Published in L'Objet, vol. 7, nº4, 2001,
Hermes Publisher, France

A Merit Factor Driven Approach to the Modularization of Object-Oriented Systems

Fernando Brito e Abreu[*] — Miguel Goulão[*]

[*] *Software Engineering Group*
FCT/UNL and INESC
Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
{fba, miguel.goulao}@inesc.pt

ABSTRACT: *Modularization has many designations in the object-oriented world although it usually corresponds to system partitioning in groups (clusters) of classes. The problem is that this partitioning can be accomplished in many different ways, with two degrees of freedom: the number of clusters and the way to group classes to build them. We use Cluster Analysis to derive the optimal grouping for a fixed number of clusters. To find the optimal number of clusters we propose a decision criterion based on the definition of a modularization merit factor. We use this theoretical framework in the MOTTO tool and apply it to a reasonably large sample of software systems. The most important results are that (i) the proposed approach is feasible and easy to apply with appropriate tool support and that (ii) analyzed systems are far from fully exploiting the benefits of modularization.*

RÉSUMÉ: *La modularisation peut avoir plusieurs sens dans le monde de l'orienté objet, même si elle correspond habituellement une division d'un système en groupes de classes. Le problème est que cette division peut être accomplie de différentes manières enjouant sur le nombre de groupes et la façon de regrouper les classes. Nous utilisons une méthode de regroupement automatique pour déterminer le regroupement optimal pour un nombre fixe de groupes. Pour trouver le nombre optimal de groupes, nous proposons un critère de décision basé sur la définition d'un facteur de mérite de la modularisation. Nous utilisons ce cadre théorique dans l'outil de MOTTO et nous l'appliquons à un échantillon raisonnablement grand de systèmes logiciels. Les résultats les plus importants sont que (i) l'approche proposée est faisable et est facile à appliquer avec le support d'un outil approprié et que (ii) les systèmes analysés sont loin d'exploiter pleinement les avantages de la modularisation*

KEY WORDS: *Modularity of object-oriented systems, cluster analysis, modularity assessment, MOTTO tool*

MOTS-CLÉS: *Modularité des systèmes à objets, regroupement automatique, évaluation de la modularité, l'outil MOTTO*

1. Introduction

Modularity is an essential aspect in all engineering domains. It allows, among other things: (i) designing and developing different parts of the same system by different people, often belonging to distinct organizations, (ii) handling the complexity of large systems by splitting loosely coupled parts that can be better understood individually, (iii) testing systems in a parallel fashion (different people simultaneously), (iv) substituting or repairing defective parts of a system without interfering with other parts, (v) reusing existing parts in different contexts, (vi) dividing the system in configuration units to be put under configuration control and (vii) restricting defect propagation.

The architecture of a software system is determined, at the more abstract level, by a set of modules and by the way they are glued together [SCH 96, pp.9]. Generically, a module can be an aggregate of algorithms' implementations and data structures that interact somehow to deliver a given kind of functionality. Each module can have its own state, shared or not, and need the collaboration of other modules to deliver its functionality. A module should have a clear interface. A protocol should be offered to other modules, by means of some exporting mechanism that makes its interface available. It is desirable, for reasons thoroughly discussed in the Software Engineering literature, that modules are highly cohesive and loosely coupled [GHE 91, JAC 92, PRE 00, SOM 00].

If we consider a module to be, as we will do henceforth, a set of classes, then the intra-modular coupling, that is, the coupling among the classes belonging to the module, can represent **module cohesion**. Complementarily, when we talk about **module coupling** we mean the inter-modular couplings, that is, the ones that cross the module borders. These may correspond to dependencies of internal classes (those belonging to the module) on external ones, or the other way round. Modularization can be flat or hierarchical (i.e. modules containing other modules). An objective criterion for modularization should be made explicit in the software documentation. On hierarchical modularization that criterion can be, different at each modularization level.

Modularity is an internal quality characteristic that influences external software quality characteristics, as suggested in [ISO 9126], and it can be observed at different levels of abstraction [CON 90]. During **requirements specification** and **detailed analysis**, modules are usually black-boxes that facilitate the dialog and understanding between domain experts and analysts. At **design level**, modularity is traditionally associated to the identification of subsystems and abstract data types [EMB 88]. Software **components**, which are usually built as an encapsulated set of interrelated classes, can also be seen as reusable modules [SZY 98]. At **source code level**, modules usually correspond to operating system files, allowing separate compilation and favoring incremental development. That is why these modules are often called compilation units. At **executable code level**, modularity also plays an important role, as with overlays or

dynamic linked libraries. In this paper we will be mainly concerned with the design level for object-oriented systems.

