Software Botryology
Automatic Clustering of Software Systems

Vassilios Tzerpos
University of Toronto
Toronto, Ontario, CANADA
vtzer@cs.toronto.edu

R. C. Holt
University of Waterloo
Waterloo, Ontario, CANADA
holt@plg.uwaterloo.ca

Abstract

It has long been recognized that the decomposition of a large software system into "meaningful" subsystems is essential for both the development and maintenance phases of a software project. We introduce the term Software Botryology[1] *for the area of research that attempts to automatically cluster a software system.*

In this paper, we survey approaches to the clustering problem from researchers in the software engineering community. We also present clustering techniques used in other disciplines, and argue that their utilization in a software context could lead to better solutions to the software clustering problem. Finally, we outline research challenges and open problems of interest.

1 Introduction

The definition of the term "large software" is constantly changing, as the size of software systems continues to increase rapidly. Advances in hardware technology concerning the size, speed, and cost of primary and secondary storage, as well as the advent of modern programming languages and object-oriented programming, have allowed the size of software systems to increase significantly in the last decade.

However, when a system becomes so large, it is very hard to ensure that its structure is the intended one. Moreover, the original documentation, if it exists at all, becomes outdated as the system evolves, since the developers are usually busy trying to meet the next deadline. The fact that developers often discontinue their association with such large projects intensifies the problem, since they take a lot of the knowledge about the system with them.

These factors contribute to the transformation of a piece of software into what is known as "legacy code", namely a piece of code that one uses but does not necessarily understand. The drawbacks of having legacy code in a software system become obvious when the time comes to alter its functionality, to adapt it to a new hardware platform or operating system, or to improve its performance. One needs to understand the code all over again.

Even systems that are still under development are impacted by these problems. Parts of the system might become legacy code, if only because they have not been maintained for some time. Also, large projects often hire new people who must be brought up to speed, but the seasoned developers are usually too busy to help with this. If the documentation is obsolete, then a newcomer is at a loss as to where to start from, and cannot know the full impact of a potential modification on the rest of the system.

Clearly, a solution to these problems is required. If one could derive a decomposition of a large software system into meaningful subsystems in an automatic (or semi-automatic) way, then much of the effort required to understand and to improve a software system would be alleviated.

Automatic clustering techniques described in the literature claim that they can do exactly this — detect the "natural" groups (or *clusters*) in a collection of entities, such as procedures or source files. However, none of these algorithms has been shown to be effective on large systems. Further research is required in order to reveal the best approach to the problem of decomposing a software system.

Since research on clustering began long before the term "software" was even coined, many techniques are already in use in other disciplines. In this paper, we argue that these techniques (called "classic" cluster-

[1] Botryology: a new name for the discipline of cluster analysis proposed by I.J. Good in 1977 in his article "The botryology of botryology". *Botrys* is the ancient Greek word for a bunch (cluster) of grapes.

ing techniques) could be used effectively in a software context, after they have been adapted to fit the peculiarities of this specific problem domain.

The structure of the rest of this paper is as follows. Section 2 is a survey of current approaches to the software clustering problem from researchers in the software community. Section 3 presents clustering techniques from other disciplines. In section 4, we explain why we think "classic" clustering techniques would be appropriate for the software version of the problem. Section 5 outlines research challenges and open problems of interest. Finally, section 6 concludes the paper.

2 Previous Work

A common approach to the problem of understanding a software system and recovering its design is the *knowledge-based* approach. In the *bottom-up* version of this approach, one attempts to reverse-engineer small fragments of the source code and to understand them, using pre-existing domain knowledge. One then combines these fragments together in an effort to understand the system as a whole, thus determining its design. This approach has been shown to work well with small systems.

When dealing with large systems, however, this approach does not perform as effectively. The size of the knowledge base becoming prohibitively large is one of the reasons. Other reasons are outlined by Neighbors [16]: "Knowledge-based understanding of large system semantics [is] currently too difficult for three reasons: absence of a robust semantic theory, lack of problem domain specific semantics, and knowledge spreading in the source code".

