Federal University of Pernambuco
Department of Computer Science
Computer Engineer Graduation

University of Leipzig
Department of Computer Science
Chair of Applied Telematics / e-Business
Prof. Dr. Volker Gruhn

Undergraduate Project

Deployment of Mobile Systems using Clustering Techniques

Author: Luiz Fernando Clapis Pacheco Chaves
Course of Study: Computer Engineer

Advisor: Dipl.-Inform. Clemens Schäfer
Chair of Applied Telematics/e-Business
Department of Computer Science, University of Leipzig

Co-advisor: Djamel Fawzi Hadj Sadok
Department of Computer Science, University of Pernambuco

Acknowledgement

I would like to thank my advisor Clemens Schäfer for accepting me and this work to go under his supervision and for the time and patience disposed. Thanks for making this experience possible and supporting it. More over I would like to express my gratitude to Djamel Sadok for the help and support provided, specially for his promptness.

I would like to thank all the professors that made the last years so remarkable and provided so many learning experiences. Such moments are certainly part of my education and will be eternally remembered.

I would also like to thank my friend Cesar Pinto, who I consider a pillar in my life and with whom I had the opportunity and honor to share so many happy experiences in the last 14 years. Thanks for always being there and forgiving my frequently negligent behavior. For being at hand in my hardest and loneliest moments, I am very grateful.

Abstract

The development of mobile systems has been growing dramatically in the last years. However such systems offer relatively narrow band to exchange data, constraining the development of more interactive and functional systems, as well as compromising its service quality.

This work intends to propose a method to determine the distribution of components among devices when deploying a mobile system, taking into consideration the amount of data exchanged between components. In this manner, demand for network resources can be reduced and quality of services improved. Our method makes use of techniques developed in software modularization, more precisely software clustering.

Table of Contents

List of Figures

1 Introduction

As computational systems pervade the society, more sophisticated, efficient and versatile solutions, demand that such systems collaborate and interact with one another. The current step in the information revolution that the world is going through cannot be described in a better word than *connectivity*. A huge information network is being established which is becoming an essential part or our infra-structure as electricity is today.

In this context, the development of mobile systems has been growing dramatically. These systems stand for the state of the art in connectivity, making it possible to be always connected into this great information network without the need to plug in cables. However with the growing complexity of interactions demanding greater amounts of data to be exchanged (e.g. youtube), communication channels become increasingly overwhelmed and such interactions are severally compromised. This happens not only in the core of this connectivity network, namely the internet, but in a more dramatic way in its terminations, more precisely mobile system's users.

Mobile systems usually offer relatively narrow bands to exchange data and because of that many mobile applications have their functionalities constrained or even become unfeasible due to network resource limitations. This issue has largely influenced the quality of mobile services and therefore demands new design efforts in order to compensate for and avoid associated problems.

In view of this problem, the development of solutions both in order to increase communication channels capacity and to reduce the amount of data exchanged across the network become important. Possible approaches include the reallocation of bands in the spectrum, development of more efficient modulation techniques, high-level compression codes for data, among others. Our proposal takes place in the software development process.

Under the Object Oriented paradigm, software is composed of objects interacting with one another through the exchange of messages. In the software development cycle these objects are initially designed as classes which have their role not so well-defined and as software evolves acquiring maturity, these become more stable with well-defined papers and functionalities. These elements will constitute the components of the system, which will provide specific services.

Once these components were implemented and tested, we reach the deployment phase. One of the decisions that must be made at this moment is how to distribute these components among the specified devices. Of course this question doesn't concern systems that are supposed to run on a single computing node. But in the context of mobile systems, where we have two or more devices, this question deserves special attention. Usually there is a specification that restricts placement of some components. It might be due to a specific functionality that must be provided in a certain device, because of some component that must be unconditionally deployed with another or even due to some restriction of language implementation or processor architecture. Besides these restrictions many components are deployed solely based on the software engineer's feelings.

Of course this engineer has a mental model which is very useful to evaluate deployment options for small scale systems. Although, as software system's become larger and more complex - as they naturally do - this activity becomes not so trivial and the number of component that could have been distributed on different computing nodes becomes of great significance.

This work intends to propose a method to determine the distribution of components among devices when deploying a mobile system, taking into consideration the relationship between components. The Unified Modeling Language (UML) provides specific diagrams that describe the interaction of classes – sequence diagrams. Therefore, such diagrams shall be our start point. More specifically, we intend to consider the exchange of messages and the amount of data carried by them in order to determine component's locality in deployment configuration. In this manner, demand for network resources can be reduced and quality of services improved.

In order to define our method, we have made an extensive research in the software modularization literature, searching for existing approaches from which we could take advantage. Software modularization is concerned with determining high-level groups formed by system's components in order to achieve modularity. This research, as we will see, led us to use clustering techniques to achieve our objectives.

In chapter 2 we briefly introduce clustering, detailing the techniques that we have chosen to use among the large spectrum of possibilities. In chapter 3 we present the research we have made in the software modularization literature and our classification of it. Chapter four contains indeed our proposed method, as well as a study case demonstrating its application. Finally, the conclusions can be found in chapter five.

2 Clustering Techniques

Our research in software modularization, as we will see in chapter 3, led us to an interesting field that presented itself very appropriated to our objectives. This is field is called software clustering, and it constitutes of applying clustering techniques in software. Thus, we dedicate this chapter to introduce, in a simplified way, clustering and some of its techniques. Also in this chapter we select the algorithms that will use as part of our method.

Clustering is a collection of data analysis techniques that groups entities into clusters in order to discover similarity and differences among them. It reveals the underlying data structure and allows the derivation of useful conclusions about the entities. Clustering is a very useful technique when handling large amount of data due to its ability of summarizing the properties of the entities within a cluster and indentifying the main characteristics that differentiate such groups [Webb02].

Our every day mental activity of classifying objects, persons, events and other entities into groups can be considered a clustering activity. It allows us to handle the huge amount of information that we receive during the day, which couldn't be all processed piece by piece. So when a person sees an entity laying on the grass, which has an innumerable number of characteristics (e.g. number of legs, size) and he classifies it as a dog, it's because the representation of his classification group 'DOG' has similar characteristics to the entity laying on the grass. This entity has features that are common to every dog and so the person doesn't need to process all its characteristics to classify it – e.g. he can tell it is a dog even though he has never heard it barking. [TK03].

Clustering is used in a wide range of applications and can be found under different names according to the field of study, such as unsupervised learning in pattern recognition, numerical taxonomy in biology and ecology, typology in social sciences or partition in graph theory. Therefore there are several terminologies and classifications – sometimes conflicting - for the components that constitute clustering. We shall use the term entity for the elements that will be grouped, and the term cluster for such groups that contain the similar entities. We present here in this chapter a rough overview of clustering and its components.

There are three main issues or steps that demand our attention when applying clustering techniques:

- Representation: this issue will answer the question: what are the entities to be clustered? Here we define which features will describe the entity.
- Similarity: quantifies how similar two entities are according to their features. The term dissimilarity is also often used to indicate how different such entities are.
- Algorithms: the step taken to group the entities.

Handling these issues with the purpose of revealing natural groupings in the data depends on specific domain knowledge. It is necessary to understand the addressed problem in order to determine the most suitable representation, similarity measure and clustering algorithm. Classifying a group of persons by their

preferred sports or by the country they were born will result in different clusters with different meanings.

In the further sections of this chapter we survey each of these issues.

2.1 Representation

The representation issue is responsible for building an abstraction of the real world in which the entities to be clustered are described according to some scheme [ALL99]. This step is of the utmost importance, since it determines almost all information available during the clustering process.

Each entity is described by certain number of selected features. There might be a 'selection feature' pre-process phase to determine which features are relevant, but there is no guideline for that. Usually specialists in the application domain provide such information. For example, in taxonomy, entities are organisms, like animals and plants, and are described using features like "color of the flower", "size of the seed", "laying eggs", etc. Each entity will then be represented as a vector in the d -dimensional space of features – each dimension is associated with a feature.

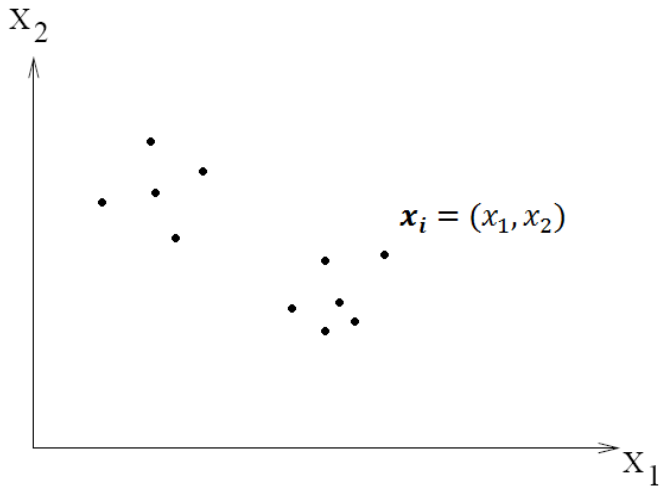


Figure 1 - Entities in the 2-dimensional space of features

These features can be classified into qualitative and quantitative [JMF99]. The quantitative features are numerical values which can be subdivided into continuous (e.g. weight), discrete (e.g. number of leaves) or interval values (e.g. duration of an event). The qualitative features can in turn be subdivided into nominal (e.g. color) or ordinal, when their ordination is meaningful (e.g. sound intensity, "loud" or "quiet").

In software clustering literature, entities may be files, routines, classes, processes and chunks of code. The features are usually references to user defined types, references to global variables, routines called, files included or macros used. Some other features, called non-formal, like reference to words in identifiers or reference to words in comments, proved also having interesting advantages [BMC96].

2.2 Similarity

Similarity is a term often used in clustering that refers to how closely related two entities are. Since clustering algorithms group entities that are similar in order to reveal a natural structure in the data, measures to determine this proximity must be specified. As an example, we can take the most popular metric for quantitative continuous features, the Euclidian distance:

$$d_2(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2\right)^{1/2}$$

$$= \|\mathbf{x}_i - \mathbf{x}_j\|_2,$$

The Euclidian distance is in fact a dissimilarity metric, which has the opposite meaning of similarity and is more often used. It is a very simple metric that has an intuitive appeal because it can be easily visualized in a two or three dimensional space (figure 2).

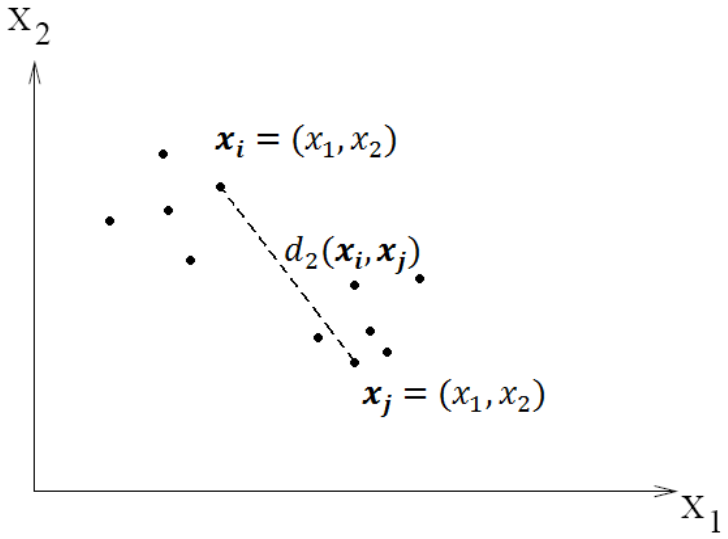


Figure 2 – The Euclidian Distance metric

There are innumerable measures available in the literature. These measures are based on the type of features used and must be carefully chosen since they have a profound impact on the resulting clusters. Besides that, domain knowledge has also crucial importance in this issue. Certain measures can handle specific data properties that are related to the problem under discussion. In the words of Theodorius and Koutrombas [TK03], “Each of them gives a different interpretation to the term similar”.

Once we have our entities described and a similarity measure specified it is possible to define a matrix that contains the calculated distance between all entities. This is very useful for a certain class of algorithms, and will have particular importance in our process. Such matrix, called Similarity Matrix, is a simple symmetric matrix where the value of the i^{th} row and j^{th} column is the similarity distance between the i^{th} and j^{th} entities. It is worth saying that such matrix is symmetric because $d(x_i,x_j)$ is always equal to $d(x_j,x_i)$.

