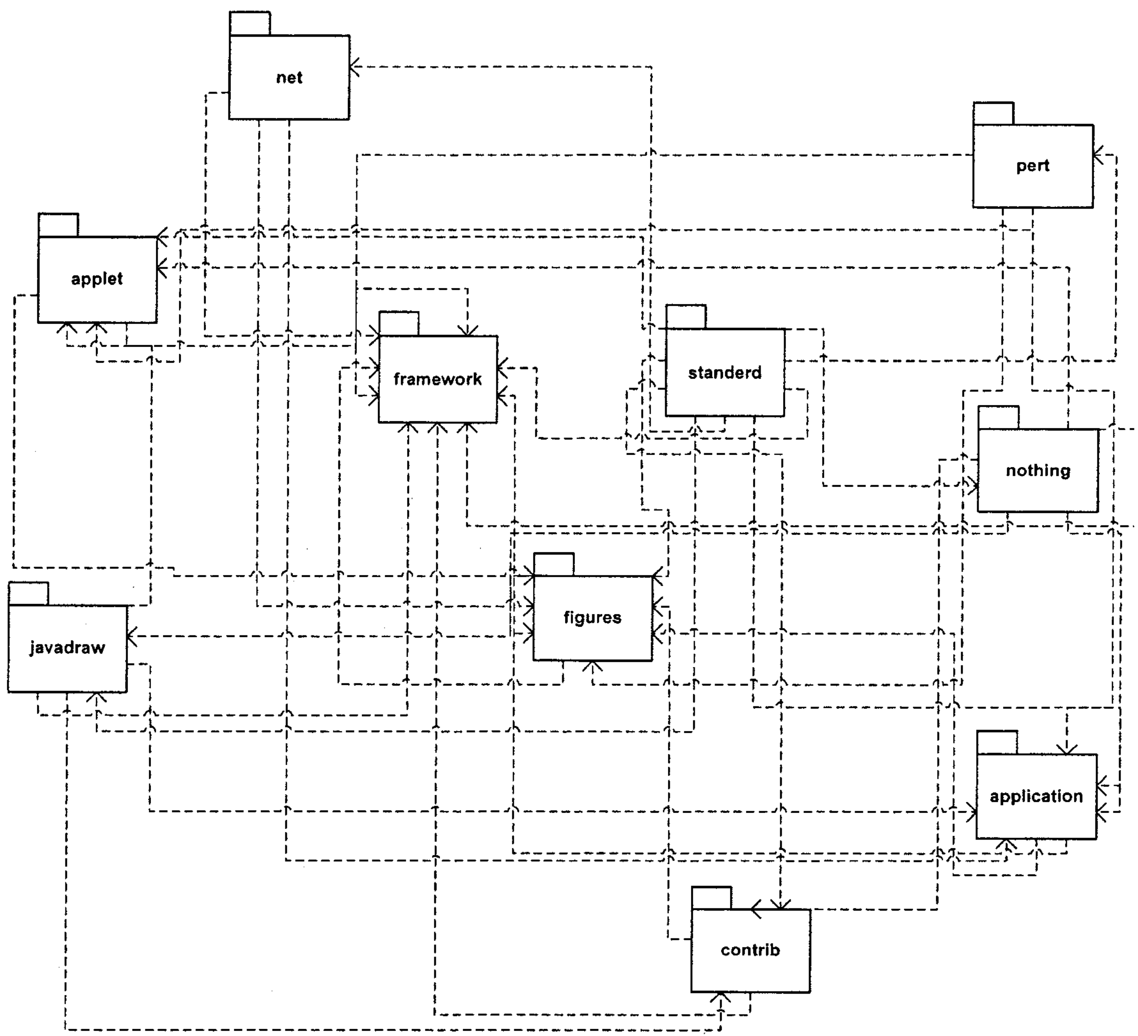

Figure 5.2. JHotDraw architecture

Figure 5.2 shows the JHotDraw architecture, which consists of the following packages: `applet`, `application`, `contrib`, `figures`, `framework`, `javadraw`, `net`, `nothing`, `pert`, `standard` and `util`. Package `sample` includes `javadraw`, `net`, `nothing` and `pert` packages, but it does not contain any class in it. For this purpose, we have excluded `sample` package from the analysis and considered all its sub packages as separate packages.

The role of JHotDraw packages is as follows:

- `applet`: The package provides two classes to run JHotDraw as an applet.
- `application`: This package contains one class to run JHotDraw as a standalone application.
- `contrib`: This package contains classes from other implementers that contribute to JHotDraw by adding new graphical capabilities, etc.
- `figures`: This package contains classes that implement the many graphical capabilities (e.g., drawing circles, rectangles, etc.) provided by JHotDraw.
- `framework`: This package contains abstract classes and interfaces that can be extended by other contributors to specialize JHotDraw capabilities.
- `Javadraw, net, nothing` and `pert` includes classes for sample applets and/or applications.
- `standard`: The package includes the standard implementation for the classes provided in framework package. The implementation of interfaces of framework package is provided in the abstract classes of this package.

- `util`: The package includes utility classes used by all other packages of JHotDraw. Similar to the Weka core package, we decided not show the `util` package in Figure 5.2 to avoid cluttering.

5.2 Constructing the Skeleton Decomposition

In this section, we apply the steps for constructing the skeleton decomposition to Weka and JHotDraw and show the resulting skeletons generated by our approach.

5.2.1 Feature Selection

Table 5.2 shows the software features selected for Weka. We carefully selected features that covered most of Weka's machine learning algorithms and data filters to ensure a balanced set of features. We used Weka documentation as the main source of information for identifying these features.

Table 5.2. Weka features used in this study

Feature	**Feature Name**
F1	Cobweb clustering algorithm
F2	EM clustering algorithm
F3	Ibk classification algorithm
F4	OneR classification algorithm
F5	Decision table classification algorithm
F6	J48 (C4.5) classification algorithm

F7	SMO classification algorithm
F8	Naïve Bayes classification algorithm
F9	ZeroR classification algorithm
F10	Decision stump classification algorithm
F11	Linear regression classification algorithm
F12	M5Prime classification algorithm
F13	Apriori association algorithm
F14	Attribute Filter
F15	Add Filter
F16	Merge Two Values Filter
F17	Instance Filter
F18	Swap Attribute Values Filter
F19	Split Dataset Filter
F20	Numeric Transform Filter

Similarly, we selected several features of JHotDraw based on the tool's documentation. The list of JHotDraw features are listed in Table 5.3.

Table 5.3. JHotDraw features used in this study

Feature	Feature Name	Feature	Feature Name
F1	Align Bottom	F29	New Window
F2	Align Center	F30	No arrow
F3	Align Left	F31	Open an Application
F4	Align Middle	F32	Open File
F5	Align Right	F33	Open
F6	Align Top	F34	Paste
F7	Arrow at back	F35	Pattern
F8	Arrow at both	F36	Pen Color
F9	Arrow at front	F37	Polygon
F10	Border	F38	Print
F11	Buffered Update	F39	Put Images
F12	Close an Application	F40	Rounded square
F13	Connecting Line	F41	Save as Serializable
F14	Copy	F42	Save as
F15	Cut	F43	Scribble
F16	Delete	F44	Select Text
F17	Draw Circle	F45	Select Multiple Objects
F18	Draw Line	F46	Select Object

F19	Duplicate	F47	Send Back
F20	Elbow Connecting Line	F48	Send To front
F21	Exit	F49	Simple Update
F22	Fill Color	F50	Square
F23	Font color changed	F51	Start Animation
F24	Font size change	F52	Stop Animation
F25	Font Style Changed	F53	Toggle
F26	Font Change	F54	Ungroup Objects
F27	Group Object	F55	Write Text
F28	New Document	F56	WriteText2

5.2.2 Trace Generation

We generated an execution trace for each of the features shown in Table 5.2 and Table 5.3. For this purpose, we instrumented the Weka and JHotDraw class files using the BIT (Bytecode Instrumentation Toolkit) framework [Lee 97] by inserting probes at entry of every method in the source code, 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 to be stored.