For these reasons, the software clustering community mainly adopts *structural-based* approaches. In these approaches, the decomposition of a software system is determined by looking at syntactic interactions (such as "call" or "fetch") between entities (such as procedures or variables). In this case, the problem of clustering a software system can be thought of as the partitioning of the vertex set of a graph, where the nodes are defined as procedures or variables, and the edges as relations between these entities.

The rest of this section presents a survey of important publications in the field of software clustering, as well as some of the most recent approaches to the problem.

2.1 Key publications

In one of the early works in software clustering, L. A. Belady and C. J. Evangelisti [4] recognized the need to automatically cluster a software system in order to reduce its complexity. They also presented a first approach to doing this for a specific system. In addition, they provided a measure for the complexity[2] of a system after it has been clustered. Their approach, however, only works with a specific kind of system, and they did not validate their complexity measure in any way. A point of interest is that they didn't extract information from the source code, but rather from the system's documentation.

Subsequent to this work, Hutchens and Basili [11] performed clustering based on *data bindings*. A data binding was defined as an interaction between two procedures based on the location of variables that are within the static scope of both procedures. They defined different kinds of data bindings; from simplistic and easy to compute to sophisticated and hard to compute. On the basis of the data bindings, a hierarchy is constructed from which a partition could be derived.

An interesting feature of their paper is that they compared their structures with the developer's mental model with satisfactory results. They also raised the important issue of *stability*; when the system changes slightly, how is the clustering affected? Finally, they recognized that it might be necessary to disregard certain information in order to get a clearer view of the structure of a software system.

One of the most active researchers in the area of software clustering in the early 1990s was Robert W. Schwanke. His papers [19, 20, 18] and his tool (called ARCH) addressed the problem of automatic clustering in an innovative way. Although his approaches were never tested against a large software system, they showed considerable promise. One of his main contributions was that he added to the "classic" low-coupling and high-cohesion heuristics by introducing the "shared neighbors" technique [20] in order to capture patterns that appear commonly in software systems. Also, his "maverick analysis" [18] enabled him to refine a partition by identifying components that happened to belong to the wrong subsystem, and placing them in the correct one.

Choi and Scacchi [6] presented an approach to finding subsystem hierarchies based on resource exchanges between modules. The complexity of their algorithm is $O(n^2)$, which is better than Schwanke's $O(n^3)$, but still probably too high for large systems. It appears to perform well on small examples, but its ability to scale up is questionable.

Hausi Müller has also been involved in the automatic clustering problem [15, 14]. His approaches tend to be

[2] Complexity here refers to how difficult it is to understand a system after it has been clustered in a specific way.

semi-automatic, meaning that they are meant to help a designer perform clustering on a software system. He introduces the important principles of *small interfaces* (the number of elements of a subsystem that interface with other subsystems should be small compared to the total number of elements in the subsystem) and of *few interfaces* (a given subsystem should interface only with a small number of the other subsystems).

2.2 Recent work

The last couple of years have seen a renewed interest in the problem of clustering a software system automatically. The main reason for this is the rapid growth of Reverse Engineering as a research field, largely due to the Year 2000 and Euro conversion[3] problems. Understanding large software has become a very important issue and clustering can help deal with it.

James M. Neighbors attempted to identify subsystems with the ultimate goal of hand extraction of reusable components [16]. He looked at compile-time and link-time interconnections between components and tried different approaches. The approaches that were successful were based on naming and on reference context. His results were validated by the developers of the system on which he experimented.

An interesting alternative approach was presented by Nicolas Anquetil and Timothy Lethbridge [3]. Instead of looking at structural information, such as procedure calls or data references, they only looked at the names of the resources of the system. Their experiments produced promising results, but their approach has the obvious drawback that it relies on the developers' consistency with the naming of their resources.