2.3 Clustering Algorithms

The last main issue is to determine the clustering algorithm that will be used to group the entities. There are several algorithms available in the literature and also several different classifications for them. We will survey briefly the most important categories and investigate the two algorithms that we chose use in this work.

The first category of clustering algorithms is called sequential algorithms. These algorithms usually assign entities to clusters in one pass and obtain a single partition of the data. The entities are presented a few times to the algorithm and the resulting clusters have great dependency from the presentation order. However they are the less computationally expensive and thus present results faster. Well-known algorithms in this category are the Basic Sequential Clustering Algorithms, the Modified Basic Sequential Algorithm, the Max-min and the Two-threshold sequential scheme.

Hierarchical clustering algorithms constitute another category of clustering algorithms. Such algorithms produce not only a single partition but a hierarchical structure of partitions. This nested grouping of entities is particularly interesting for some application domains, where such structures are aimed or already exist (e.g. taxonomy). Because software systems should also present a hierarchical structure, we have decided to make use of such algorithms. We will take a closer look at them in section 2.3.1.

The last main category embraces the algorithms based on function optimization. In such algorithms an initial partition of the entities and a function that measures how optimal is the actual solution are provided. Then small changes are made to the partition with the intention of obtaining higher values for the given function. Such approach, based on differential calculus concepts, converges always to a local optimum. The Bunch tool that will be presented in the section 3.2 belongs to this category. A shortcoming of such algorithms is the high computational cost.

Beyond these three main categories there are several others clustering algorithms like the ones based on morphological transformations, stochastic relaxation methods and competitive learning algorithms. A very interesting class of algorithms among these is composed from clustering algorithm based on graph theory. In such algorithms, nodes represent entities and edges represent relations, making their application to software straightforward, once UML class diagrams are represented through boxes (classes) and lines (relationship, e.g. association, composition). Furthermore such algorithms have a very solid mathematical background which we can profit from. Thus, we decided to make use of an interesting algorithm from this class, which we will examine in section 2.3.2

2.3.1 Hierarchical algorithms

As said before, hierarchical clustering algorithms produce several nested partitions. These algorithms can be subdivided into agglomerative algorithms and divisive algorithms. The later ones consider an initial cluster containing all the N entities. Each step of the algorithm splits a cluster into two. After N–1 steps we have N clusters containing a single entity. The agglomerative algorithms work on the opposite direction. It starts from individual entities gathering them into small clusters which are in turn gathered into larger clusters up to one final cluster

containing everything [AL99]. Because divisive algorithms must consider all possible divisions of the clusters, what is computationally a very expensive task, they are not used so often.

The resulting nested partitions are commonly represented through a tree diagram or dendrogram (figure 3). The hierarchical algorithms have also the advantage of being able to easily provide a single partition when necessary, just by sectioning the dendrogram at a given level.

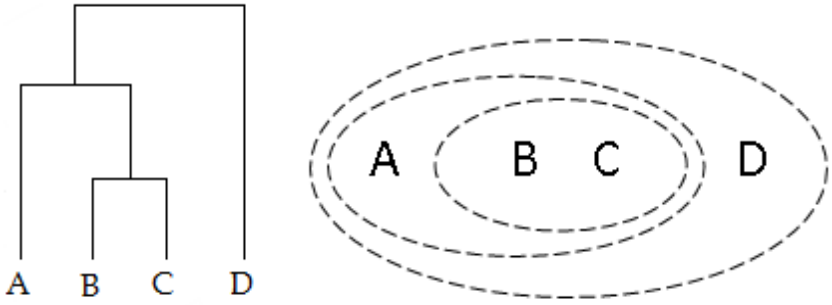


Figure 3 – Dendrogram and nested clustering

The main representatives of the agglomerative algorithms are the Single Link and the Complete Link algorithms. The difference between them arrives when joining two clusters into one and both clusters had previously different similarity values in relation to a third cluster. This situation can be better visualized through the following figure:

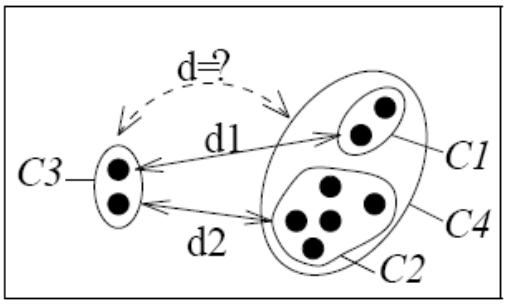


Figure 4 – Single Link vs. Complete Link

Clusters C1 and C2 were joined in one new cluster C4. C1 and C3 have similarity d1 while C2 and C3 have similarity d2. The question is: how do we determine the similarity between C3 and C4? The single link algorithm preserves the maximum value and doing so produces elongated clusters – what is called chaining effect. On the other hand, the complete link algorithm chooses the minimum value, resulting in more compact clusters. There are also several variants of these two algorithms proposing intermediate solutions.

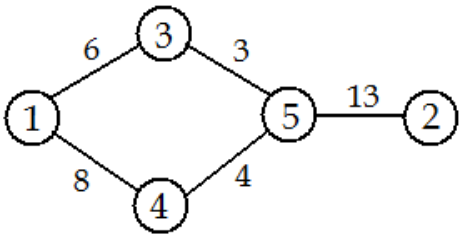
Because the algorithms in this class need constantly to know the similarity between the entities, they work over a similarity matrix, which has all values previously calculated. This also reduces the execution time of the algorithm, presenting results faster.

2.3.2 Clustering Algorithms based on Graph Theory

In order to make use of clustering algorithms based on graph theory, data must be represented as a graph. Entities will be represented by vertices and relations by edges. The similarities between the entities correspond to the weight of the edges. Generally, the clustering algorithms in this category will try to find in this graph sub-graphs to compose the clusters. Graph theory provides the basic algorithms to find sub-graphs with special properties (e.g. spanning trees, maximal complete sub-graphs) that will be themselves candidates to form these clusters or will be the input for the clustering algorithms to find them [Wig97].

Spectral graph partitioning is a special technique that instead of working on a graph makes use of the properties of its algebraic representation (e.g. Adjacency matrix, Laplacian matrix). The eigenvector associated with the second smallest eigenvalue of the Laplacian matrix is called Fiedler vector, in homage to Fiedler [Fie75] who investigated its relation to graph's properties. It has been proved that clustering the vertices of a graph in two sub-graphs according to the positive and negatives entries of such vector will lead to a partition which minimizes the total weight of the edge cut between the two sub-graphs [CTS06].

To demonstrate such technique let us consider the weighted graph below:



The Laplacian matrix of a graph G is defined as $L = D - A$, where A is the Adjacency matrix of G and D is the *degree* matrix $D = [d_{ij}]$ defined as:

$$d_{ij} = \begin{cases} \sum_{k=1}^{n} a_{ik} & ,if\ i = j \\ 0 & ,if\ i \neq j \end{cases}$$

The Laplacian matrix of our graph is shown below:

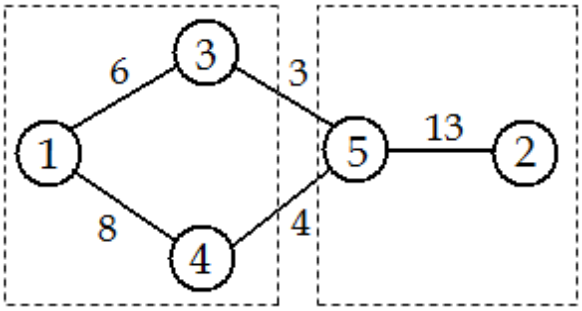
$$LM = \begin{bmatrix} 14 & 0 & -6 & -8 & 0 \\ 0 & 13 & 0 & 0 & -13 \\ -6 & 0 & 9 & 0 & -3 \\ -8 & 0 & 0 & 12 & -4 \\ 0 & -13 & -3 & -4 & 20 \end{bmatrix}$$

Calculating eigenvectors is an extremely computationally expensive task and is the major shortcoming of this technique. We will make use of Scilab [Scilab08], which is an open source platform for numerical computation, in order to calculate the Fiedler vector:

$$v_2^T = [-0.4552382 \quad 0.6534043 \quad -0.3446014 \quad -0.2705478 \quad 0.4169830]$$

The vertices corresponding to the negative values of this vector (1, 3 and 4) will compose a cluster while the ones corresponding to the positive values (2 and 5) will form another. This partition is ensured by spectral graph partitioning to be the

optimum clustering. Besides that, it's possible to obtain a hierarchical partitioning such as the ones produced in the previous section executing the same procedure to each resulting cluster.



3 Software Modularization

In this chapter we present the results of the research we have made in the software modularization literature. We have looked for previous works from which we could take advantage in order to achieving our goals.

Great part of the works in Software Modularization consists of defining modularity or proposing principles that would lead software designers to it. Such works don't bring much contribution for us. We are interested in the techniques that were developed to achieve modularity based on numerical approaches. Such techniques have value for us because our objective is mainly focused on the amount of data exchanged between components and not on the quality of the design. The majority of works that satisfy this condition is situated in a subfield called software clustering and that is the reason why we have chosen to use clustering as part of our proposal.

We have made a classification of the works in software modularization which is summed up in the table presented in Appendix I. Here we provide an overview from some of these works, investigating deeper the most relevant ones. In section 3.1 we talk about modularity as the fundamental concept in software modularization and relate some works that propose principles to achieve it. In section 3.2 we examine the works on software clustering. Section 3.3 presents the Design Structure Matrix, a very simple and powerful representation of software design structure which has been used intensively in the latest modularization works and will be used in this work. In the last section we present other works in software modularization.

3.1 Modularity

During software development process, the project will grow in size and complexity and it is almost certain that software requirements will change. New engineers will have to be allocated and even after release, the software must be maintained and of course evolve, adding new functionalities and changing existing ones – what is quite often not done by the original designers and developers. Because of this great mutability, some kind of high level organization for the software becomes imperative.

Since the earliest days of software development, larger organizational units have been extensively studied in several software disciplines (e.g. software modularization, software comprehension, package design) and have received several different names. "Booch uses the term 'class category' to describe such a granule, Bertrand Meyer refers to 'clusters', Peter Road talks about 'subject areas', and Sally Shlaer and Steve Mellor talk about 'Domains'. " [Mar96]. They have also been called more recently subsystems, groups, modules or packages [MT07a].

Determining how these modules should be constructed in order to achieve modularity is the greatest issue addressed by software modularization. Modularity is perhaps the most widely accepted quality objective for design [ABA04], since it reflects the most desired quality attributes of software:

- Understandability: In view of the size and complexity of nowadays systems, understandability is the most desired quality attribute. A high level structure aimed by modularization provides an easy map to understand the system. Moreover the great flow of new engineers that will have to catch up with the project during its life cycle, makes understandability an even more dramatic issue.
- Reusability: is the degree to which a software module or other work product can be used in more than one computer program or software system. When designing an electronic system, electronic engineers use components whose interfaces and functionality are well defined and formally described. Software Engineer aims a similar relation with software modules, so that the development of a system can be improved, spending less time re-writing modules and avoiding making mistakes that have been already corrected.
- Testability: is the ease at which software can be made to demonstrate faults through testing. Well-defined modules can be easier automatically exercised by another piece of code and the results compared with the expected output.
- Maintainability: refers to the ease of finding and correcting errors as well as adding up code to correct unforeseen problems or new functionalities requested after the software has been released. It is the longest and most expensive phase in the software cycle. If changes can be done in one module instead of changing the whole system or a new functionality can be provided by simply adding a new module is great improvement in software maintainability.

Several principles that would lead to modularity have been proposed in the software modularization literature (e.g. [Mar96], [MT07a]). These principles brought - and still bring - enormous contribution to software engineering and some of them have until today great importance. For example, Parnas proposed as criteria to decompose systems into modules the information hiding principle [Par72], expressed nowadays in concepts such as encapsulation and polymorphism. However because modularity is a subjective concept and has no single, precise, definition accepted in almost 50 years [MT07b], such principles are still under discussion.

Even though, two well-known concepts are widely used in the literature to characterize modularity: coupling and cohesion. Booch's definition of modularity is perhaps the best expression of this use. According to Booch, modularity is "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules" [Boo94]. That means that the elements within a module must have a strong relationship with one another and a weak relationship with elements from other modules. This definition was used in several works to support numerical methods which we are interested on. Therefore it is of great importance to us.

