Clustering technique for conceptual clusters

Brice Govin[1,2]
Arnaud Monegier du Sorbier[1]
[1]Thales Air Systems
{firstname.lastname}@thalesgroup.com

Nicolas Anquetil[2] Stéphane Ducasse[2]
[2]Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 - CRIStAL,
F-59000 Lille, France
{firstname.lastname}@inria.fr

Abstract

Clustering aims to classify elements into groups called classes or clusters. Clustering is used in reverse-engineering to help to understand legacy software. It is also a technic used in re-engineering to propose gatherings of software entities to engineers who can then accept them or not. This paper presents a Pharo implementation of an iterative and semi-automatic method for clustering. Our method proposes, to an end-user, clusters that are based on domain information and structural information. The method presented in this paper has been applied in an industrial project of architecture migration. We show that this method helps engineers to cluster software elements into domain concepts. The clustering gives a result of 56% of precision and 79% of recall after the automated part in a high level clustering. A deeper clustering gives a result of 51% of precision and 52% of recall.

Keywords Clustering, iterative approach

1. Introduction

Research has shown that understanding and/or refactoring a large legacy software are fastidious tasks. Since the early stages of reverse-engineering and re-engineering research, scientists have provided solutions to help engineers in such tasks. One of the many existing methods to ease reverse-engineers and re-engineers work is clustering.

Clustering is a method that aims to classify elements according to a set of new elements called classes. It is used in computer science to simplify the representation of a software into a manually manipulable set of classes. Basically, clustering gathers software elements (*e.g.* files or packages or functions) into modules that are relevant for software engineers[1].

Such a gathering enables engineers to get an overview of the existing software and to focus on the understanding of one cohesive set of elements at a time instead of the entire software. It also enables them to understand separately each set of software elements and then to dispatch more easily the tasks to several engineers. However, defining what is "relevant" for a software engineer strongly differs from software to software (depending on the producer of the software or the software team that has developed the software, for example). Moreover, clustering a software according to the libraries it uses or according to the concepts it uses is very different. Clustering's goal is another variation point on what is "relevant" for a software engineers. For instance, when the goal of a project is to delete the coupling to a library, clustering aims to restructure the physical software architecture. When the goal of a project is to gather elements into concepts, clustering aims to restructure the logical software architecture. Clustering techniques are slightly different whether the goal is to restructure the physical or the logical architecture of a software.

This paper presents an iterative and semi-automatic clustering method for clustering software elements at different hierarchical levels according to domain concepts. Our method is based on an extended version of Moose [7] for the Ada programming language. This method has been applied on a large legacy software in an industrial project of architecture migration in order to gather software elements for refactoring. Section 2 presents existing works on clustering techniques. Section 3 presents our clustering method. The next two sections present the application of our method on the industrial project and the results obtained. Section 6 summarises this paper and presents future work.

2. Existing Clustering Techniques

Clustering is a method that aims to classify elements according to a set of new elements called classes. It enables the split of a large set of elements into groups of similar objects in order to facilitate the understanding of this large set [2].

Clustering techniques have been categorised [11] into three main classes according to the kind of information used by the clustering techniques: domain-based model approaches; dataflow-based approaches; structure-based approaches. Structure-based approaches are split into four subcategories: connection-based ; metric-based; graph-based; concept-based. This classification is close to the one presented by Canfora *et al.* [4] except that Canfora *et al.* put domain-based model approaches, dataflow-based approaches and the subcategories of structure-based approaches at the same level.

Koschke and Girard [10] used a connection-based approach with a dominance analysis [5] over a graph between variables and what they called routines (a procedure or a function). Dominance analysis is based on the connections between nodes. This method enables the finding of subprogram clusters that are consistent according to the variables they are using. Clusters encapsulate subprogram with the variables they are using to create components.