Finally, various researchers have recently started looking at techniques used in other disciplines in order to come up with a better solution to the automatic clustering problem. Theo Wiggerts [24] presented a survey of techniques used by the cluster analysis community and attempted to reuse them for system remodularization. His future plans included clustering a software system in a "more or less object-oriented" way. Spiros Mancoridis [13] treated clustering as an optimization problem and employed genetic algorithms in order to overcome the local optima problem of "hill-climbing" algorithms, which are commonly used in clustering problems. His experiments demonstrate encouraging results and fast performance.

[3]Financial software in Europe has to be modified in order to accommodate the common currency.

2.3 Observations

By examining the literature on software clustering one can draw interesting observations. First, most researchers seem to agree on structural-based criteria and naming conventions as being the most promising approaches. However, there exists a variety of different interactions between modules that are used as the basis to decide which resources depend on which. Isolating the interactions that are appropriate for the software clustering problem, and determining the properties that make them so, is a problem that needs to be studied.

Another observation is that none of the approaches has been tested against a large software system. This omission becomes more interesting when one considers that these approaches were developed with such systems specifically in mind. It is not clear whether these approaches scale up to large systems.

Also, validation of an approach against more than one system is required. Many researchers present results that demonstrate that their algorithm performs very well for a given system. It would be interesting to see how the algorithm performs on a number of systems, since an algorithm can be specifically tuned to perform well on a particular system.

Finally, the issue of performance is very important. Most graph partitioning problems are shown to be NP-complete [8] or NP-hard. The approaches presented above, however, are heuristic approaches that attempt to reduce this complexity to polynomial upper bounds. What kind of complexity is acceptable for large systems remains to be seen.

A more detailed description of open problems and research challenges can be found in section 5. In the next section, we present approaches employed in the cluster analysis community to solve clustering problems found in other disciplines.

3 Classic Clustering

Cluster analysis has been used in a number of different disciplines[4] in order to solve a wide spectrum of problems. Its objective is to find algorithms and methods for grouping or classifying objects. Many diverse techniques have been developed in order to discover structure within complex bodies of data.

In this section, we will present the most important cluster analysis techniques found in the literature. Since these techniques are used in many disciplines,

[4]Examples include psychology, biology, statistics, social sciences, and various fields of engineering.

there is considerable confusion of terminology. For example, the raw material to be clustered has been called "point", "item", "data unit", "subject", "object", "element", "entity" and many other terms. For this paper, we will use the term "object". Also, the aspects of the objects that we look at in order to decide on the appropriate clustering have been called "variables", "attributes", "characters", or "features". We will use the term "feature".

3.1 Similarity Measures

One of the first things that a clustering approach usually does is to decide on what grounds will two objects be judged to be similar. Moreover, one needs a measure that will decide which pair of objects are "more similar" than any other pair. The answer to this problem is a *similarity measure.*

Similarity measures can be divided in two groups, depending on the kind of information that serves as their input. We distinguish the following kinds of information:

1. *Relations between the objects.* In this case, the problem can be represented as a graph, where the nodes are the objects and the edges are the relations. If we have more than one relation, then the graph will have multiple kinds of edges.

   Common similarity measures that deal with cases like this are based on the number of edges connecting two objects, the length of the shortest path between two objects, or the weight that different kinds of edges might have. Whether the graph is directed or undirected is also a factor.

2. *The score of the objects on different features.* In this case, similarity is usually measured by *association coefficients.* These are expressed in terms of the number of features which are present for each object. For this reason, association coefficients assume binary features (i.e. reflecting whether a feature is either present or not). The following table is used in order to calculate various coefficients between object i and object j:

	Object j **1**	Object j **0**
Object i **1**	a	b
Object i **0**	c	d

   In the above table, a is the number of features that are present for both objects, b the number of features present only for object i, and so on. Different coefficients treat 0-0 matches (their number is given by d) differently and also put different weightings on any of the four entries of the table. The most common coefficients are:

   - the *simple matching coefficient*, defined as: $\frac{a+d}{a+b+c+d}$
   - the *Jaccard coefficient*, defined as: $\frac{a}{a+b+c}$

   An extensive study of coefficients can be found in [2].