In the next sections we will survey the works in software modularization, taking a closer look at the most relevant to the purposes of this work.

3.2 Software Clustering

The initial works on software modularization can be found in the reverse engineering literature, under the name of *software re-modularization*. The main reason is that code was usually the only available specification of the system [DMM99]. Until today most of the projects are not properly documented or even documented. So during maintenance, software professionals spent at least half of their time reading and analyzing software in order to understand it [DLP05]. Software re-modularization proposes to recover the high-level structure of a system through analysis of code. This process would automatically reveal the modular structure of the system, providing developers with a road map to understand it.

The use of clustering techniques in software gave birth to a domain called software clustering. The works that established the fundaments of such domain were also the earliest ones in software re-modularization. Among these, there are three that with no doubt deserve to be mentioned. Wiggerts [Wig97] presented an overview of clustering analysis, providing theoretical background for applying it in systems re-modularization. Concepts like similarity between entities, including its measures, and categories of clustering algorithms were introduced aiming modularization of “chunks of code or procedural chunks”. Anquetil and Lethbridge [AL99] gives continuation to his work with perhaps one of the most cited works in software clustering literature. They take the background and clustering techniques provided by Wiggerts, and test them to compare their efficiency. Among several extension of Wiggerts’ work, they give special importance to the description issue proposing formal features (e.g. global variables referred to by the entity) and also informal features (e.g. references to words in comments describing the entity, name of the files), which proved having also great value. Further investigation in the same direction is done by Davey and Burd [DB00], clustering pieces of code as proposed by Wiggerts. These three works consolidate the conceptual use of clustering techniques in software modularization.

In this context Arch and Rigi are two tools that also have special importance. Arch [Sch91] provides clustering algorithms that group related procedures into modules using a heuristic design similarity measure based on Parnas’ information hiding principle. It also indentifies individual procedures that apparently violate this principle. The second tool Rigi [MOTU92] considers that software structure is the collection of artifacts used by software engineers when forming mental models. These artifacts would include software components such as procedures, modules, interfaces, dependencies among components and attributes such as component type and interconnection strength [MNT94]. Rigi helps software engineers to identify these artifacts and aggregate them to form more abstract system representations.

However these works and most of the earlier works are situated at a too low granularity level. That means that the entities to be clustered are usually pieces of code, functions, procedures or variables. With the increasing size of systems it became necessary to go upward in the hierarchy looking for solutions at a higher granularity level. A remarkable work at in such context is presented through the Bunch tool [MM06]. Bunch is a software clustering tool that has accomplished great results and brought enormous contribution to software clustering. Due to its importance, we shall take a closer look at it.

Bunch proposes an automatic technique that creates a hierarchical view of the system organization based solely on the components and relationships that exist in the source code. The first step of Bunch's procedure consists of analyzing source code and constructing a MDG – Module Dependency Graph – that will represent the system. A MDG is a directed graph where each node represents a source file and the directed edges represent source-level dependencies (e.g. procedural invocations, variable access). There are several tools that automatically accomplish this task such as CIA [Che95] for C, Acacia [CGK97] for C and C++ and Chava [KCK99] for Java. The second step is to determine the best partition to the graph. That is, the best decomposition of the set of all nodes into mutually disjoint clusters. Bunch treats this problem as an optimization problem and for that defines a measure called Modularization Quality. MQ (Modularization Quality) determines the "quality" of an MDG partition quantitatively as the trade-off between inter-connectivity (i.e. dependencies between the modules of two distinct subsystems) and intra-connectivity (i.e. dependencies between the modules of the same subsystem). Initially MQ was defined as:

$$MQ = \begin{cases} \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} & \forall k > 1 \\ A_i & k = 1 \end{cases}$$

$$A_i = \frac{\mu_i}{N_i^2} \qquad E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\varepsilon_{ij}}{2N_i N_j} & \text{if } i \neq j \end{cases}$$

A_i stands for the intra-connectivity of a cluster i with N_i components and u_i intra-edges dependencies (relationships to and from modules within the same cluster). This would correspond in Booch's definition of modularity to coupling between modules. $E_{i,j}$ denotes the inter-connectivity between clusters i and j, each consisting of N_i and N_j components respectively, with ε_{ij} inter-dependencies (relationships between the modules of both clusters) [DMM99]. In Booch's definition: cohesion within a module. The values of MQ vary between -1, where there is no internal cohesion and 1, where external coupling is null. In the figure bellow we have an example of its calculation.

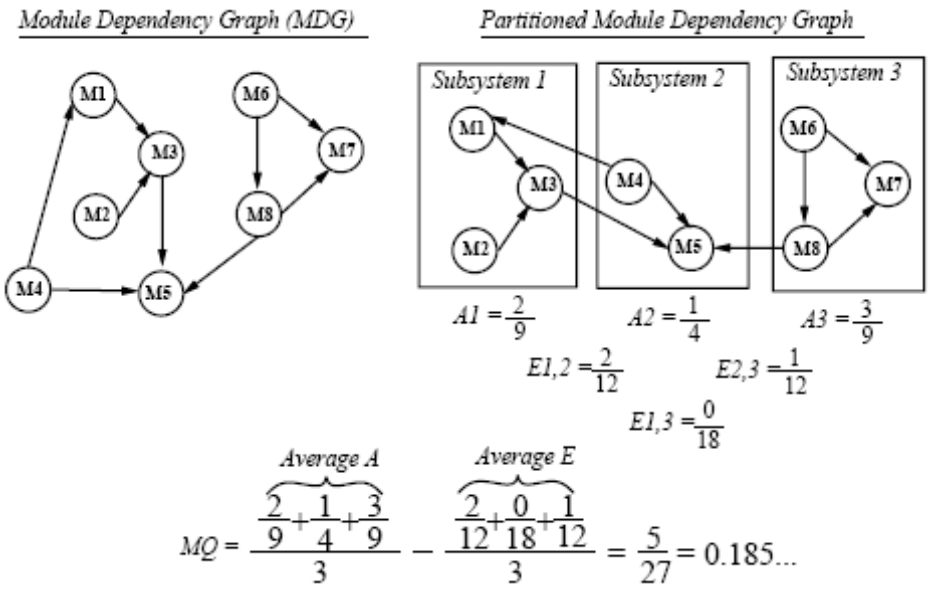


Figure 5 – Calculating the Modularization Quality of a MDG

The problem of such definition is that it doesn't take into account the interconnection strength of the relationships that exit between the modules in the software system. This flaw was overcome in the last works of Bunch with an evolution of the MQ measure in a way that it can be applied to weighted directed graphs:

$$MQ = sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise. \end{cases}$$

The new Modularity Quality measure is calculated summing the Cluster Factor (CF) for each cluster of the partitioned MDG. The CF is defined as a normalized ratio between the total weight of the edges within the cluster and half of the total weight of the edges that exit or enter the cluster [MM06].

Bunch adopted three algorithms to explore partitions that maximize the value of the MQ. The first one leads to the optimal solution since it calculates the MQ for all the possible partitions of the MDG. The obvious shortcoming of this approach is the computational cost, since the number of possible partitions grows exponentially with respect to the number of nodes. For example, a 5-node module dependency graph would have 52 partitions, while a 15-node module dependency graph would have 1,382,958,545 distinct partitions. The second algorithm uses a search strategy based on traditional hill-climbing optimization techniques. Such technique defines a random initial partition as the actual partition. Then, small perturbations are provoked aiming to achieve a better MQ value. When such partition is found, it becomes the actual partition. This algorithm repeats this procedure to the actual partition until no further better partition is found. The small perturbation constitutes of neighboring partitions, which are, exactly the actual partition except for one node that was moved from one cluster to another. This

approach profits a lot from the modularization quality definition in terms of computational cost, since the Cluster Factor must be calculated only for the two modified clusters. The last approach of Bunch is the use of Genetic Algorithms. Such algorithms are well-known for its ability of solving problems that involve large search spaces.

The good results presented by Bunch can be attributed to the function optimization approach adopted and the Modularization Quality measure defined. It would be interesting as further work to adapt Bunch in such a way we could apply it to our method in order to compare the results with the algorithms selected in this work.

The idea of using Genetic Algorithms to automatically cluster software systems presented great value and several other works emerged proposing improvements. Evo [SBBP05] and DAGC [PB05] implemented genetic algorithms supported by domain knowledge - more suitable encodings to map the clusters into a sequence of genes, genetic operators and fitness functions taking into consideration aspects like cluster size and cyclic dependencies.

All the works inspected so far, as said before, are situated in the context of software re-modularization, what is also commonly referenced as a bottom-up approach since the start point is the source code. In addition to this we found some works commonly referenced as a top-down approach. Even though the name might suggest that design artifacts, instead of implementation artifacts, would be taken as start point, such works try in fact to minimize the distance between them. Murphy [MNS01] developed a technique called “Software Reflexion Model” which captures and exploits the differences between design artifacts and source code in order to maintain design conformance. Another recent work with the same objective is presented by Huynh and Cai [HC07].They use the two different tools to extract the same type of representation from a modeled design and source code respectively. In order to make the two representations coherent with one another, some mapping must be done. A genetic algorithm takes then one representation as the optimal goal and searches for the best clustering in the other representation that maximizes the level of isomorphism between the two. The great advantage of this work though, hides in the power of the used representations – Design Structure Matrices.

3.3 Design Structure Matrixes

Design Structure Matrix, also referred as Dependency Structure Matrix or DSM, is a very simple and powerful way of representation, invented for optimizing product development processes. When developing a product a collection of tasks is defined and performed. Most of these tasks present dependencies to other tasks, either because some physical artifact flows from task to task or because some task requires information provided by another task. In order to improve management of these tasks and consecutively the efficiency of the process, DSM provides a structure to represent such dependencies and algorithms to organize them.

		1	2	3	4
Task A	1	.		X	X
Task B	2		.	X	
Task C	3	X		.	X
Task D	4				.

Figure 6 – Design Structure Matrix

The matrix is a simple adjacency matrix with tasks labeling the horizontal and vertical axes. If the i^{th} task depends on the j^{th} task, there will be a mark in the i^{th} column and j^{th} row. For example we can see, examining the third column in the figure above, that task C depends on tasks A and B. Moreover, a numerical value can be used instead of a mark to represent the strength of the dependency.

Software design structure is commonly represented though box-and-line diagrams (e.g. UML class diagram). However, such representation is not very suitable when handling large scale systems. Even though DSM is not a modularization technique, recent works in this field have been using it intensively to represent design structure of systems (with modules playing the role of tasks) due to the advantages brought by its properties:

- Cyclic dependencies: An important characteristic of DSMs is that the cyclic dependencies of the system can be easily exposed. If a DSM has no cyclic dependencies, columns and rows can be automatically permuted through algorithms, so that the matrix becomes lower triangular – that means, it has no entries above the diagonal. In software a lower triangular matrix would represent a layered system.

			1	2	3	4
Task D		1	.			
A-C	Task A	2	X	.	X	
	Task C	3	X	X	.	
Task B		4			X	.

Figure 7 – Cyclic Dependencies in DSM

  For the tasks presented in figure 6 we can see above an attempt to organize the matrix in a lower triangular form. However this is not possible and the cyclic dependency responsible for it is automatically exposed (figure 7). In order to eliminate such dependency, tasks can be aggregated creating so a hierarchy, as depicted in the figure above.

- Scalability: Through DSMs, complex systems can be much better visualized than box-and-line diagrams. Moreover, the possibility of creating a hierarchy meets perfectly software engineering needs.
- Pattern identification: The preponderance of dependencies bellow the diagonal exposes the layer patter of the system even when it is bad implemented.