Canfora and Cimitile present a clustering approach based on a metric [3]. Their metric uses the number of connections in a sub-graph and the fan-in and fan-out of an element. Their method consists in using this metric over the relationships between variables and routines to gather routines and variables into objects (a collection of software elements). The output of this method is similar to the one of Koschke and Girard but the two methods differs on how they decide the gathering, respectively a metric and a dominance analysis.

Dunn and Knight describe a tool [8] meant to help engineers identify software elements that are candidates for a reuse library. Their tool is based on a graph representation of the source code. They navigate through the whole graph to cluster elements according to a set of rules, that come from the design knowledge of the engineers. Engineers can then assess about the pertinence of the clusters and keep them or not.

Lindig and Snelting proposed an approach [12] where legacy code is modularised according to a mathematical concept analysis. They construct a concept lattice thanks to the relations between procedures and variables and use this lattice to propose clusters (called "modules"). They claim that the concept lattices can be used to have cohesion and coupling between the cluster candidates. Other papers [6, 9, 13] have shown the usefulness of concept-based approaches for clustering.

3. The Iterative Clustering Method

The clustering method proposed in this paper makes the strong hypothesis that the number of clusters and their hierarchy are known by the engineers. This hypothesis comes from the fact that when a reverse-engineering or re-engineering task starts, engineers have a general idea of the solution they want. Given that fact, the proposed method considers domain information from the engineers to compute a clustering at different hierarchical level of software elements. To do so, the proposed method is split into iterations, which instantiates a generic iteration.

Section 3.1 details the generic iteration used in our method. Section 3.2 discusses the hypothesis we made and how the proposed method is placed on the categorisation of clustering approaches[4, 11].

3.1 Generic Iteration of the Proposed Method

An iteration of our method consists in computing a clustering at a certain level. Before starting the clustering, the following points have to be defined:

i. Elements to be clustered (*e.g.* files, packages, functions and variables or types and functions...)

ii. Connections between elements defined previously (*e.g.* imports between packages...)

iii. A function for deciding whether an element is allocated to a certain cluster or not

These points are the main characteristics of a specific iteration and can be viewed as the variations points between the iterations. They lead to run the clustering for an iteration as followed:

Graph Extraction: A graph is extracted from the source code. In this graph, nodes are elements of *i* and edges are the connections of *ii*.

Clusters Kernels Selection: For each cluster, engineers have to define a set of elements of *i* as the kernel of a cluster. Elements of the graph are crystallised around this cluster kernel. This selection remains at the discretion of the engineers and two different selections will lead to two different clusterings. Kernels' selection can be automated if it lies on structural information (*e.g.* a naming convention).

Connections Navigation: From each kernel, the graph extracted is navigated through the connexions defined previously. This step consists in selecting the target of an outgoing connexion.

Allocation Decision: For each element selected by the previous step, a decision is made on whether the element is allocated to the cluster of the kernel from where the navigation has started. The function defined in *iii* is applied to decide the allocation. This function can be based on a metric or on other relevant information such as if the element is named according to a given naming convention. For all the newly allocated elements, the connections navigation step and the allocation decision step are recursively performed until reaching a breakpoint.

Breakpoint: The iteration ends when all elements that can be navigated from the chosen kernel have been navigated. It means that for each kernel, navigated elements have to be memorised.

At the end of an iteration, conflicts can occur in the elements' allocation. Conflicts are of two kinds: a *multi-allocated* element or a *non-allocated* element. A *multi-allocated* element is an element that is allocated to several clusters while a *non-allocated* element is an element that is not allocated to any cluster. Multi-allocation conflict is due to the allocation decision function not considering previous allocation of an element as depicted in figure 1. Non-allocation conflict is due to either the allocation decision function putting aside an element every time for some reason or to the fact that the kernel elements might not be a root of the extracted graph as depicted in figure 1.

In such cases, a conflict resolution is required to finish the clustering of an iteration. This conflict resolution can be automated if a method for conflict resolution has been defined earlier but it is optional. If no method has been defined for conflict resolution, it has to be done manually by the engineers.