The object-oriented paradigm, along with the spreading availability of processing power, has allowed the conception of increasingly large and complex software systems. These must be developed and integrated modularly. Although the need to aggregate classes seems to be consensual, there is, however, a lack of terminological uniformity in the designation of those aggregates or clusters, as we will further see.

In the *C++* programming language, **namespaces** support modularization by providing a mechanism for expressing logical grouping [STR 97]. **Packages** are an important modularization mechanism in *Java*. They may contain any combination of interfaces (defining types) and implementations (classes) [GOS 96]. In *Smalltalk* development environments, such as *Envy*, there is modularization support also through the use of **packages**. During runtime, those packages are loaded in a specific order, starting by the kernel one, which defines all the primitives. In the *Delphi* language, an extension of *Pascal* for object-oriented programming, modules are called **units** [CAN 96].

Bertrand Meyer, the creator of the *Eiffel* language, empathizes a modularization abstraction, the **cluster**, which is the basis of his *Cluster Model* [MEY 95]. In *OMT* (Object Modeling Technique) the modularization unit is called **subsystem** [RUM 91]. In *Objectory* the same denomination is used [JAC 92]. Grady Booch proposes the word **category** in his method [BOO 94]. Meilir Page-Jones talks about **domains** and **sub-domains** [PAG 95]. The word **package** is again used in *UML* (Unified Modeling Language) [BOO 97] and in the *Catalysis* approach, where it designates any container of developed artifacts [SOU 98, pp.18]. This kind of modules can contain classes, component specifications, component implementations, reusable frameworks, nested packages (hierarchical modularization) and other deliverable types.

In the above references we could only find qualitative indications on the need of modularization or, at most, some vague guidelines for grouping classes. We will show that the determination of an optimal modularization solution is feasible. This kind of information is obviously useful during the initial design phase. On the other hand, large software systems usually evolved incrementally from smaller ones. Although the initial architecture may have been acceptable, the evolution throughout time often causes **modularity degradation**, especially if the underlying criterion was not clearly enforced. At a certain point in time a modularity reengineering action will be required. Being able to assess the need for such an action and to point out the optimal solution, were some of the driving forces for the work presented herein.

This paper is organized as follows. Section 2 introduces the problems faced in a quantitative approach to software modularization and proposes a theoretical framework to support it. Section 3 describes the methodological approach adopted in a large scale experiment using the proposed framework. In section 4 the data collected in the experiment is analyzed and discussed. Related and future works are identified in the last two sections.

2. The Quantitative Way

2.1. How much modularity?

Ivar Jacobson mentions that an object module usually has 1 to 5 classes, although he reports on a system developed with *Objectory* where he had as much as 17 classes [JAC 92, pp.145]. On the other hand, Bertrand Meyer advocates that software systems should be divided in modules (clusters), typically with 5 to 40 classes each and developed by 1 to 4 people [MEY 95]. Still according to him, that dimension should be such that the cluster would play an important role in the system, but not too big that hinders its understanding by just one person after some effort. Here, as in most references on the subject, the brief quantitative citations have no supporting evidence. In a well-known reference book, Meyer recognizes that the criteria, rules and principles of software modularity are usually introduced only through qualitative definitions, although some of them may be amenable to quantitative analysis [MEY 97, pp.65]. We hope to prove that in this paper.

The question then is: given a system, to what degree should you decompose it, that is, how many modules should you consider? This question leads to the following one: how disparate in size should the modules in a given system be?

There is not an answer for these questions unless we define a quantitative criterion for the system sectioning. It is acceptable that the number of modules should somehow be proportional to the system size (e.g. expressed in number of classes). However, such an assertion is not of a great help since neither the proportionality constant is known, nor is it likely that the module density (classes per module) should be uniform.

At the coding level we often find the classes completely separate (e.g. one .h and one .cpp file per class in C++). If we consider the class to be our atomic part on what modularity is concerned, this situation can be seen as the radical expression of the following criterion to define the optimal modularization size:

Strawman Criterion A – The number of modules should be the maximum allowed.

Since the idea of modularization is that of grouping related items (and obviously splitting the unrelated ones), criterion A is not acceptable. On the other hand, if we bear on the Software Engineering literature, we could easily be tempted to define another criterion:

Strawman Criterion B – The number of modules should be the one that maximizes the coupling among classes within each module and minimizes the coupling for those belonging to different modules.

Although appealing, this assertion is nothing but a fallacy: the above number is **always 1** (one module). Indeed, for a given system, starting with the extreme situation of one class per module and whatever aggregation sequence is adopted, when we decrease

the number of clusters by grouping classes, the coupling between the classes within each module increases monotonically. Meanwhile the coupling among classes belonging to different modules decreases also monotonically (in the limit, with only one module, it equals zero)! Criterion B is therefore opposing the idea of modularization itself.