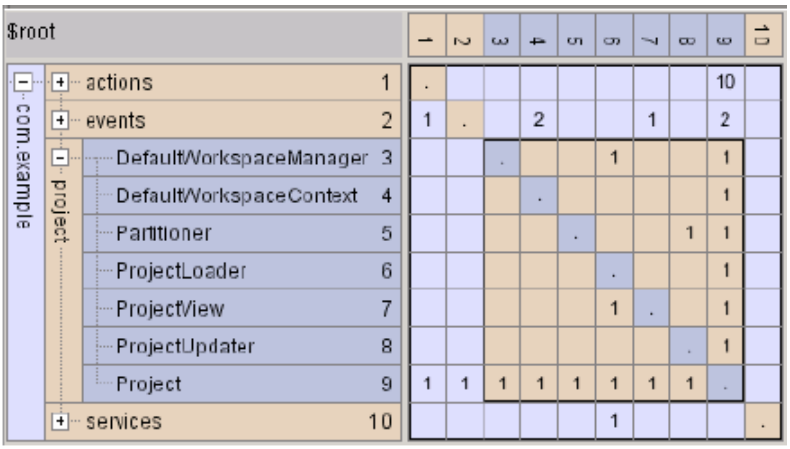


Figure 8 – Patterns Identification in DSM

Other patterns become also exposed. The DSM above, for example, exposes a *change propagator* (module 9, 'Project') - a module that depends on a large number of modules and in turn has many modules depending on it. Modules that depend on no other module (e.g. module 10, 'services') or modules that no other module depend-on are also easily identified through empty columns and empty rows respectively.

3.4 Other Works

As said before, modularization is subjective concept with no precise definition. The reason for that is that software systems are abstract models, and much of the relevant information about it is not related in the traditional software artifacts. Beyond data structures, bindings, procedure calls, interfaces, number of coded lines and any other usual numerical description of the system, software has semantic information which cannot be captured from most of the current programming paradigms. To cope with such challenges, new programming paradigms are emerging to enhance traditional programming languages and modeling techniques with additional modularity mechanisms and abstractions. Great examples of this are Aspect-Oriented Programming (AOP) and Feature-Oriented Programming (FOP) [Acom07].

Much of the work developed in this direction consists of empirical studies with supporting assessment techniques to improve understanding of the benefits and drawbacks of new software techniques and comparison with traditional paradigms [Acom07]. Therefore they don't bring many contributions to us and we shall not enter this field. Moreover it is not our intention here to propose solutions to the modularization issue, but to take advantage of the gathered experience and developed techniques in software modularization so that we can use them for our objectives.

The last category of works in software modularization intends to characterize the software granules at a given level of abstraction through the properties of the directed graph formed by them (e.g. PDG or Package Dependency Graph). Such works do not propose design principles to determine how these granules should be formed, but methods to evaluate if a given design is good according to defined metrics.

Meldon and Tempero [MT07a] present an interesting work within this category. They define a metric called Class Reachability Set Size (CRSS) which counts, for a given class, the number of classes in the system's source code that it transitively depends-on. Then the distribution of CRSS values of a PDG representing a system can be depicted in a histogram.

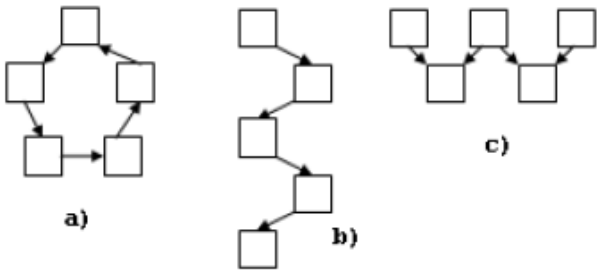


Figure 9 – Package Dependency Graphs

In Figure 9 we can see the PDGs that represent the dependencies among packages of three different systems. If we assume that each package has the same number of classes and that every class in a package depends on every other class in the same package then we get distributions like these:

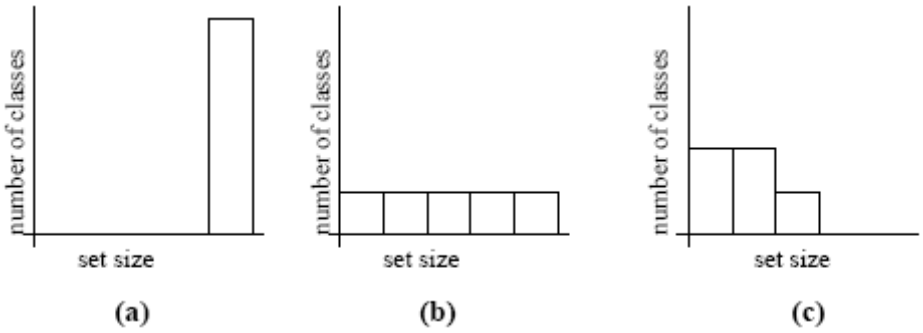


Figure 10 – CRSS Distribution

The histogram express how many classes (axis x) depend on a number y of other classes. Thus design could be evaluated analyzing these distributions – bars concentrated on the left side of the histogram reveals a good design. As we can see, system (a) has the worst distribution since all classes depend on every other class of the system. On the other hand, system (c) has the best distribution among the three since most classes of the system depend on few other ones.

Such approaches can also be very useful for us since they are numerical approaches and such metrics could be used in our clustering techniques. Unfortunately we had no time to attempt solutions in this direction, but it we leave it as suggestion for further works.

4 Building the Bridge

Our proposal aims to develop a method to determine component's location when deploying a mobile system. This method must consider the amount of information exchanged between components, in order to make data exchange between devices, and consecutively through network, as low as possible.

Considering the amount of data exchanged between components demands a precise analyze of their relationship. Because of that, our method has as start point UML artifacts (e.g. class diagrams, sequence diagrams) which provide such kind of information. However these artifacts do not contain information enough to build a bridge between design and deployment. We will find the bricks to handle this question in the section 4.1.

Once this bridge is established, we shall have the precise kind of information we need to apply the clustering techniques, we have chosen. In section 4.2 we deal with the issues of representation and similarity in clustering. As one can already imagine, similarity must be closely associated with the amount of information exchanged between components. In section 4.3 we describe the application of the different algorithms we have selected in chapter and examine the results throw a study case in section 4.4

4.1 Data Coupling

As we have said, in the software modularization literature coupling and cohesion are widely taken as measures that translate modularity. Although if we take into account that large systems consist of subsystems, which in turn consists of other subsystems, creating so a layer hierarchy, we can have an interesting insight into these concepts. Let us take a look at the hierarchical system below:

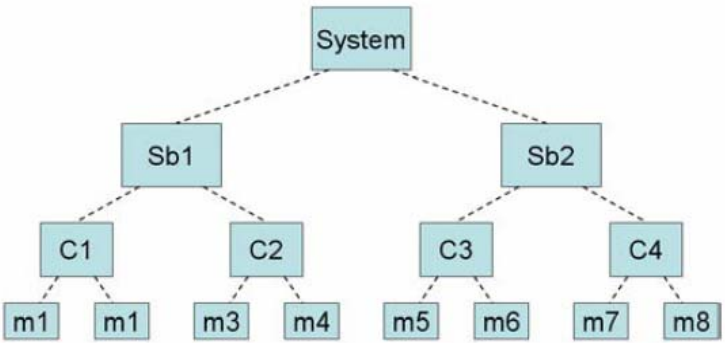


Figure 11 – Hierarchical structure of an ideal system

According to Lakos [Lak96] this binary tree is an ideal software system design. Each layer of this system represents an abstraction level. As we go upward in this system, the coupling between the components of the same abstraction level decreases [YR07a]. This can be observed through the arrows' thickness in the modular structure depicted below:

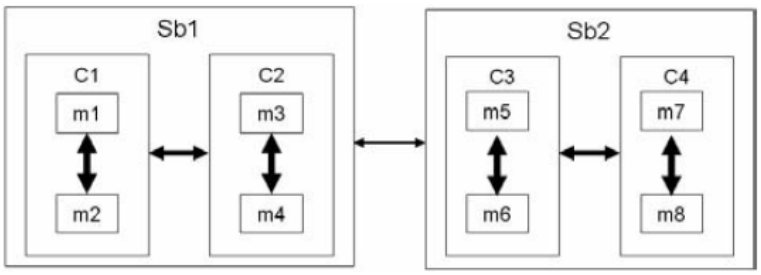


Figure 12 – Modular structure of an ideal system

Both coupling and cohesion relate to the relationship strength between modules, but at different abstraction levels or granularity levels. Therefore, in contemporary systems, which need several abstraction layers due to its size and complexity, it is more suitable to use only the term coupling associated with an abstraction level, instead of trying to determine if the interaction degree of two components should be considered coupling or cohesion.

Because there are several different types of interaction between components, many types of coupling can be identified. In the object-oriented paradigm, coupling can be categorized in four types [YR07b]:

- Parameter Coupling: Two classes have parameter coupling if one class invokes method of the another class via parameter passing;
- External Coupling: Two classes have external coupling if they access the same external medium including external files;
- Inheritance Coupling: Two classes have inheritance coupling if one class is a descendant of the other class;
- Common Coupling: Two classes have common coupling if they access the same global variable;

The interactions brought by an external medium do not interest us, but only the direct interaction between components. Common coupling should be deprecated in object-oriented software since it allows one class to access attributes of another class, rather than through message passing [YR07]. Inheritance coupling is a strong type of coupling that is easily exposed in UML class diagrams like the one in figure 13. However, it has no part in data exchange, and hence null value to this work.

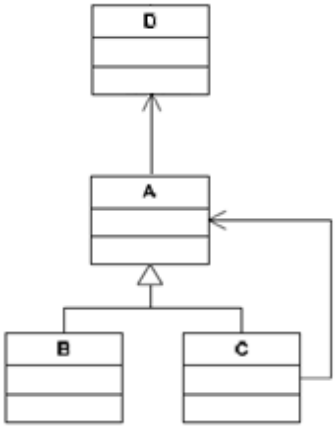


Figure 13 – Example Class Diagram

Since we are only interested in the amount of data exchanged between the components, the only interesting coupling type is the parameter coupling. Even though it is considered a weak type of coupling, parameter coupling plays the main part in our work. Inheritance coupling, for example, has a much more important role in software modularization, since it is directly related to reuse.

In order to analyze parameter coupling, we need to investigate the messages exchanged between components at method level. UML provides sequence diagrams that expose exactly such information. For example, the system presented in figure 13, could have the following sequence diagram describing a particular use case:

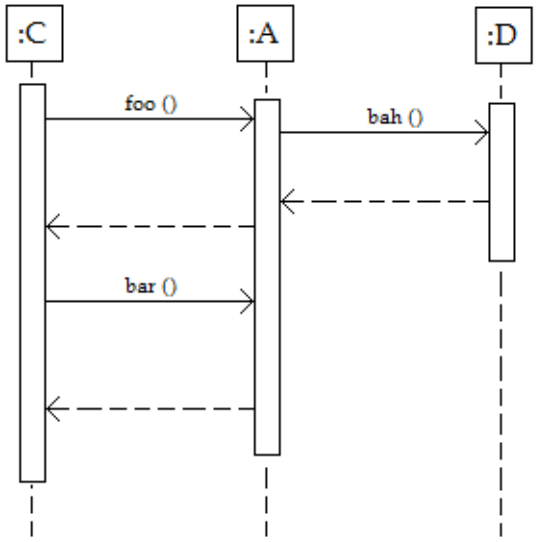


Figure 14 – Common UML Sequence Diagram

However this sequence diagram does not provide accurate information about these interactions. In order to understand that, let us analyze the following source-code, which is a possible implementation for this sequence diagram:

```
class C {
      void run () {
            A.foo(obj1); //obj1 has size 2
            A.bar(obj2);
```

```
            A.bar(obj2); //obj2 has size 5
        }
}

class A {

        Object foo (Object obj) {     //..
            D.bah(obj3); //obj3 has size 3
            D.bah(obj3);
            return obj4; //obj4 has size 4
        }

        void bar(Object obj){
            //...
        }
}

class D {
        Object bah (Object obj) {
            return obj5; //obj5 has size 1
        }
}
```

This code reveals three kinds of information that hide behind each call presented in the sequence diagram:

- Parameter size of a call: In class C at the ‘run()’ method definition we can see two calls to the same method ‘bar()’ of class A, but with different parameters size. If the parameter has size 5 or 500, is a very important information, considering that this data load must go through a narrow-banded network;
- Frequency of calls: Also the fact that ‘bar()’ is called 2 times, and not 20 or 1, must be considered when measuring the exchange of data;
- Return size of a call: Most methods return some information to the calling object. This data might also have to be transported through the network;

Because the data exchanged in a method invocation is not only attributable to parameters, we think it is a more suitable to use the term data coupling instead of parameter coupling.