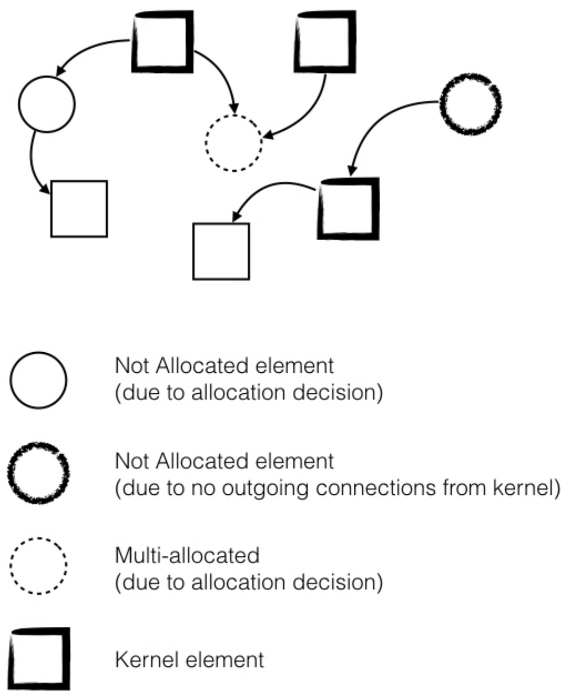
Not Allocated element
(due to allocation decision)
Not Allocated element
(due to no outgoing connections from kernel)
Multi-allocated
(due to allocation decision)
Kernel element

Figure 1. Example of allocation conflicts that may occur (each kernel element is associated to a different cluster)

3.2 Discussion About the Proposed Method

Description of the generic iteration of our method, in section 3.1, highlights points that should be discussed.

First, the required definitions *i*, *ii* and *iii* come from a study on our method and others clustering methods. This study is not a systematic review of all clustering methods and, thus, a systematic review should be done to prove that point. Nonetheless, we believe that at least *i* and *ii* are two required points to define before starting a clustering approach (unless the chosen clustering approach already defines that for the user). Moreover, our method differs from the other methods we have found because we do not fix *i* and *ii*.

Second, the optional definition of a function for conflict resolution differs also from other clustering methods found in literature. Methods in literature do not consider a potential function for conflict resolution because the clustering methods cannot allocate a software element to several clusters. We choose to allow multi-allocation of software elements into clusters because, in a re-engineering task, it can be decided that software elements will be reworked. Moreover, we do not believe that conceptual clusters can be completely found automatically but rather have to be completed and improved by a domain expert. A multi-allocation can then orient the choice of the domain expert and increase their productivity.

4. Application of our Method on an Industrial Project

We applied our method in the scope of an industrial project of architecture migration. This project aims to refactor an entire legacy software system written in Ada 95. Table 1 summarises structural characteristics about the legacy software.

Table 1. Main structural characteristics of the legacy software to which our method is applied (Variables are global variables)

Programming language	# kLoC	#Packages	#Subprograms	#Variables
Ada 95	324	1537	14650	15362

It is important that the reader roughly understands how the Ada programming language is constructed for the rest of this section. The Ada programming language, in its 95 form, is a procedural language, close to C, that is constructed as follows: files contain packages or subprograms; packages contain packages, subprograms, types and/or variables ; subprograms contain packages, subprograms, types and/or variables. Subprograms are functions and procedures. Packages and subprograms are the only Ada entities with a behaviour while types are the skeleton of a data and variables are the data.

In the legacy software presented in table 1, packages are the highest container elements in the containment hierarchy of the software and subprograms are contained within packages. Applying our clustering method on this software aims to cluster subprograms under a hierarchy of clusters. To have such a hierarchy of clusters, we instantiate two times the generic iteration described in section 3.1. The two iterations are described in the next two sections.