The number of classes involved in traces of Weka and JHotDraw system is 59 and 88 respectively, which represent 47% and 65% of the total number of classes of Weka and JHotDraw.

5.2.3 Omnipresent Class Identification

Most of the removal of omnipresent classes can be performed automatically based on the omnipresent detection approach introduced by Hamou-Lhadj and Lethbridge [Hamou-Lhadj 06]. The approach is based on fan-in analysis. However, a quick analysis of the Weka and JHotDraw documentation revealed that the packages `core` (of Weka) and `util` (of JHotDraw) are used by all other packages within the system, resulting in a high fan-in for these particular packages. We have, therefore, decided to categorize these packages manually as utility (i.e., omnipresent) packages and moved them into a utility cluster created for each system, and eliminated all classes within these package from further analysis.

5.2.4 Class-Feature Matrix

Building the class-feature matrix is a straightforward process, since we have already analyzed the feature-traces and collected the distinct set of classes involved in the different features. As previously mentioned, a cell (i, j) is set to 1 if a class in the i^{th} row is invoked in the execution trace corresponding to the feature of the j^{th} column. It is set to 0 otherwise.

The class-feature matrix for Weka is 59x20, i.e., 59 classes and 20 features. The dimension of the JHotDraw class-feature matrix is 88x56 (88 classes and 56 features).

5.2.5 Applying the Clustering Algorithm

The next step is to apply the clustering algorithm to the class-feature matrices for the Weka and JHotDraw system. As discussed in Section 4.2.5, we used an agglomerative hierarchical algorithm, with complete linkage as an updating rule, and the Jaccard coefficient as a similarity metric. The result of the hierarchical clustering of Weka and JHotDraw is shown in the dendrograms of Figure 5.3 and 5.4, respectively.

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 by Anquetil et al. in [Anquetil 99a], there exists typically more than one potential cut point value. They proposed to cut the 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.

After analyzing manually the content of both dendrograms, we decided to cut the tree at point P1 shown in Figure 5.3 and 5.4, which balances the number of clusters and the number of classes in each cluster.