2.2. How to aggregate classes in modules?

When the number of modules is fixed we get to the problem of finding the optimal grouping of classes. In the *Catalysis* approach the authors recognize that it is important to restrict the module (package) dependencies [SOU 98]. Again we are in presence of another qualitative statement. The authors do not explain how to achieve that reduction!

The grouping problem is the aim of *Cluster Analysis*, a subject concerned with the classification of similar items into groups [KAU 90, ROM 90]. Cluster Analysis techniques have been used in many areas such as economics, geology, botany, pattern matching and so on. The objective of clustering techniques is the grouping of items in such a way that the relations between items in the same group are stronger than the relations to items in other groups.

In order to cluster a group of items, two things are required: some measure of the way each item relates to the others and a method to group them. The expression of "how far" two items are, is known as the **dissimilarity** or **distance** between them. Dissimilarities can be obtained in several ways. They are often based on the values of variables that represent certain item properties. The dissimilarities between each pair of classes are usually summarized in a dissimilarities matrix, a square symmetric matrix with zeros on the main diagonal.

Besides a distance measure, one also needs an algorithm to drive the clustering process, a **clustering method**. We have used seven well-known hierarchical agglomerative clustering methods: *Single linkage, Complete linkage, Between groups linkage, Within groups linkage, Centroid, Median* and *Ward's methods* [KAU 90]. Agglomerative methods start with all items in separate clusters and proceed in iterations joining them until the defined number of clusters is reached. In Table 1 we provide a brief description of the clustering methods we use.

For all the previous methods, the clusters to be merged in each stage are the ones whose distance (as defined in the second column of Table 1) is the shortest. We also used the Ward clustering method, which is somewhat different from the others. It departs from the consideration that at each stage, the loss of information that results from the grouping of individuals into clusters can be measured by the total sum of squared deviations of every point from the mean of the cluster to which it belongs. So, this method calculates, for each case, the squared Euclidean distance to the cluster means; these distances are summed for all the cases; at each step, the two clusters that result in the smallest increase in the overall sum of the squared within-cluster distance are combined;

Method name	The distance between two groups is …
Single linkage (nearest neighbor)	… the distance between their closest members.
Complete linkage (furthest neighbor)	… the distance between their most remote pair of items (opposite of single linkage).
Between-groups linkage (group average)	… the average of the distances between all pairs of individuals in the two groups.
Within-groups linkage	… the average of the distances between all pairs of cases in the cluster that would result if they were combined.
Centroid	… the distance between the group centroids, that is the distance between their means, for all of the items.
Median	… similar to the centroid but it weights equally the two groups to combine, in the calculation of the centroid.

Table 1. *Clustering methods*

Due to the nature of our problem – clustering classes in modules – we have to identify how to express the dissimilarities among the classes. Although the solution to this problem has been pointed out long ago, based upon the cohesion (intra-modular coupling) and inter-modular coupling [STE 74], we have not yet seen it applied elsewhere in a quantitative way to object-oriented systems by using Cluster Analysis. Therefore we state the following modularization criterion:

Woodenman Criterion C – Given a constant number of classes and modules, the aggregation of classes should be the one that maximizes the coupling among classes within each module and minimizes it for classes belonging to different modules.

The coupling between two classes can be characterized by the number of coupling instances discriminated by type. Several authors have proposed coupling taxonomies in the realm of the OO paradigm [BRI 99, HIT 96, LOU 97, POE 98]. In [ABR 00] we have introduced another taxonomy with *GOODLY* in mind. *GOODLY* (a **G**eneric **O**bject **O**riented **D**esign **L**anguage? **Y**es!) is an intermediate formalism used to specify the design of systems built according to the object oriented paradigm [ABR 99]. The used taxonomy includes the following coupling categories: *Direct Inheritance (DI), Class Parameter (CP), Attribute Type (AT), Employed Attribute (EA), Parameter in Operation (PO), Parameter in Message (PM), Parameter in Call (PC), Return in Operation (RO), Return in Message (RM), Return in Call(RC), Local Attribute in Operation (LA)* and *Message Recipient (MR).*

The more coupling instances there are between two classes, the stronger is their interconnection strength, which we call **affinity**. We hypothesize that different coupling types may contribute differently to the affinity. We have used several schemes of combination of the available coupling information to derive different affinity values. These schemes were named as *Unweighted Binary (UB), Weighted Binary (WB),*

Unweighted Additive (UA), Weighted Additive (WA) and Unweighted Multiplicative (UM). Their exact mathematical definition can be found in Table 2.