4.1 First Instance of the Generic Iteration on the Legacy Software

An instance of a generic iteration requires to define *i*, *ii* and *iii*. For the first iteration these points are defined as follows:

i. Elements to be clustered are *packages*

ii. Connections between elements are *imported packages*

iii. The function of allocation decision is that if the importing package uses the imported package then the imported package is allocated to the cluster of the importing package

The definition of *ii* means that if a package A imports a package B then A has a connection directed to B. The definition of *iii* means that if package B is imported by package A, B is allocated to the same cluster as A only if A accesses a variable declared in B, A references a type declared in B or calls a subprogram declared in B. Concretely, figure 2 gives an example where package A imports package C but does not use any of C's children (variables, types or subprograms). This function of allocation decision enables the selection only packages that are used, such as package B in figure 2 No conflict resolution function is defined for this iteration.

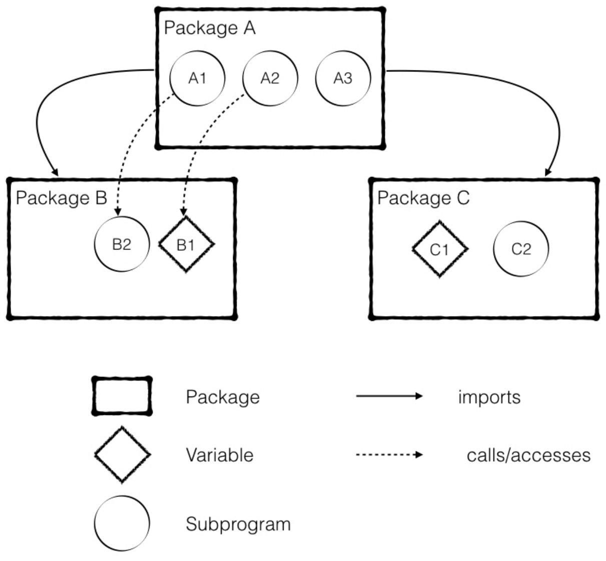

Figure 2. Example of a package imported but not used

After these definitions, the instantiation of the generic iteration is applied to the set of packages of the application. The clusters kernels are selected according to a naming convention in the legacy software. Each package respecting this naming convention is a kernel for one cluster; hence, the packages have to be clustered into 44 clusters. The resulting clustering is compared with a clustering done manually by engineers and represented by a hierarchy of folders. To compare the clustering, Anquetil and Lethbridge define the *intra-pairs* as a bidirectional connection between elements of a same cluster [1]. The *intra-pairs* resulting from the first instance of the generic iteration is compared in terms of precision and recall with the *intra-pairs* of the manual clustering. The results are summarised and commented in section 5.

4.2 Second Instance of the Generic Iteration on the Legacy Software

This iteration takes place in the context of one cluster resulting from the first iteration. The goal of this second iteration is to cluster a special subset of subprograms of the application. Then, as for the first instance, required points of section 3.1 have to be defined for this iteration.

i. Elements to be clustered are *subprograms contained in packages that are allocated to a given cluster*

ii. Connections between elements are *calls between subprograms*

iii. The function of allocation decision is that if the number of common used variables or types between two subprograms are greater than a given threshold then the two subprograms are allocated to the same cluster

The threshold used in *iii* is defined subjectively, in our case we used a threshold of 2. No conflict resolution function is defined for this iteration.

Then this instance of the generic iteration is applied to the set of subprograms of *i*. Clusters kernels are selected amongst the roots of the dependency graph and according to a naming convention. The resulting clustering is compared with a clustering done manually by engineers. This manual clustering is expressed in a file as a configuration. The comparison is done as described in section 4.1in terms of precision and recall; and results are summarised in section 5.

5. Results of the Application of our Method

We have summarised the results of the two instances of the generic iteration in Table 2.

Table 2. Results of our method on an industrial project

	Precision	Recall
1st instance of generic iteration	56%	79%
2nd instance of generic iteration	51%	52%

To avoid error combination, the second instance of the generic iteration is performed from a cluster given by the oracle. It means that if the second instance is performed on a wrong cluster resulting from the first instance, precision and recall can decrease.