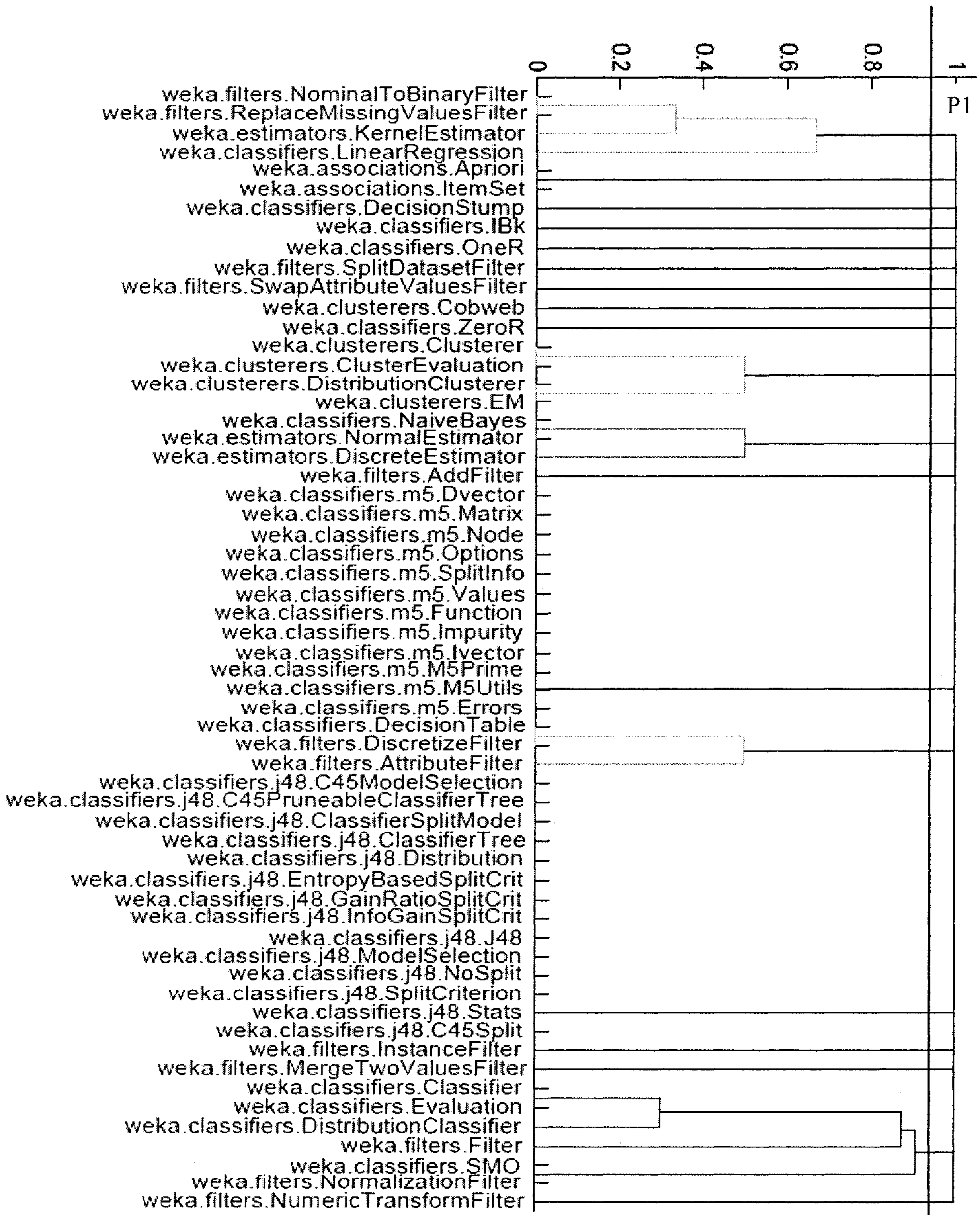

Figure 5.3. Dendrogram generated by applying the clustering

algorithm to Weka classes

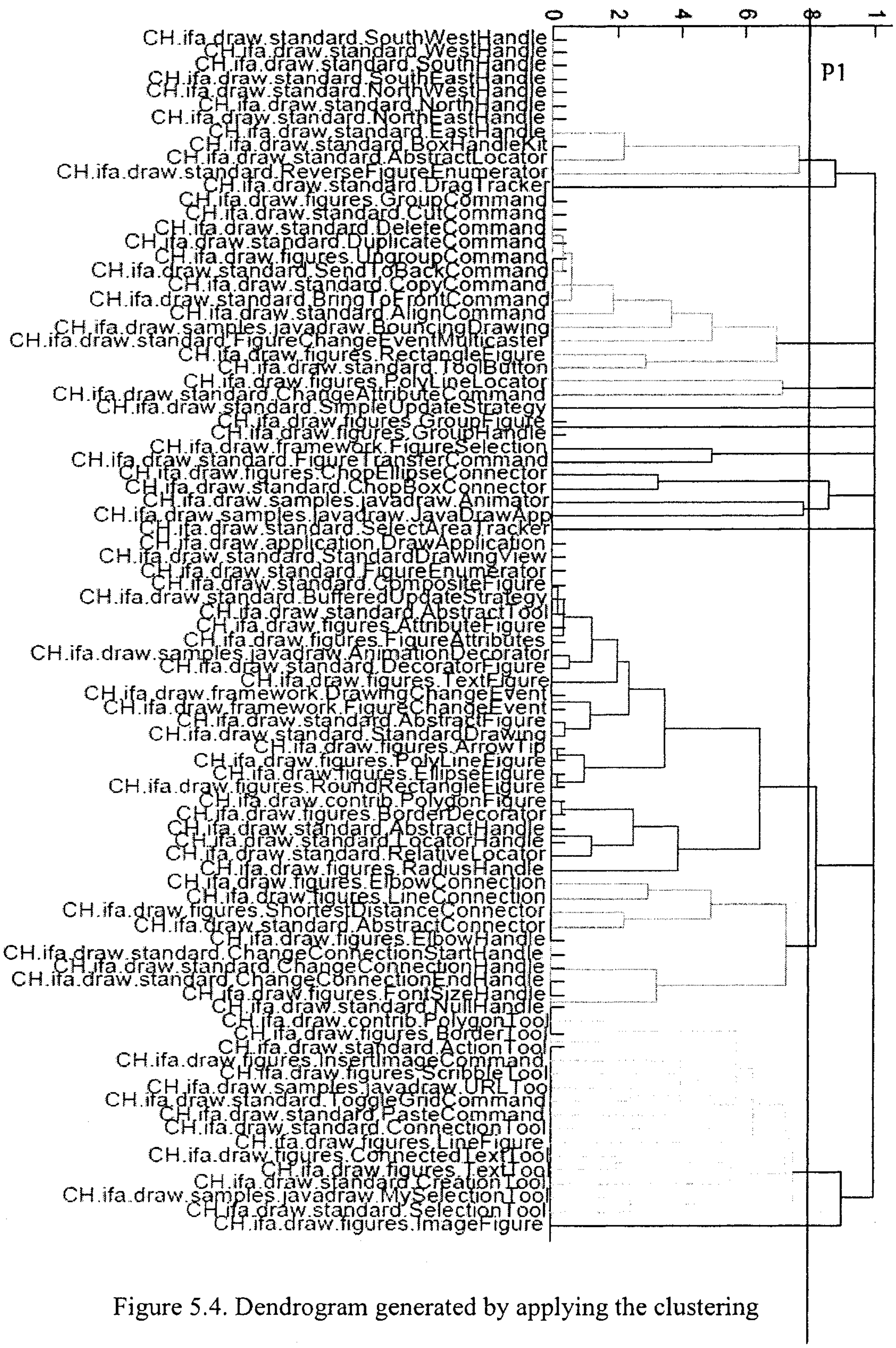

Figure 5.4. Dendrogram generated by applying the clustering algorithm to JHotDraw classes

5.2.6 Removing Singleton Clusters

During the singleton cluster removal steps, we remove now all those clusters that contain only one single class, i.e., singleton clusters from the list of clusters identified in the previous step. The classes in these singleton clusters are moved to the pool of orphans to be further processed later as part of the static analysis.

As a result, the remaining core skeleton of the Weka system consisted of eight clusters (C1 to C8), which are shown in Table 5.4. The number of classes that were clustered during the skeleton decomposition phase is 48, which represents 32.65% of the total number of classes of the system under analysis.

Table 5.4. Weka skeleton clusters

Cluster	Classes
C1	weka.estimators.DiscreteEstimator weka.classifiers.NaiveBayes weka.estimators.NormalEstimator
C2	weka.filters.AttributeFilter weka.classifiers.DecisionTable weka.filters.DiscretizeFilter
C3	weka.associations.Apriori weka.associations.ItemSet
C4	weka.classifiers.m5.M5Prime weka.classifiers.m5.Options 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.clusterers.EM weka.clusterers.DistributionClusterer weka.clusterers.Clusterer weka.clusterers.Clusterevaluation
C6	weka.classifiers.LinearRegression weka.filters.NominalToBinaryFilter weka.filters.ReplaceMissingValuesFilter weka.estimators.KernelEstimator
C7	weka.classifiers.j48.J48 weka.classifiers.j48.C45ModelSelection 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.InfoGainSplitCrit weka.classifiers.j48.SplitCriterion weka.classifiers.j48.GainRatioSplitCrit weka.classifiers.j48.Stats
C8	weka.classifiers.DistributionClassifier weka.classifiers.Classifier weka.classifiers.Evaluation weka.filters.Filter weka.classifiers.SMO weka.filters.NormalizationFilter

The resulting skeleton decomposition of JHotDraw consists of 10 clusters (see Table 5.5). The number of JHotDraw classes clustered during this phase is 84, which represents 62.22% of the total number of classes of the system under analysis.

Table 5.5. JHotDraw Skeleton clusters

Cluster	Classes
C1	CH.ifa.draw.standard.SouthWestHandle CH.ifa.draw.standard.WestHandle CH.ifa.draw.standard.SouthHandle CH.ifa.draw.standard.SouthEastHandle CH.ifa.draw.standard.NorthWestHandle CH.ifa.draw.standard.NorthHandle CH.ifa.draw.standard.NorthEastHandle CH.ifa.draw.standard.EastHandle CH.ifa.draw.standard.BoxHandleKit CH.ifa.draw.standard.AbstractLocator CH.ifa.draw.standard.ReverseFigureEnumerator
C2	CH.ifa.draw.figures.GroupCommand CH.ifa.draw.standard.CutCommand CH.ifa.draw.standard.DeleteCommand CH.ifa.draw.standard.DuplicateCommand CH.ifa.draw.figures.UngroupCommand CH.ifa.draw.standard.SendToBackCommand CH.ifa.draw.standard.CopyCommand CH.ifa.draw.standard.BringToFrontCommand CH.ifa.draw.standard.AlignCommand CH.ifa.draw.samples.javadraw.BouncingDrawing CH.ifa.draw.standard.FigureChangeEventMulticaster CH.ifa.draw.figures.RectangleFigure CH.ifa.draw.standard.ToolButton
C3	CH.ifa.draw.figures.PolyLineLocator CH.ifa.draw.standard.ChangeAttributeCommand

C4	CH.ifa.draw.figures.GroupFigure CH.ifa.draw.figures.GroupHandle
C5	CH.ifa.draw.framework.FigureSelection CH.ifa.draw.standard.FigureTransferCommand
C6	CH.ifa.draw.figures.ChopEllipseConnector CH.ifa.draw.standard.ChopBoxConnector
C7	CH.ifa.draw.samples.javadraw.Animator CH.ifa.draw.samples.javadraw.JavaDrawApp
C8	CH.ifa.draw.application.DrawApplication CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.standard.FigureEnumerator CH.ifa.draw.standard.CompositeFigure CH.ifa.draw.standard.BufferedUpdateStrategy CH.ifa.draw.standard.AbstractTool CH.ifa.draw.figures.AttributeFigure CH.ifa.draw.figures.FigureAttributes CH.ifa.draw.samples.javadraw.AnimationDecorator CH.ifa.draw.standard.DecoratorFigure CH.ifa.draw.figures.TextFigure CH.ifa.draw.framework.DrawingChangeEvent CH.ifa.draw.framework.FigureChangeEvent CH.ifa.draw.standard.AbstractFigure CH.ifa.draw.standard.StandardDrawing CH.ifa.draw.figures.ArrowTip CH.ifa.draw.figures.PolyLineFigure CH.ifa.draw.figures.EllipseFigure CH.ifa.draw.figures.RoundRectangleFigure CH.ifa.draw.contrib.PolygonFigure CH.ifa.draw.figures.BorderDecorator CH.ifa.draw.standard.AbstractHandle CH.ifa.draw.standard.LocatorHandle CH.ifa.draw.standard.RelativeLocator