Scheme	Definition
Unweighted Binary	$A_{UB}(i,j) = \sum_{cc=DI}^{MR} C_{cc}(i,j)$
Weighted Binary	$A_{WB}(i,j) = \sum_{cc=DI}^{MR} \alpha_{cc} C_{cc}(i,j)$
Unweighted Additive	$A_{UA}(i,j) = \sum_{cc=DI}^{MR} N_{cc}(i,j)$
Weighted Additive	$A_{WA}(i,j) = \sum_{cc=DI}^{MR} \alpha_{cc} N_{cc}(i,j)$
Unweighted Multiplicative	$A_{UM}(i,j) = \prod_{cc=DI}^{MR} N_{cc}(i,j)$

Table 2. *Affinity rating schemes*

where:
α_{cc} - positive non-null weight associated with a given "CC" coupling category;
$C_{cc}(i,j)$ - predicate with values 1 or 0 stating whether classes "i" and "j" are coupled by at least one CC type of coupling or not;
$N_{cc}(i,j)$ - number of instances of CC coupling type, between classes "i" and "j".

In the multiplicative schemes, only terms with non-null $N_{cc}(i,j)$ value are considered.

The weights used in some of these rating schemes (the "weighted" ones, obviously) vary in the range [1, 10] and are calculated with information extracted from the original solution. Each weight for a given coupling category is given by:

$$\alpha_{CC} = \begin{cases} 10 & : \; CI_{CCIN} \neq 0 \wedge CI_{CCOUT} = 0 \\ 1 & : \; CI_{CCIN} = 0 \wedge CI_{CCOUT} = 0 \\ Round(0{,}5 + 10 \times \dfrac{CI_{CCIN}}{CI_{CCIN} + CI_{CCOUT}}) & : \; otherwise \end{cases}$$

where CI_{CCIN} and CI_{CCOUT} are, respectively, the number of intra-modular and inter-modular coupling instances for that category in the original modularization proposal. This technique partially preserves the original modularization criteria, at least as their developers perceived it.

From the affinity values $A(i, j)$, we calculate the **dissimilarities** $D(i,j)$ among classes using the following transformation:

$$D(i,j) = \begin{cases} \dfrac{1}{1 + A(i,j)} & \forall i \neq j \\ 0 & \forall i = j \end{cases}$$

This standardization transformation guarantees that $D(i,j) \in [0,1]$. Classes with low affinity have high dissimilarity and vice-versa.

With couplings rated according to the proposed schemes, and using this transformation, we can produce 5 dissimilarity matrices, one for each distinct scheme. Since we use 7 clustering methods, we obtain 35 distinct modularization solutions for a given number of modules. If we vary the number of modules in the interval ***]1, NC[***, where *NC* is the total number of classes, then the number of configurations ***NS*** to evaluate, in order to select the best one, is given by:

$$NS = 35 * (NC - 2)$$

For instance, a system with 100 classes would have 3430 alternatives. This takes us back to the problem of finding an adequate criterion to select the best among these alternatives.

2.3. A Dual Decision Modularization Criterion

Criterion C is incomplete since it implies that we already know how many clusters we should have. To solve the problem of finding the most adequate number of clusters, we herein propose the following quantitative modularization criterion:

Ironman criterion D – The number of modules should maximize the Modularization Merit Factor.

The ***Modularization Merit Factor****, henceforth designated by the MMF acronym, is defined as:*

$$MMF = \frac{ICD}{AMM} = \frac{CC_{IN}}{CC_{TOT}} * \frac{NM}{NC}$$

where*:*

$ICD = CC_{IN} / CC_{TOT}$ = Intra-modular Coupling Density
AMM = NC / NM = Average Module Membership
CC_{IN} = Class Couplings within modules
CC_{OUT} = Class Couplings between modules
$CC_{TOT} = CC_{IN} + CC_{OUT}$ = Total class couplings
NM = Number of Modules
NC = Number of Classes

The rationale for this metric is the following:

1. when NM and NC are held constant, the modularization will be better if the Intra-modular Coupling Density (ICD) increases;
2. when NC and ICD are held constant, the modularization will be better if the number of modules (NM) increases; in other words, we should split the system as much as possible, as long as ICD is not sacrificed (reduced);
3. when NM and ICD are held constant, the modularization will be worst when the number of classes increases, because we would then be increasing the Average Module Membership without increasing ICD.

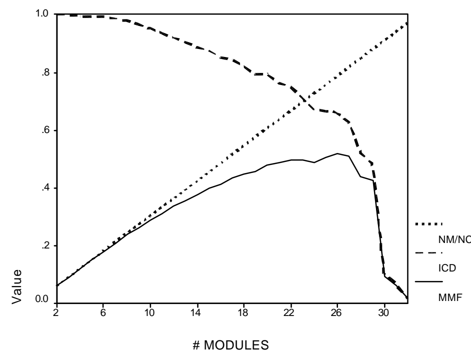

Figure 1. *MMF versus number of modules*

Figure 1 shows the dependency of MMF and its components (intra-modular coupling density and the inverse of the average module membership) on the number of modules for a fixed number of classes. The values were taken from one system of our sample described in Appendix A. The MMF curves for the other systems have similar shapes, all with an identifiable maximum. Notice that ICD $\in$ [0,1] and 1/AMM $\in$]0,1]. Therefore, MMF is necessarily restrained to the interval [0,1].