Even though we are able to identify such information, it is not possible to determine it precisely because we cannot predict code interactions. Perhaps some evaluation at run time could bring more precise values, but that is beyond the scope of this work. We will assume in this work that such information is provided to us.

Hence, we can build a richer UML sequence diagram to expose such information. For the parameter size and return size we will just write it down at the calling and returning arrow respectively. And to expose the frequency of calls we shall write before the name of the method the number of times it is called followed by an ‘x’ character. Thus, we would have for the following sequence diagram for the provided example:

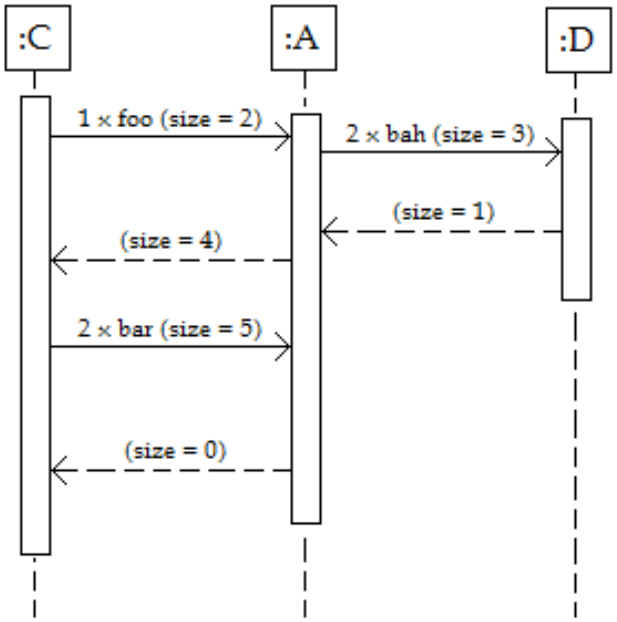


Figure 15 – Rich UML Sequence Diagrams

4.2 Representation

Once we are going to use clustering techniques, we need to handle the three issues we have spoken about in chapter 2: representation, similarity and algorithm. In this section we shall handle the first two and in the next section we deal with the clustering algorithms.

The normal clustering approach would be to determine the features that will describe the entities – in our case, classes – and specify a similarity measure between these classes. For our objective though, we wish that two classes exchanging great amount of data have great proximity so that they can end in the same cluster after we apply our algorithms. Thus our similarity measure will directly relate to data coupling. Because such coupling information can be extracted from our richer sequence diagrams, there is no need to describe the classes through features. Instead we propose a sequence of steps to extract a numerical representation for data coupling and to obtain the similarity matrix that will be the input for our algorithms.

Since we have to make an analysis of the system at level method and we are dealing with systems of great complexity and size, we need a more efficient way to work with this information than box-and-line representations, which become almost impossible under these circumstances. DSMs fit like a glove in our problem. We will profit from DSMs' ability to visualize complex systems with the extra advantage of working in a matrix and thus making the extraction of a similarity matrix straightforward, as we will see. The basic DSM structure for our example is shown below:

			1	2	3	4
Component C	run()	1	•	0	0	0
Component A	foo()	2	0	•	X	0

• recursive calls

X calls inside component

	bar()	3	0	X	•	0
Component D	bah()	4	0	0	0	•

0 calls between components

As we can see, classes are already named as components. Classes and components correspond to the same abstract element at different development phases. In design such elements are called classes and provide functionalities through its methods. In the development phase they are called components and provide services. The differences between them are very subtle and are mostly related to reuse. Classes belongs to development phases before deployment and therefore are more subjected to changes, while components are mature and have well defined role, being less dependent from the rest of the system. Besides that, components are usually formed by more than one class. But as we have seen DSM can easily create a hierarchy, as so it isn't a problem join classes into components. For the sake of simplicity we shall consider that each class will become a component. Other differences between them will have no relevance to this work. Let us now go through the proposed steps:

1. The first step is to construct the PSM - Parameter Size Matrix. This DSM is filled with the values corresponding to the size of the parameters.

PSM			1	2	3	4
Component C	run()	1	0	0	0	0
Component A	foo()	2	**2**	0	0	0
	bar()	3	**5**	0	0	0
Component D	bah()	4	0	**3**	0	0

2. The second step in to construct the RSM – Return Size Matrix. This DSM is filled with the values corresponding to the size of the returns.

RSM			1	2	3	4
Component C	run()	1	0	**4**	**0**	0
Component A	foo()	2	0	0	0	**1**
	bar()	3	0	0	0	0
Component D	bah()	4	0	0	0	0

3. Construct the CFM – Call Frequency Matrix. This DSM is filled with the values corresponding to the resulting interaction of the frequency of calls. Unlike in the first two steps, we cannot simply fill the matrix with the values disposed in the sequence diagram, because if the implementation of 'run()' calls 'foo()' two times and the implementation of 'foo()' calls 'bah()' two times, 'bah()' is called a total of 4 times. Thus, we initially fill the matrix with the values disposed in the sequence diagram. From our example, we obtain:

CFM			1	2	3	4
Component C	run()	1	0	0	0	0
Component A	foo()	2	**1**	0	0	0
	bar()	3	**2**	0	0	0
Component D	bah()	4	0	**2**	0	0

Then we apply the following algorithm to obtain the final Call Frequency Matrix:

1. Take as start point the empty lines of the matrix. These lines reveal the methods that aren't called by any other method.
2. For every empty line check the non-zero elements at the respective column.
3. For each valued element $a_{i,j}$ multiply the column i by a.
4. Apply step 3 and 4 for each column i.

CFM			1	2	3	4
Component C	run()	1	0	0	0	0
Component A	foo()	2	**1**	0	0	0
	bar()	3	**2**	0	0	0
Component D	bah()	4	0	**1x2**	0	0

4. Finally we calculate the DSM that represents the data coupling exposed by the sequence diagram:

$$DSM = RFM \cdot PSM + RFM^T \cdot RSM$$

where $A \cdot B$ is the Hadamard multiplication of the matrix A and B. For our example we obtain:

DSM			1	2	3	4
Component C	run()	1	0	**4**	0	0
Component A	foo()	2	**2**	0	0	0
	bar()	3	**12**	0	0	0
Component D	bah()	4	0	**6**	0	0

Each use case of the system has a sequence diagram to which the steps above must be applied. Thus, each resulting DSM translates the data coupling that the respective use case is responsible for. Summing all these DSMs will result in the final DSM representing the system's design in terms of data coupling. An interesting possibility here is to apply weights to this sum, if it's known that some use case is more frequently executed than other and consecutively plays a more important role in structuring system's data coupling.

Well, since the final DSM is a matrix that represents our system in terms of our data coupling and we want to use data coupling as our similarity measure, we could simply say that our final DSM is exactly our similarity matrix. The only

problem is that our DSM contains information about the direction that data flows and the similarity matrix is symmetric, containing no such information. Summing the amount of data flowing in both directions, we can easily obtain our similarity matrix. This is done by folding the final DSM:

$$SM_{i,j} = SM_{j,i} = DSM_{i,j} + DSM_{j,i}$$

Finnaly, for our example we obtain:

$$SM = \begin{bmatrix} 0 & 18 & 0 \\ 18 & 0 & 6 \\ 0 & 6 & 0 \end{bmatrix}$$

4.3 Clustering Components

We apply now the clustering algorithms we have detailed in sections 2.3.1 and 2.3.2 using the similarity matrix obtained in the previous section as input. Since the values of such matrix are directly related to the amount of data exchanged between our components we expect our clustering algorithms to aggregate these components in a way that the similarity between the resulting clusters is as low as possible. Each resulting cluster will represent a device in a mobile system. Therefore, minimizing the similarity between the resulting clusters means reducing the amount of data exchanged between devices.

The first class of algorithm selected was the agglomerative hierarchical clustering algorithms class. From this class we will apply the Single Link and the Complete Link, which are the most popular in this class. We have chosen to use these algorithms because they produce hierarchical structures, like well designed systems present. Beyond that, these algorithms work over a similarity matrix, where all values are previously calculated and such approach is very suitable to the circumstances that we defined our similarity measure. The data coupling information is retrieved from the provided sequence diagrams and that cannot be done while the algorithm is running, it must be done previously. Moreover this algorithm has low computational costs being able to handle great amount of data – systems with great size – in a relative short time.

The other algorithm selected is based on graph theory, more precisely on Spectral Graph partitioning. It takes as input a weighted graph and cut some of its edges to produce two new sub-graphs. We have chosen this algorithm because it mathematically ensures an optimal partition of the graph. However we are conscious of its computational cost and the limitation of applying it to large scale systems. Moreover, such technique does not demand that we work calculations on the graph itself, but instead over the Laplacian matrix representing it. In section 4.3.2 we show its application in our method and we will see that this fact will turn into an advantage for us and was also one of the reasons we decided to use it.

4.3.1 Single Link and Complete Link

Applying the two hierarchical algorithms is pretty direct once we have the Similarity Matrix. From now on, each line (or column) from the matrix will stand for a cluster. Therefore in the first interaction every component of our system forms a cluster. The algorithm will search for the highest value in the matrix and gather the corresponding clusters into one. That means, two lines (and respective columns)

will be removed and a new one corresponding to the just formed cluster will be added to the matrix. This process will repeat until the matrix has only one element – the final cluster containing all components.

The difference between Single Link and Complete Link, as explained in chapter 2, appears when joining two clusters. The similarities between the new cluster and the remaining clusters must be determined and how that is done differentiates these algorithms. These values are calculated from the removed lines and will correspond to the new line added to the matrix. Suppose we have removed the lines s_1 and s_2. The new line s_3 to be added to the similarity matrix will be calculated so:

- Single Link: s_3 will be formed by the highest values from s_1 and s_2.
- Complete Link: s_3 will be formed by the lowest values from s_1 and s_2.

Each time two clusters are joined, track is kept in a dendrogram which will serve as representation for the nested partitions. The algorithm will in practice start joining the most communicating components. So in the bottom of the dendrogram we are able to identify the most communicating cores, while in the top we have a high level view of the communicating structure of the system.

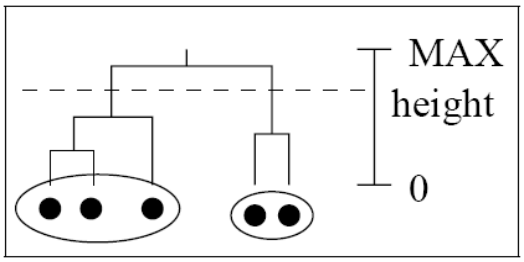


Figure 16 – Sectioning the Dendogram

Sectioning the dendrogram at a given level will result in a single final partition. A question that arrived often in the software clustering literature referred exactly to this problem. How to find the appropriate height at which to cut the dendrogram? This is a difficult problem itself and may there be more than one place to do it [AL99]. Most of the time this question is not answered and the advantages of having a hierarchical structure to analyze are exalted. We propose to measure each level of the dendrogram using the Modularization Quality (MQ), described in section 3.2, and cut the dendrogram where the MQ achieves highest values. This approach will lead us to the partition which has the lowest communicating clusters containing highly communicating components.

Since the example we have been using through this chapter is too small we obtain the same results applying the two algorithms (figure 17). The partition obtained determines that components A and C should be deployed in the same device while component D in another device. If we take a look again at similarity matrix calculated for this system, is pretty easy to assure that this is the most interesting solution. We will present a bigger example in the section 4.4 and we will be able to compare the differences and identify problems.