	CH.ifa.draw.figures.RadiusHandle
C9	CH.ifa.draw.figures.ElbowConnection CH.ifa.draw.figures.LineConnection CH.ifa.draw.figures.ShortestDistanceConnector CH.ifa.draw.standard.AbstractConnector CH.ifa.draw.figures.ElbowHandle CH.ifa.draw.standard.ChangeConnectionStartHandle CH.ifa.draw.standard.ChangeConnectionHandle CH.ifa.draw.standard.ChangeConnectionEndHandle CH.ifa.draw.figures.FontSizeHandle CH.ifa.draw.standard.NullHandle
C10	CH.ifa.draw.contrib.PolygonTool CH.ifa.draw.figures.BorderTool CH.ifa.draw.standard.ActionTool CH.ifa.draw.figures.InsertImageCommand CH.ifa.draw.figures.ScribbleTool CH.ifa.draw.samples.javadraw.URLTool CH.ifa.draw.standard.ToggleGridCommand CH.ifa.draw.standard.PasteCommand CH.ifa.draw.standard.ConnectionTool CH.ifa.draw.figures.LineFigure CH.ifa.draw.figures.ConnectedTextTool CH.ifa.draw.figures.TextTool CH.ifa.draw.standard.CreationTool CH.ifa.draw.samples.javadraw.MySelectionTool CH.ifa.draw.standard.SelectionTool

5.3 Applying the Orphan Adoption Algorithm

In the second step of our approach, we apply the orphan adoption algorithm to cluster the remaining classes by assigning them to either an existing or a new cluster.

5.3.1 Building the Class Dependency Graph

We built a class dependency graph using a tool called Javex[4] to extract dependencies of orphan classes. Javex is a fact extractor for Java based software systems. A fact consists of the two classes and the relationship between them. An example of a fact is "<Relation Type> <entity1> <entity2>", which indicates that entity1 and entity2 of the system are related using RelationType.

Javex takes into account many dependencies that may exist between two classes such as method calls, associations, inheritance relationships, etc. The facts were saved in TA (Tupple Attribute)[5] format, which is a common format in reverse engineering for representing facts. It should be noted that the class dependency graph includes the core classes as well as the orphan classes.

The implementation of the orphan adoption algorithm was provided to us by the author of the algorithm, Vassilios Tzerpos. However, his implementation takes as input facts expressed in RSF (Rigi Source Format) format [Müller 93], which is another common data exchange format in the area of reverse engineering. We, therefore, had to create a converter that converts TA to RSF.

---

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

[5] TA : http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm

The resulting RSF files for both Weka and JHotDraw were fed to the orphan adoption algorithm along with the skeleton decomposition of both systems.

5.3.2 Generation of the Final Decomposition

The orphan adoption algorithm works by assigning each orphan to the suitable core cluster. If the algorithm cannot find a suitable cluster then it puts the orphan into a newly created cluster called "orphan container".

When applied to Weka, the final decomposition resulted in eight clusters, which is the same number of clusters built during the skeleton decomposition phase. In other words, all orphans were successfully placed in one of the core clusters.

The final decomposition of JHotDraw contained 11 clusters. An additional cluster, "orphan container", was created during the orphan adoption phase, containing five classes that were originally categorized as orphans. Except these five classes, all other orphan classes were moved in one of the core clusters.

The complete decompositions for both systems including the clusters and their classes are presented in Appendix A.

5.4 Analysis of Results

In what follows we first compare the result of our clustering approach with an expert decomposition and then provide a more detailed analysis of the various clusters created for both systems.

5.4.1 Comparison with Expert Decomposition

For initial evaluation of our approach, we compared the results obtained from our clustering approach, with the decompositions based on the package structure of these systems. We assumed that this package structure, defined by the original developers, reflects closely their structural decomposition of these systems. The package structures for the Weka and JHotDraw system are shown in Table 5.7 and Table 5.8 respectively.

For the comparison, we measured the extent to which two given decompositions are similar or apart to each other, using the MoJoFM distance [Wen 04], which is an improved version of a MoJo metric introduced by the same authors [Tzerpos 99]. 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:

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

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.

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.

An example of applying the *MoJoFM* distance to the two partitions of Figure 5.5 is as follows:

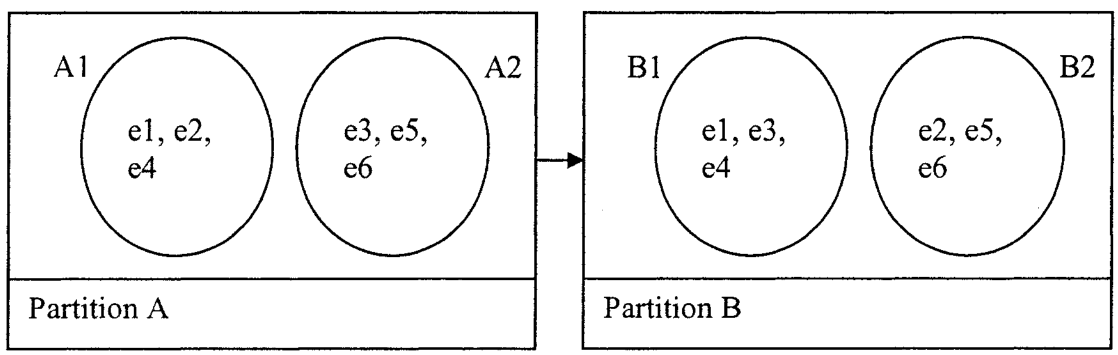

Figure 5.5. Example of two partitions

Figure 5.5 shows two partitions, A and B with A1, A2 and B1, B2 as their respective clusters. The value is calculated by first calculating the number of "move" and "join" operations needed to transform partition A into B, i.e., $mno(A,B)$. By analyzing the partition A, we can see that we need to replace elements e2 and e3 in order to obtain partition B. This can be achieved by performing two "move" operations: Moving e2 from cluster A1 to cluster A2, and moving e3 from cluster A2 to cluster A1. After the move operations, the partition A will become identical to partition B. Therefore, the cost of $mno(A,B)$ is 2 (i.e., two operations). The value of $\max(mno(\forall P,Q))$, which shows the maximum value of "move" and "join" operations needed from any partition P to Q, where the set of entities in partition P and partition Q is equal, is described in an algorithm provided by Wen and Tzerpos in [Wen 04]. By applying this algorithm, the maximum cost of transforming any partition to partition B of Figure 5.5 is 8. Therefore, MoJoFM(A, B) is $(1-(2/8))\times 100\% = 75\%$. In other words, partition A is 75% similar to partition B.

Table 5.6 shows the result of comparing Weka and JHotDraw decompositions, which are recovered by our clustering algorithm, to the expert decompositions.

Table 5.6. The result of comparing Weka and JHotDraw extracted decompositions with the expert decompositions using MoJoFM

Target System	**MoJoFM**
Weka	87.83%
JHotDraw	36.8%

As shown in this table, the MoJoFM values indicate a significant difference between our recovered decomposition and the package decomposition provided by the system developer. The results of this comparison shows that the our decomposition results in a behavioural interactions among classes that might provide additional insights to support the comprehension of the system behaviour and the recovery of architectural views.

In what follows, we further compare the internal quality of cluster results, by superimposing the resulting clusters onto the Weka and JHotDraw architectures (shown in Tables 5.7 and 5.8, respectively). For example, the cluster C1 (Table 5.7), which was recovered by our approach contains classes that belong to the Weka packages: `estimators` and `classifiers`.

5.4.2 Internal Analysis of Weka Clusters

From Table 5.7, we can observe that the `associations`, `clusterers`, `j48` and `m5` packages of Weka were completely recovered by our clustering technique. The corresponding clusters are C3, C5, C7 and C4 respectively.