Criterion D does not account for the **dispersion** on the module sizes. If we simply use it to seize the best solution, the distribution of the module sizes would be highly skewed since the maximization of ICD would lead to a situation where we would have one module with a large membership and all the rest with very few classes, often only one. This configuration still guarantees that the average module membership (AMM) is kept low, but does not consider its dispersion, as mentioned previously.

Therefore, we add a second modularization criterion:

Ironman criterion E – The dispersion of the module size in the system should be constrained.

We have chosen a common dispersion measure, the standard deviation and applied its formula considering that we are dealing with the complete population of modules. We then obtain the following formula where CM_m is the number of classes in module *m*.

$$D = \sqrt{\frac{NM * \sum_{m=1}^{NM} CM_m^2 - \left(\sum_{m=1}^{NM} CM_m \right)^2}{NM * (NM - 1)}}$$

To find the most adequate solution we are faced with a multiple decision criteria problem. Our approach to this problem was to apply first the MMF-based criterion to select a subset of the best modularization solutions (e.g. 10% of the generated alternatives) and then apply the second criterion to this subset to derive the proposed “best” solution. This order reflects our view that criterion D is more important than criterion E.

3. The Methodological approach

3.1. Hypothesis

We hypothesized that current software systems are far from taking full advantage of the modularization mechanisms provided by OO design (a). The defined theoretical framework to guide the process of partitioning OO systems into clusters, showed that this task is feasible. Our experiment aimed at demonstrating that, with appropriate tool support, this task is also easy to accomplish (b).

All of the mentioned clustering methods and affinity rating schemes were tried. It is likely that the selection of a particular clustering method has an impact on the computed cluster organization (c). A similar reasoning can be made about affinity rating schemes (d).

3.2. The experiment

To evaluate these hypotheses we designed and conducted a controlled experiment. This experiment consisted basically on computing optimal modularization solutions for a relatively large set of OO software systems and then evaluating how far the original modularization solutions were from them. While doing so, the feasibility and easiness hypothesis (b) was evaluated. After completing the computation of the MMFs for all the systems in our sample, we used data analysis techniques to discuss hypotheses (a), (c) and (d).

The *MOTTO* tool was used to compute a range of the best values for MMF while applying criterion D and then, from that range, select the optimal value for MMF based on the application of criterion E. *MOTTO* (MOdularization Trial Tool for Object Oriented systems) was developed at INESC and is used in conjunction with *MOODKIT G2* described elsewhere [ABR 00] and *SPSS*, a commercial statistical package tool. *MOODKIT G2* generates *GOODLY* code both from design models or code. The same tool produces the coupling relations information that serves as input for *MOTTO*. The latter then computes the dissimilarity matrices for the selected rating schemes and generates *SPSS* scripts for the requested cluster methods and for the given cluster number interval. A *SPSS* batch facility is then invoked by *MOTTO* to compute alternative clustering solutions. These are then used by *MOTTO* to generate matrices of MMF values (one for each cluster number) that are then used to apply the mentioned criteria to derive the optimal solution.

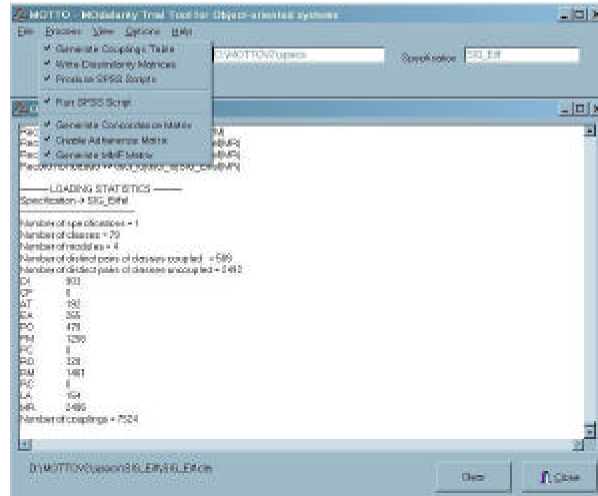

Figure 2. *The MOTTO tool*

3.3. Experimental design

3.3.1. Data

The input to this experiment consists on a sample of specifications of OO systems briefly described in Appendix A. The columns represent the system name, type (***APP**lication,* ***LIB**rary* and ***ENV**ironment*), formalism from which the *GOODLY* code was generated, counts of classes and couplings and two other columns that will be explained later in this paper. However, the bulk of the input data consists of typified class coupling instances that are used to calculate the affinity between classes and, from those, the dissimilarities matrix required as input by the clustering methods.