We conducted a survey amongst the engineers and architects of the industrial project in order to know what could be the satisfaction threshold for the precision and recall. Engineers and architects consider that a precision and recall greater than 50% is enough for this experiment. They stated that their results are not the absolute truth and they consider that a cluster different from theirs can be good as well. Nonetheless, as explained in section 4.1 and 4.2, we consider their results as our oracle.

Recall between first and second instance of the generic iteration goes from 76% to 52%. This decrease can be explained by the fact that the clustering oracle results from a semantic analysis of the functions. On the contrary, the automated clustering results from a structural analysis of the functions (function calls and variable uses). Although the value of the recall is greater than the given threshold of 50%, the decrease can also be explained by the definition of the allocation decision function. A function of allocation decision that is better defined could increase the recall of the second instance of the generic iteration.

6. Conclusions and Future Work

Clustering is a technique that enables the gathering of elements into new ones. This technique is used in everyday life to understand and find information more easily. It is also mainly used in reverse engineering of software application to get a summarised view of an application. However, automatic clustering into semantical clusters is still a challenge.

We proposed in this paper a clustering approach for software application that is semi-automatic and that gives semantical clusters. Our approach is iterative and is based on a generic iteration that can be adapted to deeply cluster software elements. Adaptation of the generic iteration is based on *i*, *ii* and *iii*: definition of the elements to cluster, how to navigate and how to allocate the elements. Our approach differs from the existing ones on the fact that we do not consider the clustering to be fully automated. We strongly believe that semantical clustering has to be done by including engineers and their knowledge of the application in the process. We applied our approach on a large legacy system in the context of an industrial project of architecture migration. This application consists of running two different instances of the generic iteration of our method on the legacy software system. Then results of our method are compared with the ones obtained by engineers of the company which owns the software. Results of the two instances are satisfying according to the engineers and according to the precision and recall rate we obtain.

Although recall and precision are still greater than the threshold given by the engineers, it could be improved. We plan on the future to improve the precision and recall by tuning the function that decides on the allocation of a software element. Another improvement track for the future is to provide tools to the engineers to the engineers in order to analyse the resulting clustering. Therefore, these analysis tools have to be defined and developed. A last lead for future works is to provide our method as a working tool for a software engineers team and check whether this method improves industrial reverse engineering and re-engineering process.

Acknowledgments

This work was supported by Thales and Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

[1] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of Working Conference on Reverse Engineering (WCRE'99)*, pages 235–255, 1999. ISBN 0-7695-0303-9. doi: 10.1109/WCRE.1999.806964.

[2] P. Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[3] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Softw. Pract. Exper.*, 26(1):25–48, 1996. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(199601)26:1¡25::AID-SPE994¿3.3.CO;2-K.

[4] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code. In *Proceedings of IWPC '99 (7th International Workshop on Program Comprehension)*, pages 136–143. IEEE, IEEE Computer Society, May 1999.

[5] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems and Software*, 28:117–127, 1995.

[6] A. Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of ICSE '99 (21st International Conference on Software Engineering)*, pages 246–255. ACM Press, 1999.

[7] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, CoSET '00, June 2000. URL `http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.`

[8] M. F. Dunn and J. C. Knight. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th international conference on Software Engineering*, pages 381–390. IEEE Computer Society Press, 1993.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[10] J.-F. Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *ICSM*. IEEE Press, 1997.

[11] R. Koschke. An incremental semi-automatic method for component recovery. In *Working Conference on Reverse Engineering*, pages 256–, 1999. URL `http://citeseer.nj.nec.com/koschke99incremental.html.`

[12] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, pages 349–359, Boston, 1997.

[13] M. Siff and T. Reps. Identifying modules via concept analysis. *Transactions on Software Engineering*, 25(6): 749–768, Nov. 1999.