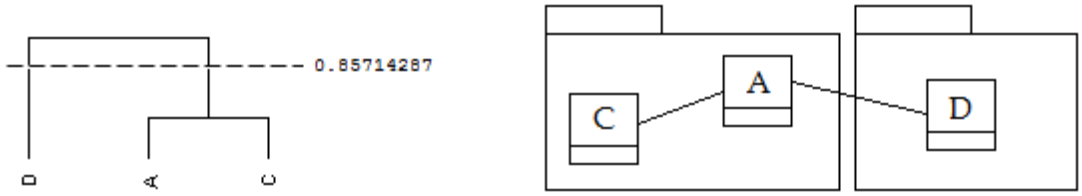


Figure 17 – Single Link and Complete Link Dendogram and Partition

4.3.2 Spectral Graph Partitioning

When applying Spectral Graph Partitioning to software, as said in section 3.3.2, the first step is to represent the system as a weighted graph. In the Unified Modeling Language the general structure of a system is already represented through boxes - denoting classes - and lines connecting these boxes - expressing relations like associations and compositions. Therefore, in relation to this first step there is not much to be done. The second step would be to extract the correspondent Laplacian Matrix of such graph. This is very comfortable for us because the previously obtained SM is exactly the adjacency matrix used to make such calculation. In section 2.3.2 we have seen that the Laplacian Matrix is defined as L = D – A, where the Degree Matrix $D = [d_{ij}]$ is defined as:

$$d_{ij} = \begin{cases} \sum_{k=1}^{n} a_{ik} & ,if\ i = j \\ 0 & ,if\ i \neq j \end{cases}$$

Therefore the Laplacian Matrix obtained from the SM of our example is:

$$L = \begin{bmatrix} 18 & 0 & 0 \\ 0 & 24 & 0 \\ 0 & 0 & 6 \end{bmatrix} - SM = \begin{bmatrix} 18 & -18 & 0 \\ -18 & 24 & -6 \\ 0 & -6 & 6 \end{bmatrix}$$

Calculating now the Fiedler vector of this matrix, we are able to determine the final partition of the system. The components corresponding to the negative values of this vector will form a cluster while the positive values will form another. This technique guarantees that the total weight of the cut edges is the lowest possible. Since the edges of such graph denote the amount of data exchanged between components, we can be sure of having achieved the best possible partition for our objectives. The shortcoming of this approach, as known, is the computational cost of calculating the Fiedler vector. This is not a problem for small systems, but it can become unfeasible for large scale systems.

$$v_2^T = [-0.5199416 \quad -0.2852315 \quad 0.8051731]$$

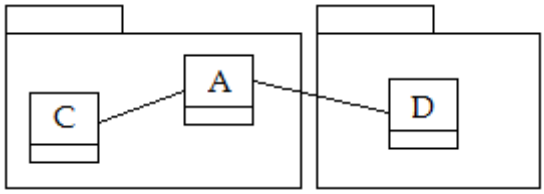


Figure 18 – Spectral Graph Partition

In the figure above we have the calculated Fiedler vector and the partition determined by its values. The resulting partition, as expected, is the same as the one obtained with the hierarchical algorithms.

In the following section we apply these two classes from algorithms to a system containing more classes. It is obviously not the scale of systems we are aiming but it is enough to make our proposal clearer and easier to analyze.

4.4 Study Case

In this section we apply the method we have proposed in this work to a system with nine classes. This is still a very small system comparing to the real mobile systems that our method intends to handle, but it will give us the opportunity to analyze the proposed method and take our conclusions about it. Below is depicted the system's general structure:

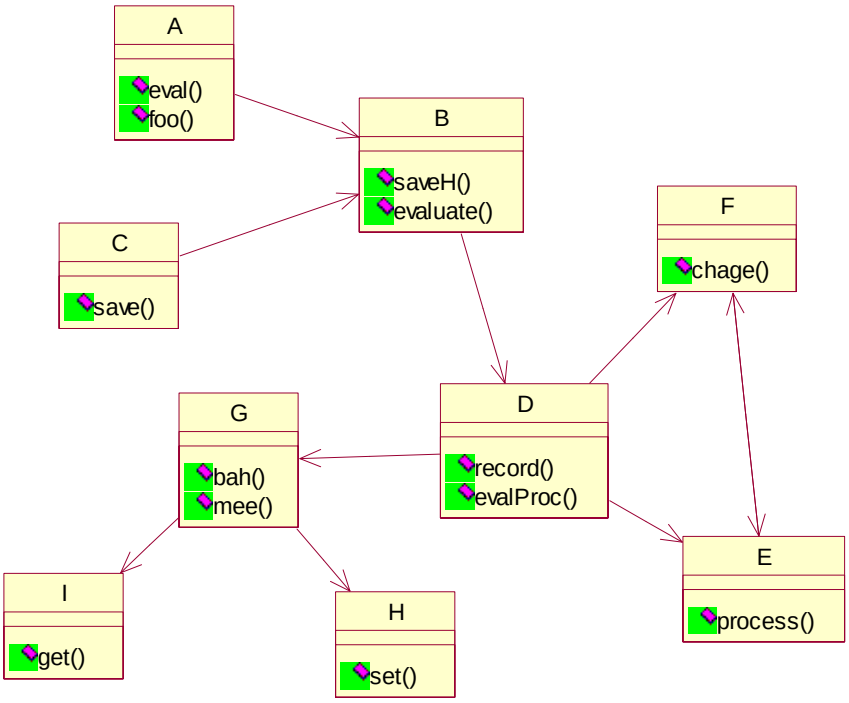


Figure 19 – Class Diagram

For this system we will consider two use cases. The first use case is called "Eval" and the second one "Save". We shall provide for each use case richer sequence diagrams, which contains more specific information about the interaction between components that we have presented before.

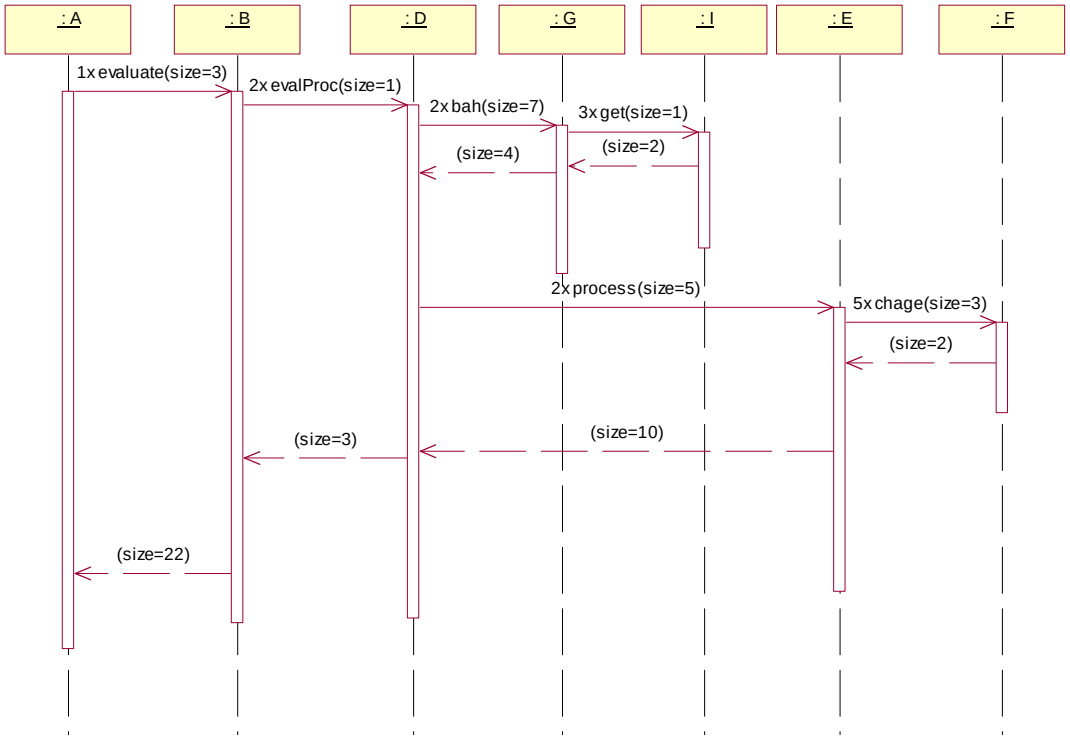


Figure 20 – Richer Sequence Diagram for Use Case „Eval“

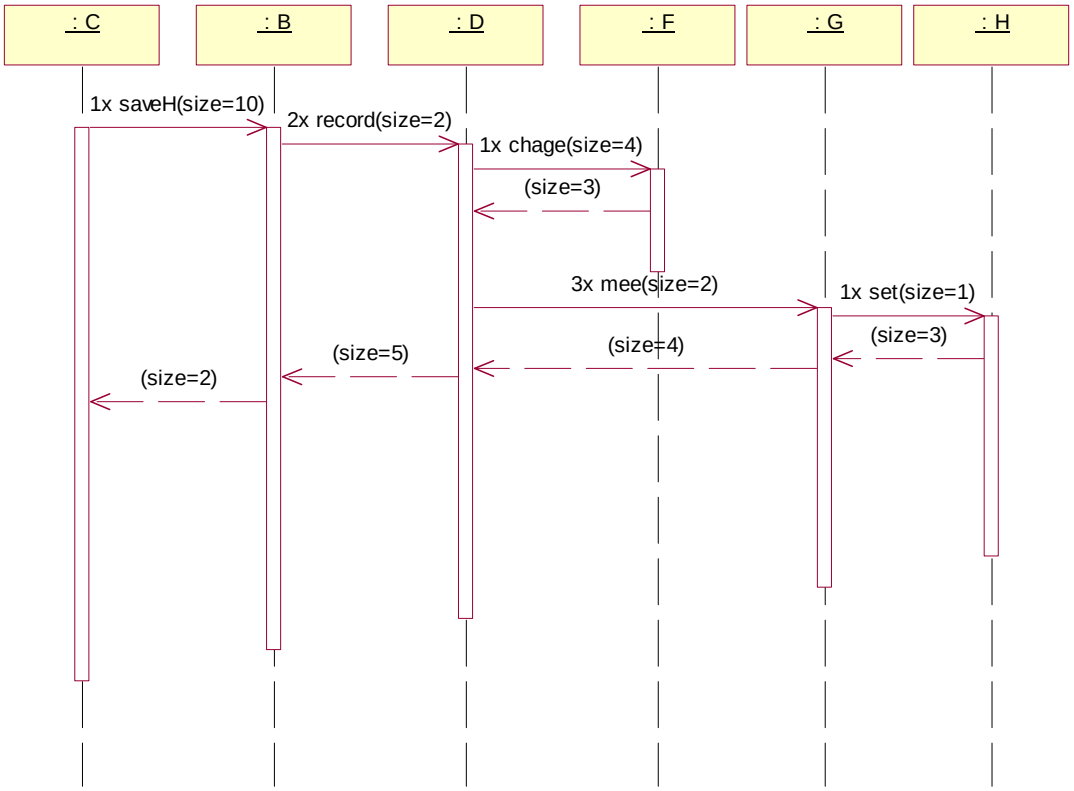


Figure 21 – Richer Sequence Diagram for Use Case „Save“

As proposed, we calculate now the DSM representing the system in term of data coupling for each given sequence diagram.

DSM - Eval		1	2	3	4	5	6	7	8	9	10	11	12	13
A eval()	1	0	0	0	22	0	0	0	0	0	0	0	0	0
foo()	2	0	0	0	0	0	0	0	0	0	0	0	0	0
B saveH()	3	0	0	0	0	0	0	0	0	0	0	0	0	0
evaluate()	4	3	0	0	0	0	0	6	0	0	0	0	0	0
C save()	5	0	0	0	0	0	0	0	0	0	0	0	0	0
D record()	6	0	0	0	0	0	0	0	0	0	0	0	0	0
evalProc()	7	0	0	0	2	0	0	0	40	0	16	0	0	0
E process()	8	0	0	0	0	0	0	20	0	40	0	0	0	0
F change()	9	0	0	0	0	0	0	0	60	0	0	0	0	0
G bah()	10	0	0	0	0	0	0	28	0	0	0	0	0	24
mee()	11	0	0	0	0	0	0	0	0	0	0	0	0	0
H set()	12	0	0	0	0	0	0	0	0	0	0	0	0	0
I get()	13	0	0	0	0	0	0	0	0	0	12	0	0	0

DSM - Save		1	2	3	4	5	6	7	8	9	10	11	12	13
A eval()	1	0	0	0	0	0	0	0	0	0	0	0	0	0
foo()	2	0	0	0	0	0	0	0	0	0	0	0	0	0
B saveH()	3	0	0	0	0	10	10	0	0	0	0	0	0	0
evaluate()	4	0	0	0	0	0	0	0	0	0	0	0	0	0
C save()	5	0	0	2	0	0	0	0	0	0	0	0	0	0
D record()	6	0	0	4	0	0	0	0	0	6	0	24	0	0
evalProc()	7	0	0	0	0	0	0	0	0	0	0	0	0	0
E process()	8	0	0	0	0	0	0	0	0	0	0	0	0	0
F change()	9	0	0	0	0	0	8	0	0	0	0	0	0	0
G bah()	10	0	0	0	0	0	0	0	0	0	0	0	0	0
mee()	11	0	0	0	0	0	12	0	0	0	0	0	18	0
H set()	12	0	0	0	0	0	0	0	0	0	0	6	0	0
I get()	13	0	0	0	0	0	0	0	0	0	0	0	0	0

To achieve the following Similarity Matrix we considered that both use cases have the same participation in the system. That is, they are equally requested by external actors. Thus, we sum both DSMs obtained without weights and get:

SM =

A	B	C	D	E	F	G	H	I	
0	25	0	0	0	0	0	0	0	A
25	0	12	26	0	0	0	0	0	B
0	12	0	0	0	0	0	0	0	C
0	26	0	0	60	14	80	0	0	D
0	0	0	60	0	100	0	0	0	E
0	0	0	14	100	0	0	0	0	F
0	0	0	80	0	0	0	24	36	G
0	0	0	0	0	0	24	0	0	H
0	0	0	0	0	0	36	0	0	I

This matrix translates the communication aspect of our system and is the start point for both algorithm categories we proposed to use. Just a quick look at it is enough to say that the system presents dense communication in the system's middle modules D, E, F and G. We expect from our algorithms at least that these modules are kept together in the final partitions.