3.3.2. Threats to internal validity

A possible threat to this sort of experiment is the existence of unknown factors that may influence the MMF values. By conducting an experiment that is fully reproducible (in the sense that given the same inputs, i.e., the analyzed systems, the results of the described experiment are always the same) this risk is somewhat minimized. Nevertheless, there is a potential problem in the experiment design: all the original systems are converted from their source formalism to *GOODLY.* For instance, *Smalltalk* is a weakly typed language while *GOODLY* is a strongly typed one. To be able to fully

capture the exact class couplings in the translation from *Smalltalk* to *GOODLY* would require a much more powerful type inference mechanism than the one available in *MOODKIT G2.* Any loss of information caused by the conversion may have an effect on the conclusions derived from the analysis of the *GOODLY* version of the specification.

3.3.3 Threats to external validity

Since one goal is to generalize the results of this experiment, the representativity of our sample is a main concern here. To mitigate this problem, we chose a relatively large sample consisting of 21 OO systems, totaling 1224 classes. The following criteria were used while selecting the cases:

- systems should have some diversity; this is achieved with specifications of different programming languages, different types (libraries, environment, or applications), different application domains and different sizes;
- systems should be in use for several years or be produced in a recognized academic setting, so that their design structure was carefully engineered or had time to be refined by human experts.

3.3.4. Analysis Strategy

Our first step is to analyze some descriptive statistics on the computed Merit Factors. This is important as it allows us to choose adequate statistical tests to explore our hypotheses. Possible relationships between MMF and the configuration options available for their computation (clustering methods and affinity rating schemes) are also investigated, using standard analysis of variance methods (hypothesis (c) and (d)). The distance between the proposed optimal solution and the implemented one is then computed. This allows us to test hypothesis (a).

3.4. Conducting the experiment

The experiment consists on computing the MMFs for our systems sample. The experimental process is relatively straightforward as most tasks are fully automated by the *MOTTO* tool. To provide the reader with a grasp of the time involved in computing the best MMF for a given system, here is an example: suppose an experienced *MOTTO* user wishes to compute the best MMF for a given combination of clustering method and weighting scheme. On a Pentium III @500MHz, with 256Mb of RAM and 1Gb of virtual memory, the times presented in Table 3 were registered for 3 different systems. These elapsed times include the minimum required user interaction with the tool. It should be noted that for these three examples, the amount of human interaction with *MOTTO* is the same. It consists on selecting a specification and then launching in sequence all the production commands needed to compute the best MMF.

System	Classes	Couplings	Elapsed time
Structure	14	713	00m:28s
Stix	110	4119	02m:50s
GNUSmallTalk	246	6934	11m:29s

Table 3. *MOTTO benchmark*

4. Data Analysis

4.1. Choosing a clustering method

Seven clustering methods are used in this experiment. Different clustering methods may point to different clustering solutions and therefore to different MMFs. We can speculate that some methods lead to better solutions than others.

As a null hypothesis (H_0), we assume that there are no significant differences between the MMFs computed by all the methods. The alternative hypothesis (H_1) is that there are significant differences for at least one of the methods.

H_0: $\mu_1 = \mu_2 = \ldots = \mu_7$
H_1: $\exists\, i, j: i \neq j \Rightarrow \mu_i \neq \mu_j$

We use a one way analysis of variance (ANOVA) of the best MMFs obtained with each clustering method, to test the null hypothesis. Table 4 presents the results on this test.

	df	Sum of Squares	Mean Square	F	Sig.
Between Groups	6	2.633	.439	47.936	.000
Within Groups	876	8.009	.009		
Total	882	10.642			

Table 4. *ANOVA test for clustering methods*

Table 4 gives us diagnostics concerning between groups variance and within groups variance. The first data column gives us the degrees of freedom (df) connected with both the between groups variance (number of groups – 1) and the within groups variance (number of observations – number of groups). The sum of squares is presented next to the mean square. The statistic to check the null hypothesis is the ratio between the mean squares between groups and the mean squares within groups. The high value for the F

statistic allows us to reject the null hypothesis. The significance of the test tells us that the probability that these results occurred by chance is approximately zero. Therefore, we can reject H_0 and assume H_1: the chosen coupling method has an influence in the value of the MMF.

A Tukey-HSD (Tukey's Honestly Significant Difference) test with a .050 significance level will show where the difference discovered with the ANOVA really is (Table 5).