Other similarity measures that are found in the literature include distance measures (usually Euclidean or Manhattan), correlation coefficients, and probabilistic measures (based on the assumption that agreement on rare features is more important than agreement on frequent ones).

3.2 Algorithms

Once the similarity measure has been decided upon, an appropriate algorithm has to be chosen as well. The algorithms that are used in order to cluster a number of objects are generally divided into two categories:

1. *Hierarchical algorithms.* These algorithms produce a nested sequence of partitions. In one end of this sequence is the partition where each object is in a different cluster (we will call this partition ALL) and in the other end the partition where all the objects are in the same cluster (we will call this partition ONE). At each step through this sequence two of the clusters are joined together.

   Hierarchical algorithms are divided into two categories:

   (a) *Agglomerative (or bottom-up).* These start with partition ALL and iteratively join the most similar clusters based on the similarity measure. One of the partitions of the obtained sequence is selected as the solution (also known as the "cut-point").

   An interesting point of debate between researchers is how to compute similarity between a newly-formed cluster and the rest of the already formed clusters. This is called the *update rule* problem. Many different solutions exist for it. The most common include the single-link update rule (the similarity of the newly formed cluster to an existing cluster C is the *maximum* of the similarities of its constituents to C), the complete-link update rule (the similarity of the newly formed cluster to an existing cluster C is the *minimum* of the similarities of its constituents to C), and others.

(b) *Divisive (or top-down).* These start with partition ONE and try to iteratively split it until we reach partition ALL. Such algorithms, however, suffer from excessive computational complexity, as one has to look at an exponential number of partitions at every step. This is the main reason why these algorithms are not very popular.

2. *Partitional algorithms.* The way these algorithms usually work is that they start with an initial partition and try to modify it in an attempt to optimize a criterion that represents the quality of a given partition. The challenge that partitional algorithms face is the combinatorial explosion of the number of possible partitions.

   The usual workaround to this problem is to start with an initial partition (chosen randomly or based on some heuristics) and attempt to optimize the chosen criterion by modifying that partition in an appropriate way. These algorithms (called hill-climbing algorithms) do converge [2], but usually to local optima. Therefore, the choice of the initial partition is crucial for the success of the algorithm.

3.3 Observations

By examining the literature on cluster analysis, one can draw some interesting observations. First, researchers agree that "a classification is neither true or false" [7]. This means that no particular partition can be the ideal answer to the problem of classifying a large number of objects. Based on different points of view, one can come up with two different, but equally valid, decompositions of the same set of objects. Some classifications, however, are more useful than others. It is the job of the researcher on cluster analysis to find what factors determine the usefulness of a particular clustering.

Another observation is that "the multitude of alternatives makes it difficult to say that a particular measure and a specific method are clearly superior selections for treating the problem at hand" [2]. On any given problem, a large group of different methods will give practically the same results, while perhaps a few other methods will give distinctively different results. A theoretic explanation of the behaviour of different methods does not exist, however.

Finally, looking back at the literature on software clustering, we see that some cluster analysis techniques have indeed been used by software researchers. Hutchens and Basili [11] did use a hierarchical agglomerative method in order to perform their clustering. Schwanke [20] did define binary features in the same way as defined earlier in this section. Mancoridis [13] did use a hill-climbing optimization approach similar to the one presented in section 3.2.

It remains to be seen whether the software community will adopt more cluster analysis techniques. The next section explains why we think this would be a good idea.

4 Using classic clustering techniques for software clustering

One of the main points of this paper is that the software community should benefit from the cluster analysis techniques available. As explained at the end of the previous section, many software researchers are reinventing several of these techniques. It would be beneficial if the software community could adopt and adapt the already well-studied algorithms of cluster analysis.