4.4.1 Hierarchical Algorithms

We first applied the Single Link algorithm to the Similarity matrix just obtained. The resulting dendrogram, with Modularity Quality calculated for each partition can be seen in the figure below:

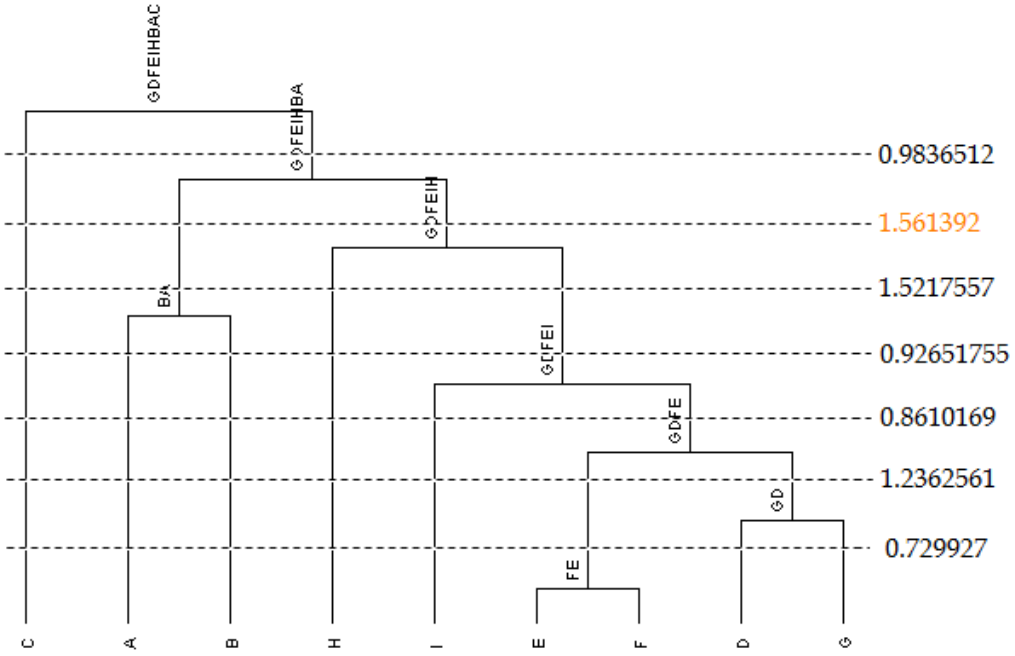


Figure 22 – Single Link Dendogram

The results determine the following partition:

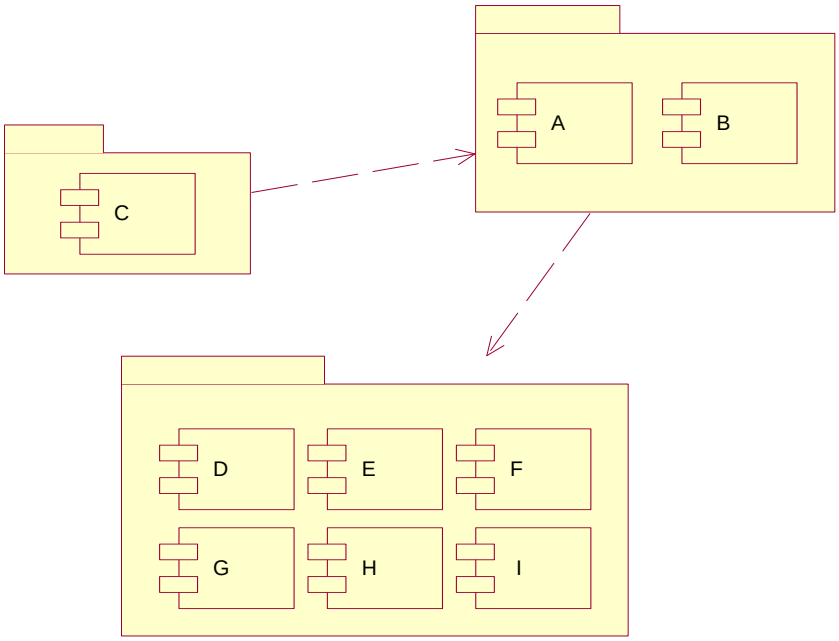


Figure 23 – Single Link Partition

The application of the Complete Link will results in the following dendrogram, also with Modularization Quality calculated for each partition:

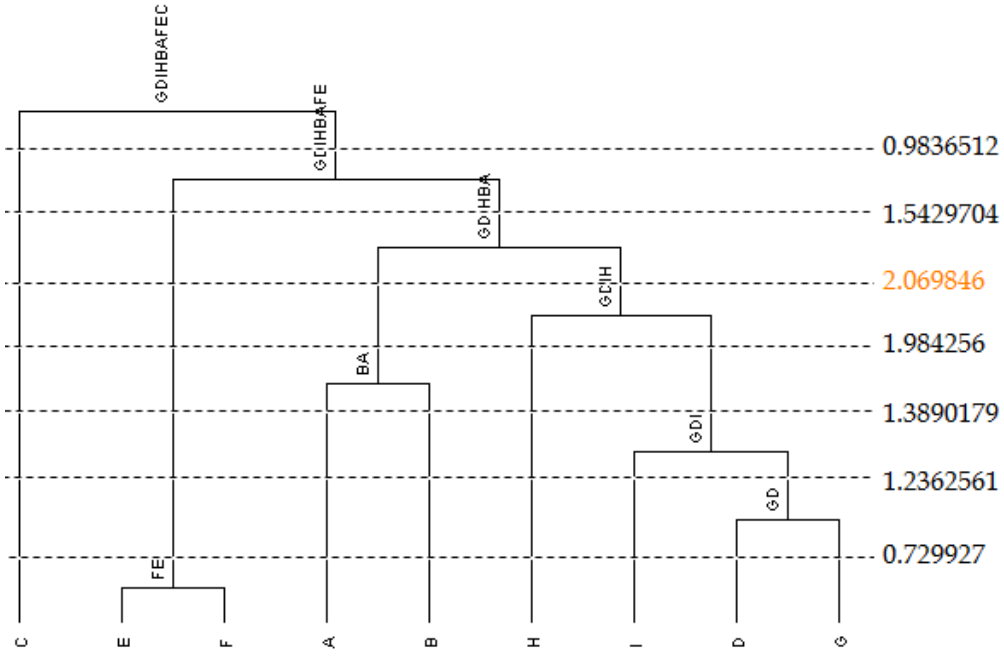


Figure 24 – Complete Link Dendogram

The highest value in the dendrogram suggests the following partition:

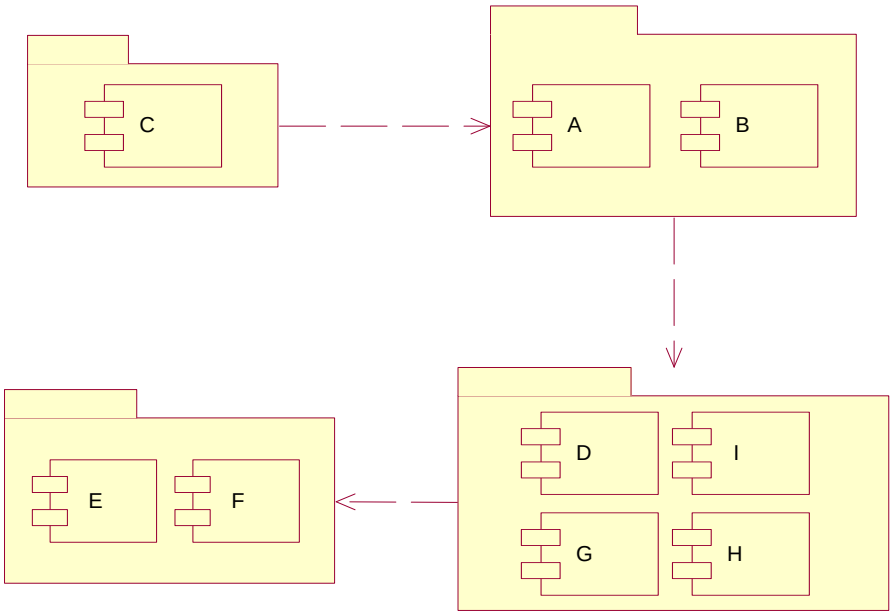


Figure 25 – Complete Link Partition

The first thing to be noticed is that the resulting number of clusters, as said before, is not under our control. The Single Link resulted in three clusters, while the Complete Link resulted in four. Of course, when deploying a system the number of devices is fixed and so we must execute some pos-process in order to achieve the same number of clusters as devices available. This can be better done by a specialist analyzing the similarity of the system. For example, if the components of this system must be distributed among three devices, we can say, just looking at the similarity matrix of the current configuration (matrix bellow), that the cluster containing components E and F should be merged with the cluster containing components D,I,G and H, since they are the strongest communicating clusters.

SM* =	AB	C	EF	DGHI	
	0	12	0	26	AB
	12	0	0	0	C
	0	0	0	74	EF
	26	0	74	0	DGHI

This will bring us to the same result as Single Link did - leading us to another important observation. Complete Link achieved higher Modularization Quality values than Single Link. The reason for that can be found in Complete Link's property of forming more compact clusters than Single Link. That means, Complete Link finds dense clusters in data, reflecting readily the relation of elements within clusters, while Single Link doesn't consider much the cohesion of resulting groups. If we remember that Bunch's Modularization Quality takes into consideration the trade-off between intra-connectivity and inter-connectivity, it is expected that Complete Link achieves higher values than Single Link.

Since the relations between clusters have much more influence in the Single Link's results as they do in Complete Link, we could say that Single Link is the most appropriate choice for our objectives. However taking into consideration that some pos-process must be done to make the number of clusters match the number of devices and also that other aspects of the deployment specification must be fulfilled, we think it's more interesting to identify the most dense communicating clusters and dispose such information the specialist.

4.4.2 Spectral Graph Partitioning

The spectral Graph Partitioning, as said before, is supposed to give us the optimal partition of the system according to our objectives. Unfortunately, it has a high computational cost and for certain system scales it becomes unfeasible to apply it.