Table 5.7. Mapping Weka clusters to Weka packages provided by system expert

Cluster	**Classes from Packages**
C1	`Estimators, classifiers`
C2	`attributeSelection, filters, classifiers`
C3	`Associations`
C4	`m5`
C5	`Clusterers`
C6	`Classifiers, estimators, filters`
C7	`j48`
C8	`attributeSelection, classifiers, filters`

The other clusters, C1, C2, C6 and C8, map to more than one Weka package. To understand the reasons behind this, we analyzed the static dependencies of the classes residing in these clusters using a static analysis tool called SA4J. The tool has a feature called "explorer" that allowed us to visually explore the relationships between Weka components. We discuss the analysis of each of these clusters in what follows:

Cluster C1 contains the classes of the package `estimators` and one class, `NaiveBayes`, which belongs to the `classifiers` package. The class `NaiveBayes` implements the naive Bayesian classification algorithm which uses various estimators to measure so-called precision values. From the implementation point of view, the class `NaiveBayes` depends tightly on classes of the `estimators` package, which explains why our clustering approach grouped these elements together.

Cluster C2 recovered most classes of the `attribulteSelection` package. However, it also included classes of the `filters` package (namely `AttributeFilter` and `DiscretizeFilter`) and one class named `DecisionTable` of the `classifiers` package. Based on Weka documentation, the classes of the `filters` package included in this cluster serve as utility classes to classes of the `attribulteSelection` package. The class `DecisionTable` does not depend on classes of `attributeSelection` package but rather uses classes of the `filters` package, which explains its inclusion in C2. Ideally, the class `DecisionTable` should have been clustered separately. However, the utility classes of the `filters` packages have caused this class to be misplaced in C2. We believe that this could have be avoided if we had considered `AttributeFilter` and `DiscretizeFilter` as utilities and removed them from the clustering process as we did with the system-level utility package `core`. This would require using utility detection techniques that can detect utilities that are local to part of the system other than system-level utilities.

Each of the clusters C6 and C8 includes classes from three Weka packages as shown in Table 5.7. The static analysis of the classes of C6 and C8 shows that these packages contain classes that are tightly coupled.

Cluster C8 represents mainly two packages `classifiers` and `filters`. The various classification algorithms of Weka use classes of the `filters` package to filter data prior to performing the classification algorithms, which again justifies the grouping of these classes into one cluster, C8.

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.

5.4.3 JHotDraw Clusters

Table 5.8 shows the mapping between the recovered clusters and JHotDraw packages. At first glance, one can attribute the discrepancy between the recovered decomposition and the expert decomposition to the high volume of dependencies between the JHotDraw packages (shown in Figure 5.2). A further in-depth analysis of each cluster using the SA4J tool and the documentation revealed the following:

- Cluster C1 includes only one class from the `figure` package and 12 classes from the package `standard`. The classes of this cluster are centered around one core class `BoxHandleKit`, which dragged into the same cluster a set of utility classes, deal with JHotDraw handles such as `SouthWestHandle`, `WestHandle`, `SouthHandle`, `SouthEastHandle`, `NorthWestHandle`, `NorthHandle`, `NorthEastHandle`, `EastHandle`. The handle classes are responsible for creating the handles for locations of figures on the display.

Table 5.8. Mapping JHotDraw clusters to JHotDraw packages provided by system expert

Cluster	**Classes from Packages**
C1	`standard and figures`
C2	`Figures, standard, javadraw, applet, contrib, framework`
C3	`Figures, standard`
C4	`Figures`
C5	`Framework, standard`
C6	`Figures, standard`
C7	`Javadraw`
C8	`application, standard, figures, javadraw, framework, contrib, net, pert`
C9	`atandard, figures`
C10	`Contrib., figures, standard, javadraw, net, nothing, pert`
C11	`standard, framework`

- C2 groups classes responsible for the editing, alignment, grouping/ungrouping, and bouncing of the figure elements. These classes are responsible for editing figure elements as well as for performing cut, copy, delete and duplicate functions. The alignment functions implement positioning of figure elements (top, left, right, center, bottom or middle). Furthermore also two classes are included to perform grouping and ungrouping of figure elements. The cluster includes one class for start and stop of the animation, which performs bouncing of figure elements on the display. Another class included in this cluster is responsible for figure selection.

- Cluster C3 to C7 contain only two classes each. Cluster C3 includes one class, `PolyLineLocator`, which is responsible for optional line decoration at the start and end of the polygon. The other class, `ChangeAttributeCommand`, provides the command to change a named figure attribute. Cluster C4 has one class (`GroupFigure`) to group figure elements into one object, and another class (`GroupHandle`) that provides a handle to the newly created object. The two classes included in cluster C5 (`FigureSelection` and `FigureTransferCommand`) allow copying a figure element from the display to the clipboard and importing the figure element from the clipboard to the display. Cluster C6 contains two classes (`ChopEllipseConnector` and `ChopBoxConnector`) that collaborate with each other to provide a connection point to an eclipse figure element. Cluster C7 includes two basic classes (`JavaDrawApp`, `Animator`) that start the application. In summary, Clusters C3 to C7 seem to provide specific features supported by JHotDraw.

- Cluster C8 is the largest cluster recovered by our approach and contains many interfaces and base classes of the JHotDraw system. 50% of the classes grouped in the C8 cluster are from the `framework` and `standard` package. Both of these packages support basic operating environment functionality for the JHotDraw drawing tool. Cluster C8 also includes classes responsible for the standard implementation of drawing, buffered update strategy (to draw a view into a buffer followed by copying the buffer to the figure display), as well as the default implementation to support various tools such as: line, polygon. Furthermore it provides the abstract handle for any figure element, figure element enumerator. Figure change listener is also implemented in C8 to track any change to a figure as well as a basic figure interface with its handles to manipulate its shape or attributes, as well as connectors to define how to locate a connection point are provided.

- Cluster C9 contains a core abstract class `AbstractConnector` that provides default implementation for connector interface. Other classes provide handles to various connector activities by implementing `AbstractConnector`. Connectors locate connection points on a figure element. Cluster C9 groups the classes for various connector activities such as elbow connection, line connection, shortest distance connection, etc.

- Cluster 10 contains a core abstract class `AbstractTool` that provides implementation support for various tools, and concrete classes that inherit from `AbstractTool`. Such individual classes provide support for URL tool (To create a hyperlink object in the figure), scribble tool, text tool, border tool, polygon tool and connected text tool. Cluster 10 includes a class `CreationTool` to create a new tool for the application and to create a figure for that new tool. One class which seems to be wrongly placed in cluster C10 is the `PasteCommand` class that should be placed in cluster C2.

The classes of the system that were not clustered in any of the 10 core clusters are placed in the orphan container cluster. These classes, moved to the orphan container cluster, were `GridConstrainer`, `PointConstrainer`, `LineDecoration`, `HJDError` and `TextHolder`. The static analysis of these classes using SA4J revealed that `HJDError` extends the java Error class and does not have a static relation with the other classes. The class `GridConstrainer` implements the `PointConstrainer` interface to provide a constraint for creating a point on the grid. `PointConstrainer` is an interface to create different grids. `TextHolder` is also an interface for editable text contents, which is only used by `TextTool`.

From the above analysis of the JHotDraw decomposition we can conclude that the resulting clusters contain classes that are inter-connected based on the responsibilities they share rather than the type of functionality they provide. Cluster C1 for example creates handles for locating figures, cluster C2 provides various commands for editing and placing figure elements, Cluster C3 to C7 provides supports for various small features, cluster C8 provides core operating environment functionality for the tool, cluster C9 supports the connector activity and cluster C10 provides support capabilities.