	Subset for alpha = .05			
Clustering Method	**1**	**2**	**3**	**4**
Clustering Method	**1**	**2**	**3**	**4**
CL	.229			
CL	.229			
WM	.232			
WM	.232			
WG		.265		
WG		.265		
BG		.286		
BG		.286		
MM			.326	
MM			.326	
CM			.341	
CM			.341	
SL				.387
SL				.387

Table 5. *Tukey-HSD test for clustering methods*

The rows of Table 5 display the mean for the best MMF obtained with the corresponding clustering method within our sample of systems. Methods are grouped in homogeneous subsets (columns 1 through 4). There, we can observe that the SL (Single Linkage) method is the one that provides the best coupling solutions, according to MMF. On the other extreme, CL (Complete Linkage) and WM (Ward Method) provide the less optimized solutions. The conclusion is that if we were to select only one of the clustering methods based on the MMFs obtained with this sample of OO systems, then SL would be the preferred one.

The SL method defines the distance between groups as the distance between their closest members.

4.2. Choosing an affinity rating scheme

The MOTTO tool provides 6 different affinity rating schemes. We hypothesize that different schemes may lead to different coupling solutions. A similar approach to the one used with the clustering method can shed some light into this issue. As a null hypothesis, we will assume the rating scheme has no influence in the MMF. The alternative hypothesis states that that influence exists.

H_0: $\mu_1 = \mu_2 = \ldots = \mu_6$
H_1: $\exists$ i, j: $i \neq j \Rightarrow \mu_i \neq \mu_j$

Table 6 presents the results of the ANOVA test.

	df	Sum of Squares	Mean Square	F	Sig.
Between Groups	5	1.077	.215	19.734	.000
Within Groups	877	9.565	.011		
Total	882	10.642			

Table 6. ANOVA test for affinity rating schemes

The F statistic value and its significance level confirm that we can reject our null hypothesis. As expected, the chosen rating scheme does have an influence on the selected cluster solutions and, consequently, on the MMF. A Tukey-HSD test (Table 7) shows that the binary schemes seem to lead to lower MMFs than their multiplicative and additive counterparts. Another interesting feature about this is that all the unweighted schemes had a slight advantage when compared to their weighted peers. Although the gap is relatively small, the whole purpose of including weights is to improve the performance of the rating schemes. In short, these results suggest that the weights need some calibration to become more effective.

	Subset for alpha = .05	
Affinity Rating Scheme	**1**	**2**
Affinity Rating Scheme	**1**	**2**
WB	.235	
WB	.235	
UB	.263	
UB	.263	
WM		.299
WM		.299
WA		.321

UM		.324
UA		.329
UA		.329

Table 7. *Tukey-HSD test for affinity rating schemes*

4.3. Modularization usage level

Earlier in this paper, we hypothesized that current software systems are far from taking full advantage of the modularization mechanisms provided by OO design.

We computed the optimal couplings for our sample of software systems, OPT_MMF, and then defined IMP_MMF as the MMF computed for the implemented system. These values are represented in Appendix A.

Figure 3 represents the IMP_MMF score against the optimal OPT_MMF, in each pair of vertical bars. The vertical scale represents the MMF score, which can theoretically vary between 0 and 1. We can observe that the large majority of the systems obtain a significantly lower score with IMP_MMF.

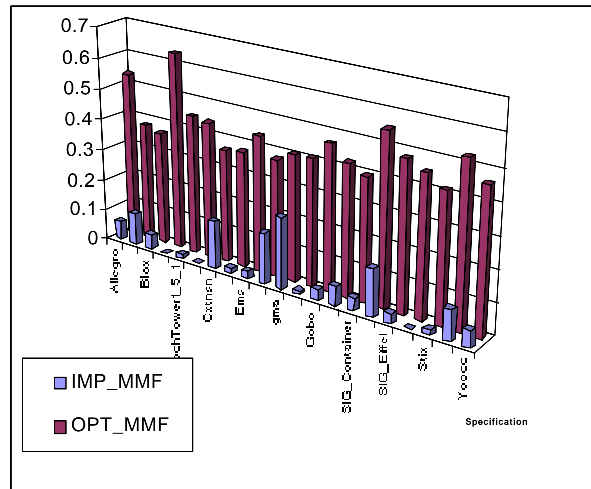

Figure 3. ***Implemented versus optimal MMF***

About 20% of the implementations use a larger number of modules than the one required by the optimal modularization. This means that classes are scattered throughout

an excessive number of modules, thus creating unnecessary coupling between modules relationships. The majority of the systems, however, should be split in more modules. Indeed, around 80% of the implementations had some classes with low affinity stored into the same module. This reduces the cohesion within the modules.

5. Related work

5.1. Cluster analysis in procedural systems

Clustering techniques have been applied in system structure recovering and identification of abstract data types in the context of reengineering of procedural based systems. They derive some similarity or dissimilarity measures based in the sharing of characteristics such as data structures or the existence of procedural calls.