Taking as start point the similarity matrix obtained previously from the system, we calculate its Laplacian Matrix:

$$LM = \begin{bmatrix} 25 & -25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -25 & 63 & -12 & -26 & 0 & 0 & 0 & 0 & 0 \\ 0 & -12 & 12 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -26 & 0 & 180 & -60 & -14 & -80 & 0 & 0 \\ 0 & 0 & 0 & -60 & 160 & -100 & 0 & 0 & 0 \\ 0 & 0 & 0 & -14 & -100 & 114 & 0 & 0 & 0 \\ 0 & 0 & 0 & -80 & 0 & 0 & 140 & -24 & -36 \\ 0 & 0 & 0 & 0 & 0 & 0 & -24 & 24 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -36 & 0 & 36 \end{bmatrix}$$

From this Laplacian Matrix we calculate its Fiedler vector:

$$v_2 = \begin{bmatrix} 0.2575765 \\ 0.1587510 \\ 0.7910709 \\ -0.1044579 \\ -0.1403646 \\ -0.1404450 \\ -0.2020707 \\ -0.3365942 \\ 0.2754660 \end{bmatrix}$$

As said before, the values' signs of the Fiedler vector indicate which components should be placed together. In the following figure we can see the disposal of the components:

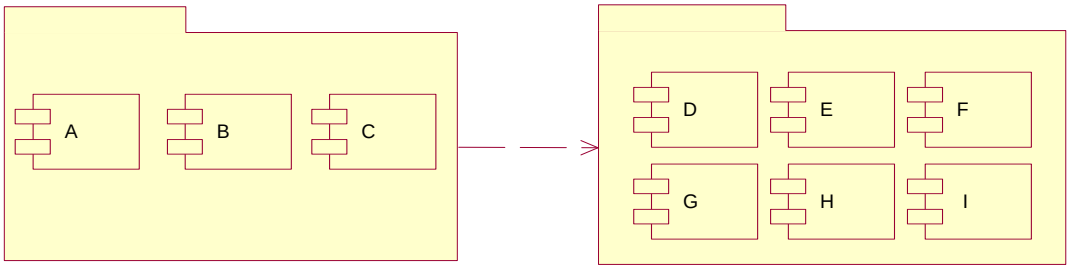


Figure 26 – Spectral Graph Partition

Although we thought this method would result in the optimal partition, we can observe in this example that such method has a special condition. We can be sure that the edge that will be cut has the lowest weight in the system. However, cutting the edge must result in two sub-graphs. If we take a look at the similarity matrix again we will see that the lowest edge is the one that connects components C and B. Back in figure 19 though, we can see that cutting this edge will not result in two sub-graphs but instead in a graph without one component. Because one component doesn't form a sub-graph, this edge is not a candidate to be cut. Therefore, this method just considers the possibility of deploying one component alone in a device when there is no other choice (example in section 4.3.2). Another point to be remembered is the number of devices. As we said, we can re-apply the same method to the resulting clusters until the number of clusters is equal to the number of devices.

5. Conclusion

This work has proposed a method to determine the distribution of software system's components among computing nodes in a mobile context. In order to determine this partitioning, this method considers the data load exchanged between such components. In the context of mobile systems, this can be of great use since data is transmitted over narrow-banded channels. Therefore, we can maximize quality of service minimizing the amount of data transmitted over these channels.

We have taken as start point UML sequence diagrams, since they expose the relationship among components. However, during investigation of the different relationship between components, we have seen that such diagrams do not contain information enough to our objective. Thus, we have defined richer sequence diagrams from which we could have a more precise estimation of the amount of data flowing through the system.

We also have made a research in the software modularization literature so that we could take advantage of numerical methods developed to achieve modularity. In special we have found a subfield called software clustering, which consists of applying clustering techniques to software. The works we found in software clustering came to show great suitability to our intentions and we have decided to make use of clustering techniques.

Our method includes several transformations in order that the systems can be represented in terms of data load exchange. Such transformations were considerably convenient for the algorithms we have chosen and made visualization much easier. Clustering has proved, as expected, very adequate to this problem, but due to the vast extension of clustering techniques, further work must be done to find the most appropriate technique.

We expect this work to help in the evolution of tools and processes directed to development of mobile systems. The evolution of such systems, and consequently of the greater scenario in which these are embraced, is severely compromised by communication channels and proposal like this represent major steps to overcome such problems.

References

[Par72] PARNAS, D. L.: On the Criteria to Be Used in Decomposing Systems into Modules. In: *Communications of the ACM*, Vol.15, Nr.12, pages 1053-1058, 1972.

[Fie75] Fiedler, M.: A property of eigenvectors of non-negative symmetric matrices and its applications to graph theory. In: *Czechoslovak Mathematical Journal*, 25(100), pages 619-633. 1975.

[HB85] Hutschens, David H.; Basili, Victor R.: System Structure Analysis: Clustering with Data Bindings. In: IEEE Transactions on Software Engineering, vol. SE-11, Nr. 8, 1985.

[Sch91] SCHWANKE, Robert W.: An Intelligent Tool for Re-engineering Software Modularity. In: *Proceedings of the 13th International Conference on Software Engineering*, pages 83-92, 1991.

[MOTU92] Müller, H.; Orgun, M.; Tilley, S.; Uhl, J.: Discovering and reconstructing subsystem structures through reverse engineering. In: Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, 1992.

[Boo94] Booch, G.: Object-Oriented Analysis and Design. Redwood City, CA.: Benjamin/Cumming.

[MWT94] MÜLLER, Hausi A.; WONG, Kenny; Tilley, Scott R.: Understanding Software Systems Using Reverse Engineering Technology. In: *Colloquium on Object Orientation in Databases and Software Engineering; The 62nd Congress of L'Association Canadienne Francaise pour L'Avancement des Sciences (ACFAS)*. 1994.

[Che95] Chen, Y.: Reverse Engineering. In: *Practical Reusable Unix Software*, chapter 6, pp.177-208, 1995.

[BMC96] Burd, E.; Munro, M.;Wezeman, C.: Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity. In: Working Conference on Reverse Engineering, pages 189-196, 1996.

[Lak96] Lakos, J.: Large-scale C++ software design. Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA. 1996.

[Mar96] Martin, R. C.: Granularity. In: C++ Report 8(10), pages 57-62, 1996.

[CGK97] Chen, Y.; Gansner, E.; Koutsofios, E.: A C++ Data Model Suporting Reachability Analysis and Dead Code Detection. In: *Proc. Sixth*

European Software Eng. Conf. And Fifth ACM SIGSOFT Sysmp. Foundations of Software Eng., 1997.

[Wig97] WIGGERTS, T. A.: Using Clustering Algorithms in Legacy Systems Remodularization. In: *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'07).* 1997.

[MMRCG98] MANCORIDIS, S.; MITCHELL, B. S.; Rorres, C.; Chen, Y.; Gansner, E. R.: Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: *Proceedings 6th Intl. Workshop on Program Comprehension*, 1998.

[AL99] Anquetil, N.; Lethbridge, T. C.: Experiments with Clustering as a Software Remodularization Method. In: Proceedings of the Sixth Working Conference on Reverse Engineering, page 235, 1999.

[DMM99] Doval, D.; Mancoridis, S.; Mitchell, B. S.: Automatic Clustering of Software Systems using a Genetic Algorithm. In: *Proceedings of the Software Technology and Engineering Practice*, page 73, 1999.

[JMF99] Jain, A.K.; Murty, M. N.; Flynn, P.J.: Data Clustering: A Review. In: ACM Computing Surveys, Vol. 31, Nr. 3, 1999.

[KCK99] Korn, J; Chen, Y; Koutsofios,E.: Chava: Reverse Engineering and Tracking of Java Applets. In: *Proc. Working Conf. Reverse Eng.*,1999.

[MMCG99] MANCORIDIS, S.; MITCHELL, C.; Chen, Y.; Gansner, E. R.: Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In: *Proceedings of the IEEE International Conference on Software Maintenance*, page 50, 1999.

[DB00] Davey, J.; Burd, E.: Evaluating the Suitability of Data Clustering for Software Remodularization. In: *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. 2000.

[MNS01] Murphy, G. C.; Notkin, D.; Sullivan, K. J.: Software Reflexion Models: Bridging the Gap between Design and Implementation. In: IEEE Transactions on Software Engineering. Vol. 27, Nr. 4. 2001.

[SGCH01] Sullivan, K. J.; Griswold, W. G.; Cai, Y.; Hallen, B.: The Structure and Value of Modularity in Software Design. In: ACM SIGSOFT Software Engineering Notes. Vol. 26, Issue 5. 2001.

[Hau02] Hautus, E.: Improving Java Software Through Package Structure Analysis. In: *The 6th IASTED International Conference Software Engineering and Applications.* 2002.

[Webb02] Clustering Chapter In: Webb, A. R.: Statistical Pattern Recognition, Second Edition. John Wiley & Sons, 2002, pages 361-406. ISBN 0-470-84513-9.

[TK03] Theodoridis, S.; Koutroumbas, K.: Pattern Recognition, Second Edition. Elsevier, 2003. ISBN 0-12-685875-6.

[ABA04] Al-Otaiby, T. N.; Bond, W. P.; AlSherif, M.: Software Modularization using Requirements Attributes. In: *Proceedings of the 42nd annual Southeast regional conference (ACMSE 42)*. 2004.

[AAB05] Al-Otaiby, T. N.; AlSherif, M.; Bond, W. P.: Toward Software Requirements Modularization using hierarchical clustering techniques. In: *Proceedings of the 43rd annual southeast regional conference (ACMSE 43)*. 2005.

[CS05] Cai, Y.; Sullivan, K. J.: Simon: Modeling and Analysis of Design Space Structures. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering (ASE'05)*. 2005.

[DLP05] Ducasse, S.; Lanza, M.; Ponisio, L.: Butterflies: A visual approach to characterize Packages. In: *11th IEEE International Metrics Symposium (METRICS 2005)*. 2005.

[PB05] Parsa, S.; Bushehrian, O.: The Design and Implementation of a Framework for Automatic Modularization of Software Systems. In: *The journal of Supercomputing*, Vol. 32, Nr. 1. 2005.

[SBBP05] Seng, O.; Bauer, M.; Biehl, M.; Pache, G.: Search-based Improvement of Subsystem Decompositions. In: *Proceedings of the 2005 conference on Genetic and Evolutionary computation (GECCO'05)*. 2005.

[SJSJ05] Sangal, N.; Jordan, E.; Sinha, V.; Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages and applications*. 2005.

[CTS06] Chatzigeorgiou, A.; Tsantalis, N.; Stephanides, G.: Application of Graph Theory to OO Software Engineering. In: *Proceedings the 2006 international workshop on Workshop on interdisciplinary software engineering research (WISER'06)*. 2006.

[MM06] Mitchell, B. S.; Mancoridis, S.: On the Automatic Modularization of Software Systems Using the Bunch Tool. In: *IEEE Transactions on Software Engineering*, Vol. 32, Nr. 3. 2006.

[Wan06] Wang, H.: Nearest Neighbor by Neighborhood Counting. In: *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. 28, Nr. 6, pages 942-953, 2006.

[ACoM07] Garcia, A.; Greenwood, P.; Heineman, G.; Walker, R.; Cai, Y.; Yang, H. Y.; Baniassad, E.; Lopes, C. V.; Schwanninger, C.; Zhao, J.: Assessment of Contemporary Modularization Techniques –

AcoM'07: Workshop report. In: *ACM SIGSOFT Software Engineering Notes*, page 31, volume 32, number 5. 2007.

[HC07] Huynh, S.; Cai, Y.: An Evolutionary Approach to Software Modularity Analysis. In: *First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)*. 2007.

[MT07a] Melton, H; Tempero, E.: The CRSS Metric for Package Design Quality. In: *Proceedings of the thirtieth Australasian Conference on Computer Science*, pages 201-210, 2007.

[MT07b] Melton, H.; Tempero, E.: Towards assessing modularity. In: *IEEE First International Workshop on assessment of Contemporary Modularization Techniques*, 2007

[YR07a] Yu, LG; Ramaswamy, S.: Verifying Design Modularity, Hierarchy, and Interaction Locality Using Data Clustering Techniques. In: *Proceedings of the 45th annual southeast regional conference (ACMSE)*. 2007.

[YR07b] Yu, LG; Ramaswamy, S.: Component Dependency in Object-Oriented Software. In: *Journal of Computer Science and Technology*, 20(3), pages 379-386, 2007.

[Scilab08] INRIA. Scilab: The open source plataform for numerical computation. www.scilab.org (10.06.08).

Appendix I – Software Modularization Approaches

Approach		Granularity	Description	Works
Modularity Principles			Proposition of principles to achieve modularity	[Par72] [Mar96][MT07b]
Software Clustering	Bottom-Up	Code	Software re-modularization	[AL99] [DB00] [Wig97] [Sch91] [MNT94] [HB85]
		Classes	Module Dependency Graph	[MMRCG98] [DMM99] [MMCG99] [MM06] [SBBP05] [PB05]
	Top-down		Design/Implementation Conformance	[MNS01] [HC07]
Package Properties Analysis		Packages	Evaluation based on metrics	[MT07a] [Hau02] [DLP05] [SJSJ05]
New programming paradigms			Aspect-Oriented Programming / Feature-Oriented Programming	[Acom07]*

* In this workshop report can be found references to the most recent works following this approach.