5.4.4 Conclusion of Analysis

It should be noted that it is very common for a clustering approach to impose a structural clustering rather than finding "natural" clusters [Tzerpos 00]. The one imposed by the feature-based clustering approach can be viewed as a behavioural decomposition of the system. This view can be used with a combination of other views (e.g., a structural decomposition) to support the comprehension process of large systems and their architectures.

We were also able to observe for both systems that the skeleton decompositions provided stable clusters, which allowed the static analysis phase to incorporate most of the remaining components (i.e., "orphans") into the skeleton clusters.

Chapter 6 Conclusions

Understanding a large software system can be greatly facilitated if the system is decomposed into smaller, more manageable clusters. Unlike existing clustering that focus mainly on static analysis of the source code, in this thesis, we introduced a new clustering technique that combines dynamic analysis and static dependencies. Our main objective is to group a system's entities according to the way they collaborate to implement software features of the system rather than relying only on mere structural relationships found in the source code.

The remaining of this chapter is organized as follows. In Section 6.1, we revisit the main contribution of the thesis. In Section 6.2, we discuss future directions that can improve the work presented in this thesis. We present our closing remarks in Section 6.3.

6.1 Research Contributions

In this research we introduced a novel clustering approach that uses software features as clustering criteria to cluster the entities of a poorly documented software 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, (2) they are abstract concepts and, as such, they can readily be used to extract abstract components of the system.

We provided two case studies, Weka and JhotDraw, to evaluate our approach. The results of both case studies are satisfactory. The approach is successful to construct decomposition where the components that interact with each other are grouped into clusters.

6.2 Future Directions

The approach is evaluated and gives satisfactory results on medium scale software system. However there is a need 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 threshold for calculating the number of features needed for better results and how this number correlates with code coverage.

We also need to compare our technique with existing techniques in order to evaluate its effectiveness and precision.

6.3 Closing Remarks

While there are many clustering techniques already presented in the literature, we identified a unique clustering technique which effectively uses the software features to identify the relation between components. Our contribution to the software clustering domain is to identify the benefit of using dynamic analysis in general and software features in particular as potential clustering criteria to perform software clustering. We hope that further research in the area of software clustering can benefit from the research provided in this thesis.

References

Agrawal 94 — Agrawal, R. and Srikant, R., "Fast Algorithms for Mining Association Rules in Large Databases", In *Proc. of the 20th International Conference on Very Large Data Bases*, pp. 487-499, 1994.

American Heritage 85 — American Heritage, The American Heritage Dictionary. Houghton Mifflin, Boston, MA, 1985.

Anderberg 73 — Anderberg, M.R., Cluster Analysis for Applications. Academic Press, 1973.

Anquetil 99a — Anquetil, N., Fourrier, C., and Lethbridge, T. C., "Experiments with Clustering as a Software Remodularization Method", In *Proc. of the 6th Working Conference on Reverse Engineering,* IEEE Computer Society, pp. 235-255, 1999.

Anquetil 99b — Anquetil, N. and Lethbridge, T. C., "Recovering software architecture from the names of source files", *Journal of Software*

Maintenance and Evolution: Research and Practice, 11(3), pp. 201-221, 1999.

Anquetil 03 Anquetil, N. and Lethbridge, T. C., "Comparative study of Clustering Algorithms and Abstract Representations for Software Remodularisation", *IEE Proceedings on Software,* 150(3), pp. 185-201, 2003.

Antoniol 06 Antoniol, G. and Guéhéneuc, Y., "Feature Identification: An Epidemiological Metaphor", *Transactions on Software Engineering,* 32(9), pp. 627-641, 2006.

Bauer 04 Bauer, M. and Trifu, M., "Architecture-Aware Adaptive Clustering of OO Systems", In *Proc. of the 8th European Working Conference on Software Maintenance and Reengineering,* IEEE Computer Society, pp. 03-14, 2001.

Calder 03 Calder, M., Kolberg, M., Magill, E., and Reiff-Marganiec, S., "Feature interaction: a critical review and considered forecast", *The International Journal of Computer and Telecommunications Networking,* 41(1), pp. 115-141, 2003.

Davis 82 Davis, A. M., "The Design of a Family of Application-Oriented

Requirements Languages", *IEEE Computer,* 15(5), pp. 21-28, 1982.

Dempster 77 Dempster, A., Laird, N. and Rubin, D., "Maximum likelihood from incomplete data via the EM algorithm", *Journal of the Royal Statistical Society*, 39(1), pp. 1–38, 1977.

Dugerdil 07 Dugerdil, P., "Using trace sampling techniques to identify dynamic clusters of classes", In *Proc. of the 2007 Conference of the Center For Advanced Studies on Collaborative Research*, ACM, pp. 306-314, 2007.

Eisenbarth 03 Eisenbarth, T., Koschke, R., and Simon, D., "Locating Features in Source Code", *IEEE Transactions on. Software Engineering,* 29(3), pp. 210-224, 2003.

Eisenberg 05 Eisenberg, A. D. and De Volder, K., "Dynamic Feature Traces: Finding Features in Unfamiliar Code", In *Proc. of the 21st IEEE international Conference on Software Maintenance,* IEEE Computer Society, pp. 337-346, 2005.

Fisher 87 Fisher, D. H., "*Knowledge Acquisition Via Incremental Conceptual Clustering*", PhD dissertation, University of California, Irvine, USA, 1987.

Griss 98 — Griss, M. L., Favaro, J., and Alessandro, M. D., "Integrating Feature Modeling with the RSEB", In *Proc. of the 5th international Conference on Software Reuse*, IEEE Computer Society, pp. 76-85, 1998.

Hamou-Lhadj 06 — Hamou-Lhadj, A. and Lethbridge, T., "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", In *Proc. of the 14th International Conference on Program Comprehension,* IEEE Computer Society, pp. 181-190, 2006.

Holte 93 — Holte, R.C., Very simple classification rules perform well on most commonly used datasets. *Machine Learning,* 11(1), pp. 63-90, 1993.

Hutchens 85 — Hutchens, D. H. and Basili, V. R., "System Structure Analysis: Clustering with Data Bindings", *IEEE Transactions on Software Engineering, 11(8)*, pp. 749-757, 1985.

Jacobson 94 — Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1994.

Jain 88 — Jain, A. K. and Dubes, R. C. Algorithms for Clustering Data. Prentice-Hall, 1988.

John 95 George H. John and Pat Langley. "Estimating Continuous Distributions in Bayesian Classifiers". In *Proc. of the Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann, pp. 338-345, 1995.

Kang 90 Kang, K., Cohen, S., Hess, J., Novak, W. and Peterson, A., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report: CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, Carnegie Mellon University, 1990.

Kang 98 Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M., "FORM: A feature-oriented reuse method with domain-specific reference architectures", *Annals of Software Engineering*, 5, pp. 143-168, 1998.

Lee 97 Lee, H. B. and Zorn, B. G., "BIT: A tool for instrumenting java bytecodes", In *Proc. of the USENIX Symposium on internet Technologies*, pp. 73-82, 1997.

Lee 02 Lee, K., Kang, K. C., and Lee, J., "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering", In *Proc. of the 7th international Conference on Software Reuse: Methods, Techniques, and Tools*, Lecture Notes In Computer Science, vol.

2319, Springer-Verlag, pp. 62-77, 2002.

Liu 03 — Liu, D. and Mei, H., "Mapping requirements to software architecture by feature-orientation", In *Proc. of the 2nd International Workshop on From Software Requirements to Architectures*, pp. 69-76, 2004.

Mancoridis 99 — Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R., "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures", In *Proc. of the International Conference on Software Maintenance,* IEEE Computer Society, pp. 50-59, 1999.