One of the main reasons why we think this would be a good idea, is that the peculiarities of software as a clustering domain could be used to alleviate a lot of the problems facing "classic" clustering techniques. For example, with software we already have a rather good idea of what a cluster should look like. Software Engineering principles such as *information hiding* [17], or *few interfaces* could guide the clustering process towards a desirable solution. Also, since our goal is usually to understand a software system, we can cluster to different levels of proximity[5] depending on our perspective. Therefore, specifying the number of clusters is not a big problem for software clustering. Furthermore, a software system can have more than one valid view, which is what different clustering algorithms can give us.

Another interesting issue in the cluster analysis literature is that of randomness. Most clustering algorithms can be "accused" of imposing a structure on a set of data, even if no structure exists. In this case, it is possible that the structure presented as the final solution is an artifact of the algorithm used, rather than a "natural" grouping of the objects in question. With legacy software however, this need not be a problem. As it has been noted [24], any structure is better than no structure, since one needs to start somewhere. Besides, one would hope that even the most badly written piece of code would have some structure.

In their classic text on "Algorithms for Clustering Data" [12], Jain and Dubes present a framework for a

[5] Proximity, in the cluster analysis literature, refers to how close to the data we look, i.e. are we trying to find a few or a lot of clusters.

cluster analysis project, which is divided into 7 steps. To demonstrate that it would also fit the software clustering problem, we present it in a software context (the titles are theirs, but the explanations ours):

1. *Data collection.* This refers to extracting the relevant information from the source code. A critical issue here is what kind of information one needs to extract.

2. *Initial screening.* The data extracted from the source code usually requires some massaging before it can be used. As noted in [11], certain information may have to be deleted as it might interfere with the clustering process, e.g. omnipresent nodes[6] [15].

3. *Representation.* This refers to choosing the appropriate similarity measure. It is usually based on the type of information available, the experience of the investigator, and the insight of system experts. In the case of software clustering, there exists a wealth of different software metrics [1] that could be used for this purpose.

4. *Clustering tendency.* This step checks if the available data have a natural tendency to cluster or not. As explained before, this is not a problem in a software context.

5. *Clustering strategy.* The algorithm to be used and the value of any parameters in it are chosen during this step. It is up to the investigator to decide on the most appropriate algorithm. Comparative studies between different existing algorithms would facilitate the process of choosing or developing effective algorithms.

6. *Validation.* Formal techniques for the validation of a partition exist, but in a software context there is usually a different way. Developers associated with the examined software project can compare the partition obtained from the automatic clustering approach with their own mental model of the structure of a system. Also, in the case of legacy software, empirical studies could evaluate whether the clustering actually helped in the understanding of the system.

7. *Interpretation.* This refers to comparing results with other studies, drawing conclusions, and getting ideas for improvements on any of the previous steps.

[6]This refers to nodes (typically procedures in the software case) with a large in- or out-degree. In the software case, this might correspond to library routines, the interactions with which are not necessary in order to decide on the structure of the rest of the system.

In the next section, we will present open problems and research challenges a researcher in the field of software clustering might have to face.

5 Research Challenges

Throughout the text of this paper, we have mentioned various problems facing researchers in software clustering and presented the reasons that make these problems difficult. In this section, we present an organized list of open problems that pose interesting challenges to the researchers in the area:

- It is not clear which kinds of relations between "software objects" are appropriate from a clustering point of view. Procedure calls and data references are commonly used, but what about relations such as source inclusion [5] or type references? Should they be used, and if so, with equal weight to other relations or not? The field is in need of a "theory of dependencies" that characterizes such relations.

- Another interesting research issue is the selection of appropriate algorithms. A comparative study tested on a number of systems is long overdue. It is possible that certain algorithms are best suited for a particular type of software system. A categorization of algorithms and the types of software for which they work best would be beneficial to the software clustering field.

- There exists a gap between the structures obtained by the software clustering researchers and the ones presented by the software architecture community [9, 21]. Closing this gap is not easy, as a compromise has to be found between the automatic approaches of the clustering community, and the "supervised" ones of the software architecture community, as in [10].