Merlo 93 — Merlo, E., McAdam, I., Mori, R. D., "Source Code Informal Information Analysis Using Connectionist Models", In *Proc. of the 13th International Joint Conference on Artificial Intelligence*, pp. 1339-1344, 1993.

Müller 93 — Müller H. A., Orgun M. A., Tilley S. R and Uhl J. S., "A reverse engineering approach to subsystem structure identification", *Journal of Software Maintenance and Evolution: Research and Practice, 5(4),* pp. 181 – 204, 1993.

Patel 07 Patel, C., Hamou-Lhadj, A. and Rilling, J., "Software clustering based on behavioural features", In *Proc. of the 11th IASTED International Conference on Software Engineering*, Acta Press, pp. 591-185, 2007.

Poshyvanyk 07 Poshyvanyk, D., Guéhéneuc, Y., Marcus, A., Antoniol, G., Rajlich, V., "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", *IEEE Trans. Software Eng, 33(6)*, pp. 420-432, 2007.

Platt 99 Platt, J. C., "Fast training of support vector machines using sequential minimal optimization", In *Advances in Kernel Methods: Support Vector Learning*, MIT Press, pp. 185-208, 1999.

Quinlan 93 Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

Rohatgi 08 Rohatgi, A., Hamou-Lhadj, A. and Rilling, J., "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis", In *Proc. of the International Conference on Program Comprehension*, IEEE Computer Society, pp. 236-241, 2008.

Schwanke 91 Schwanke, R. W., "An intelligent tool for re-engineering software modularity", In *Proc. of the 13th International Conference on Software Engineering,* IEEE Computer Society, pp. 83-92, 1991.

Tzerpos 98 — Tzerpos, V. and Holt, R. C., "Software Botryology, Automatic Clustering of Software Systems", In *Proc. of the 9th International Workshop on Database and Expert Systems Applications,* pp. 811-818, 1998.

Tzerpos 99 — Tzerpos, V. and Holt, R. C., "MoJo: A Distance Metric for Software Clustering", In *Proc. of the 6th Working Conference on Reverse Engineering*, pp. 187-193, 1999.

Tzerpos 00 — Tzerpos, V. and Holt, R. C., "ACDC: An Algorithm for Comprehension-Driven Clustering", In *Proc. of the Seventh Working Conference on Reverse Engineering*, pp. 258-267, 2000.

Wang 97 — Wang, Y. and Witten, I. H., "Induction of model trees for predicting continuous classes", In *Proc. of European Conference on Machine Learning*, pp. 128-137, 1997.

Wen 04 — Wen, Z. and Tzerpos, V., "An Effectiveness Measure for Software Clustering Algorithms", In *Proc. of the 12th IEEE international Workshop on Program Comprehension*, IEEE Computer Society, pp. 194-203, 2004.

Wen 05 Wen, Z. and Tzerpos, V., "Software Clustering based on Omnipresent Object Detection", In *Proc. of the 13th international Workshop on Program Comprehension,* pp. 269-278, 2005.

Wiggerts 97 Wiggerts, T. A., "Using Clustering Algorithms in Legacy Systems Remodularization", In *Proc. of the 4th Working Conference on Reverse Engineering,* pp. 33-43, 1997.

Wilde 95 Wilde, N. and Scully, M. C., "Software reconnaissance: mapping program features to code", *Journal of Software Maintenance and Evolution: Research and Practice,* 7(1), pp. 49-62, 1995

Witten 05 Witten, I. H. and Frank, E., Data Mining: Practical Machine Learning Tools and Techniques (Second Edition). Morgan Kaufmann, 2005.

Xiao 05 Xiao, C. and Tzerpos, V., "Software Clustering Based on Dynamic Dependencies", In *Proc. of the 9th European Conference on Software Maintenance and Reengineering,* pp. 124-133, 2005.

Zave 99 Zave, P., "FAQ Sheet on Feature Interaction", AT&T Labs-Research, 1999. URL:

http://www.research.att.com/~pamela/faq.html, 1999.

Appendix A

In this appendix, we show the complete decompositions resulting from applying our clustering approach to Weka and JHotDraw systems.

Table A.1. Weka complete decomposition

Cluster	Classes
C1	weka.estimators.DiscreteEstimator weka.classifiers.NaiveBayes weka.estimators.NormalEstimator weka.estimators.DDConditionalEstimator weka.estimators.DKConditionalEstimator weka.estimators.DNConditionalEstimator weka.estimators.NDConditionalEstimator weka.estimators.ConditionalEstimator weka.estimators.Estimator weka.estimators.MahalanobisEstimator weka.estimators.NNConditionalEstimator weka.estimators.PoissonEstimator
C2	weka.filters.AttributeFilter weka.classifiers.DecisionTable weka.filters.DiscretizeFilter weka.attributeSelection.CfsSubsetEval weka.attributeSelection.GainRatioAttributeEval weka.attributeSelection.InfoGainAttributeEval weka.attributeSelection.SymmetricalUncertAttributeEval weka.attributeSelection.ASEvaluation weka.attributeSelection.AttributeEvaluator

	weka.attributeSelection.AttributeSelection weka.attributeSelection.UnsupervisedSubsetEvaluator weka.attributeSelection.UnsupervisedAttributeEvaluator weka.attributeSelection.RankedOutputSearch weka.attributeSelection.SubsetEvaluator weka.attributeSelection.BestFirst weka.attributeSelection.ForwardSelection weka.attributeSelection.Ranker weka.attributeSelection.ReliefFAttributeEval
C3	weka.associations.Apriori weka.associations.ItemSet
C4	weka.classifiers.m5.M5Prime weka.classifiers.m5.Options 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 weka.classifiers.m5.Measures
C5	weka.clusterers.EM weka.clusterers.DistributionClusterer weka.clusterers.Clusterer weka.clusterers.ClusterEvaluation weka.clusterers.Cobweb
C6	weka.classifiers.LinearRegression

	weka.filters.NominalToBinaryFilter weka.filters.ReplaceMissingValuesFilter weka.estimators.KernelEstimator weka.estimators.KDConditionalEstimator weka.estimators.KKConditionalEstimator
C7	weka.classifiers.j48.J48 weka.classifiers.j48.C45ModelSelection 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.InfoGainSplitCrit weka.classifiers.j48.SplitCriterion weka.classifiers.j48.GainRatioSplitCrit weka.classifiers.j48.Stats weka.classifiers.j48.BinC45ModelSelection weka.classifiers.j48.BinC45Split weka.classifiers.j48.C45PruneableDecList weka.classifiers.j48.ClassifierDecList weka.classifiers.j48.EntropySplitCrit weka.classifiers.j48.MakeDecList weka.classifiers.j48.PruneableDecList weka.classifiers.j48.PART weka.classifiers.j48.PruneableClassifierTree
C8	weka.classifiers.DistributionClassifier weka.classifiers.Classifier

	weka.classifiers.Evaluation weka.filters.Filter weka.classifiers.SMO weka.filters.NormalizationFilter weka.attributeSelection.OneRAttributeEval weka.attributeSelection.WrapperSubsetEval weka.classifiers.AdaBoostM1 weka.classifiers.ZeroR weka.classifiers.Bagging weka.classifiers.BVDecompose weka.classifiers.CheckClassifier weka.classifiers.ClassificationViaRegression weka.filters.MakeIndicatorFilter weka.classifiers.CVParameterSelection weka.classifiers.DecisionStump weka.classifiers.IB1 weka.classifiers.IBk weka.classifiers.Id3 weka.classifiers.KernelDensity weka.classifiers.Logistic weka.classifiers.LogitBoost weka.classifiers.LWR weka.classifiers.MultiClassClassifier weka.classifiers.MultiScheme weka.classifiers.NaiveBayesSimple weka.classifiers.OneR weka.classifiers.Prism weka.classifiers.RegressionByDiscretization weka.classifiers.Stacking weka.classifiers.VotedPerceptron weka.filters.AddFilter

	weka.filters.AllFilter weka.filters.AttributeSelectionFilter weka.filters.FirstOrderFilter weka.filters.InstanceFilter weka.filters.MergeTwoValuesFilter weka.filters.NullFilter weka.filters.NumericTransformFilter weka.filters.SplitDatasetFilter weka.filters.SwapAttributeValuesFilter weka.attributeSelection.ASSearch weka.classifiers.UpdateableClassifier