- Clustering approaches need to be tested on large systems, as success on small systems does not guarantee effective scaling up to large systems. Obtaining access to large systems is not easy, but it can be done, and it is crucial for the validation of candidate approaches. The imminent release of the source code for Netscape could mean the creation of a benchmark for different algorithms.

- Most software approaches currently present static views of the structure of a software system. However, most large systems are complex enough to require more elaborate views, such as dynamic views. This is certainly an important challenge for the software clustering community.

- The issue of stability is also important. Minor changes to the software system should not drastically affect its generated structure. A study of the types of structures generated by different algorithms and their stability would be very interesting.

- A related issue is that of incremental clustering. Assuming that a satisfactory partition of a software system exists, how do we update this structure when the software system changes, and how do we do it in a way that still reflects the actual structure of the system and causes the least possible modification? An approach to this problem has been presented by the author in [23].

- Software is a peculiar clustering domain since the developers that are associated with the system being examined can provide a lot of help. Integrating information obtained from the developers with the automatic approaches described in this paper is an important challenge [22].

The aforementioned problems pose interesting challenges to researchers, and suggest that the software clustering field is a fertile one for research.

6 Conclusion

The goal of this paper was threefold:

- To present the state of the art in the research of software clustering

- To survey "classic" clustering techniques and show that they can be utilized in a software context.

- To demonstrate that the software clustering field has research potential.

In section 2, we presented the most important approaches to the software clustering problem, and outlined their advantages and disadvantages. In section 3 we surveyed cluster analysis approaches that have been used in other disciplines, and in section 4, argued that they could be used effectively in a software context. Finally, in section 5 we presented a number of open problems in the area of software clustering.

We believe that further research on the problem of decomposing a software system automatically is very important, and that it will benefit not only the research community, but also the people involved in the development of large software systems.

References

[1] R. Adamov and P. Baumann. *Literature Review on Software Metrics.* Institut für Informatik der Universität Zürich, Oct. 1987.

[2] M. R. Anderberg. *Cluster Analysis for Applications.* Academic Press Inc., 1973.

[3] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *Proceedings of CASCON 1997*, pages 184–195, Nov. 1997.

[4] L. A. Belady and C. J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2:23–29, 1981.

[5] I. H. Carmichael, V. Tzerpos, and R. Holt. Design maintenance : Unexpected architectural interactions. *International Conference on Software Maintenance*, pages 134–137, Oct. 1995.

[6] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[7] B. S. Everitt. *Cluster Analysis.* John Wiley & Sons, 1993.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman and co., 1979.

[9] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993.

[10] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. *International Conference on Software Engineering*, pages 186–195, Apr. 1995.

[11] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, Aug. 1985.

[12] A. Jain and R. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[13] S. Mancoridis, B. Mitchell, et al. Using automatic clustering to produce high-level system organizations of source code. In *submitted to Workshop on Program Comprehension.* IEEE Computer Society Press, 1998.

[14] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, Dec. 1993.

[15] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Conference on Software Maintenance*, pages 12–19, Nov. 1990.

[16] J. M. Neighbors. Finding reusable software components in large systems. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 2–10. IEEE Computer Society Press, Nov. 1996.

[17] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, Dec. 1972.

[18] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[19] R. W. Schwanke, R. Altucher, and M. A. Platoff. Discovering, visualizing, and controlling software structure. In *International Workshop on Software Specification and Design*, pages 147–150. IEEE Computer Society Press, 1989.

[20] R. W. Schwanke and M. A. Platoff. Cross references are features. In *Second International Workshop on Software Configuration Management*, pages 86–95. ACM Press, 1989.

[21] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[22] V. Tzerpos and R. C. Holt. A hybrid process for recovering software architecture. In *CASCON 1996*, pages 1–6, Nov. 1996.

[23] V. Tzerpos and R. C. Holt. The orphan adoption problem in architecture maintenance. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 76–82, Oct. 1997.

[24] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE Computer Society Press, Oct. 1997.