Table A.2. JHotDraw complete decomposition

Cluster	Classes
C1	CH.ifa.draw.standard.SouthWestHandle CH.ifa.draw.standard.WestHandle CH.ifa.draw.standard.SouthHandle CH.ifa.draw.standard.SouthEastHandle CH.ifa.draw.standard.NorthWestHandle CH.ifa.draw.standard.NorthHandle CH.ifa.draw.standard.NorthEastHandle CH.ifa.draw.standard.EastHandle CH.ifa.draw.standard.BoxHandleKit CH.ifa.draw.standard.AbstractLocator CH.ifa.draw.standard.ReverseFigureEnumerator CH.ifa.draw.figures.ElbowTextLocator

	CH.ifa.draw.standard.OffsetLocator
C2	CH.ifa.draw.figures.GroupCommand CH.ifa.draw.standard.CutCommand CH.ifa.draw.standard.DeleteCommand CH.ifa.draw.standard.DuplicateCommand CH.ifa.draw.figures.UngroupCommand CH.ifa.draw.standard.SendToBackCommand CH.ifa.draw.standard.CopyCommand CH.ifa.draw.standard.BringToFrontCommand CH.ifa.draw.standard.AlignCommand CH.ifa.draw.samples.javadraw.BouncingDrawing CH.ifa.draw.standard.FigureChangeEventMulticaster CH.ifa.draw.figures.RectangleFigure CH.ifa.draw.standard.ToolButton CH.ifa.draw.applet.DrawApplet CH.ifa.draw.contrib.DiamondFigure CH.ifa.draw.framework.Tool CH.ifa.draw.standard.SimpleUpdateStrategy CH.ifa.draw.applet.SleeperThread CH.ifa.draw.framework.Painter
C3	CH.ifa.draw.figures.PolyLineLocator CH.ifa.draw.standard.ChangeAttributeCommand
C4	CH.ifa.draw.figures.GroupFigure CH.ifa.draw.figures.GroupHandle

C5	CH.ifa.draw.framework.FigureSelection CH.ifa.draw.standard.FigureTransferCommand
C6	CH.ifa.draw.figures.ChopEllipseConnector CH.ifa.draw.standard.ChopBoxConnector
C7	CH.ifa.draw.samples.javadraw.Animator CH.ifa.draw.samples.javadraw.JavaDrawApp
C8	CH.ifa.draw.application.DrawApplication CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.standard.FigureEnumerator CH.ifa.draw.standard.CompositeFigure CH.ifa.draw.standard.BufferedUpdateStrategy CH.ifa.draw.standard.AbstractTool CH.ifa.draw.figures.AttributeFigure CH.ifa.draw.figures.FigureAttributes CH.ifa.draw.samples.javadraw.AnimationDecorator CH.ifa.draw.standard.DecoratorFigure CH.ifa.draw.figures.TextFigure CH.ifa.draw.framework.DrawingChangeEvent CH.ifa.draw.framework.FigureChangeEvent CH.ifa.draw.standard.AbstractFigure CH.ifa.draw.standard.StandardDrawing CH.ifa.draw.figures.ArrowTip CH.ifa.draw.figures.PolyLineFigure CH.ifa.draw.figures.EllipseFigure

	CH.ifa.draw.figures.RoundRectangleFigure CH.ifa.draw.contrib.PolygonFigure CH.ifa.draw.figures.BorderDecorator CH.ifa.draw.standard.AbstractHandle CH.ifa.draw.standard.LocatorHandle CH.ifa.draw.standard.RelativeLocator CH.ifa.draw.figures.RadiusHandle CH.ifa.draw.contrib.ChopPolygonConnector CH.ifa.draw.contrib.PolygonHandle CH.ifa.draw.contrib.PolygonScaleHandle CH.ifa.draw.contrib.TriangleFigure CH.ifa.draw.contrib.TriangleRotationHandle CH.ifa.draw.figures.ImageFigure CH.ifa.draw.figures.NumberTextFigure CH.ifa.draw.figures.PolyLineConnector CH.ifa.draw.figures.PolyLineHandle CH.ifa.draw.samples.javadraw.FollowURLTool CH.ifa.draw.samples.javadraw.JavaDrawViewer CH.ifa.draw.samples.net.NodeFigure CH.ifa.draw.standard.ConnectionHandle CH.ifa.draw.samples.pert.PertDependency CH.ifa.draw.samples.pert.PertFigure CH.ifa.draw.standard.DragTracker CH.ifa.draw.standard.HandleTracker

	CH.ifa.draw.standard.SelectAreaTracker CH.ifa.draw.framework.DrawingEditor CH.ifa.draw.framework.Drawing CH.ifa.draw.framework.FigureEnumeration CH.ifa.draw.framework.Figure CH.ifa.draw.framework.Locator CH.ifa.draw.framework.FigureChangeListener CH.ifa.draw.framework.ConnectionFigure CH.ifa.draw.framework.DrawingView CH.ifa.draw.framework.DrawingChangeListener CH.ifa.draw.framework.Connector CH.ifa.draw.framework.Handle CH.ifa.draw.samples.javadraw.PatternPainter
C9	CH.ifa.draw.figures.ElbowConnection CH.ifa.draw.figures.LineConnection CH.ifa.draw.figures.ShortestDistanceConnector CH.ifa.draw.standard.AbstractConnector CH.ifa.draw.figures.ElbowHandle CH.ifa.draw.standard.ChangeConnectionStartHandle CH.ifa.draw.standard.ChangeConnectionHandle CH.ifa.draw.standard.ChangeConnectionEndHandle CH.ifa.draw.figures.FontSizeHandle CH.ifa.draw.standard.NullHandle CH.ifa.draw.standard.LocatorConnector

C10	CH.ifa.draw.contrib.PolygonTool CH.ifa.draw.figures.BorderTool CH.ifa.draw.standard.ActionTool CH.ifa.draw.figures.InsertImageCommand CH.ifa.draw.figures.ScribbleTool CH.ifa.draw.samples.javadraw.URLTool CH.ifa.draw.standard.ToggleGridCommand CH.ifa.draw.standard.PasteCommand CH.ifa.draw.standard.ConnectionTool CH.ifa.draw.figures.LineFigure CH.ifa.draw.figures.ConnectedTextTool CH.ifa.draw.figures.TextTool CH.ifa.draw.standard.CreationTool CH.ifa.draw.samples.javadraw.MySelectionTool CH.ifa.draw.standard.SelectionTool CH.ifa.draw.samples.javadraw.JavaDrawApplet CH.ifa.draw.samples.net.NetApp CH.ifa.draw.samples.nothing.NothingApp CH.ifa.draw.samples.nothing.NothingApplet CH.ifa.draw.samples.pert.PertApplet CH.ifa.draw.samples.pert.PertFigureCreationTool CH.ifa.draw.samples.pert.PertApplication
C11 (Orphan container)	CH.ifa.draw.standard.GridConstrainer CH.ifa.draw.framework.PointConstrainer

	CH.ifa.draw.figures.LineDecoration CH.ifa.draw.framework.HJDError CH.ifa.draw.standard.TextHolder