A study to evaluate module-generating techniques, to help reducing the scope of errors, is presented in [HUT 85]. Four types of data bindings are used there to determine how two procedures are related. Two dissimilarity measures are presented and the authors use single linkage and a variation of this method to do the clustering.

In [SCH 91] a tool is presented for maintaining large software systems. The tool supports the grouping of related procedures into modules. The author proposes a similarity measure for procedures that is a function of the design information they share. A clustering algorithm is used to group related procedures into modules. The tool also allows finding procedures that seem to have been assigned to the wrong module.

The approach presented in [KUN 96] proposes a measure to evaluate the design of distributed applications, with several communicating processes. The similarity measure between pairs of processes uses data types related to communication. The author also applied hierarchical clustering algorithms for automatic generation of process hierarchies.

Cluster analysis was also used in [PER 98] to derive hierarchies of modules. Procedures are related using a dissimilarity measure that uses characteristics shared by the procedures. Several clustering algorithms were implemented in a tool that allows the specification of groups of characteristics to be considered in each analysis.

None of the mentioned works refer to object-oriented software systems.

5.2. Modularity analysis in OO systems

The evaluation of modularity in OO systems is a current research topic. In [POE 98] a modularity assessment technique based on the coupling strength of MERODE domain object classes is used. MERODE is an OO approach to model-driven development [SNO 98].

Two metrics for evaluating the modularity of OO frameworks: Variation points as Interfaces (VI) and Interface Coupling (IC) are proposed in [PRE 98].

None of those studies employs clustering techniques.

6. Future work

This paper described the steps followed while using cluster analysis techniques and class-coupling information to obtain modularization solutions in the realm of object-oriented systems.

The approach presented in this paper can be used both in the initial design phase and in the reengineering of object-oriented legacy systems, allowing to identify ill-defined modularization situations and proposing alternative ones.

Although cluster analysis techniques have been applied in procedural systems, we have not yet seen them applied in the context of OO systems. We hope this paper will contribute to foster this research area.

We plan to apply other multiple decision criteria techniques, currently used in the Operations Research world, to the problem we described in this paper.

The approach advocated here considers structural coupling only, and that may not match semantic coupling. We argue, but cannot prove here, that the latter causes the appearance of the former in coupling relations such as inheritance, aggregation or message passing ones. On the other hand, counter example situations may arise. In a graphical widgets library, for instance, the *Arc* and *Rectangle* classes could be completely apart but still it could make sense to include them in the same module since they both are two-dimension geometric figures. Nevertheless, our approach is completely generic and applicable as long as it will be possible to express quantitatively the semantic association between each pair of classes.

The presented work does not yet consider hierarchical modularization, which is a very important aspect when considering very large systems. Here we could argue that we could apply our approach in a stepwise manner, by first splitting the system in a given number of subsystems (e.g. fixing its number), and then applying the same approach again within each subsystem to identify nested packages. In the near future we intend to apply this nested approach possibly using different criteria at each abstraction level.

References

[ABR 99] F. B. ABREU, L. M. OCHOA, AND M. A. GOULÃO, "The GOODLY Design Language for MOOD2 Metrics Collection", presented at the ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Lisboa, Portugal, 1999.

[ABR 00] F. B. ABREU, G. PEREIRA, AND P. SOUSA, "A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems", presented at the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000), Zurich, Switzerland, 2000.

[BOO 94] G. BOOCH, *Object Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, LA, USA: The Benjamin Cummings Publishing Company Inc, 1994.

[BOO 97] G. BOOCH, I. JACOBSON, AND J. RUMBAUGH, "UML Semantics", Rational Software Corporation Version 1.0, January 1997.

[BRI 99] L. BRIAND, J. W. DALY, AND J. K. WUST, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", IEEE Transactions on Software Engineering, vol.25, n.1, 1999.

[CAN 96] M. CANTÙ, *Mastering Delphi 2 for Windows 95/NT*: Sybex, 1996.

[CON 90] L. L. CONSTANTINE, "Object-oriented and function-oriented software structure", Computer Language, vol. 7, pp. 34-56, January, 1990.

[EMB 88] D. W. EMBLEY AND S. N. WOODFIELD, "Assessing the Quality of Abstract Data Types Written in Ada", presented at the International Conference on Software Engineering (10th ICSE), 1988.

[GHE 91] C. GHEZZI, M. JAZAYERI, AND D. MANDRIOLI, *Fundamentals of Software Engineering.* Englewood Cliffs, NJ, USA: Prentice Hall, 1991.

[GOS 96] J. GOSLING AND F. YELLIN, *The Java Application Programming Interface*, vol. #1 (Core Packages) / #2 (Window Toolkit and Applets). Reading, Massachussets, USA: Addison-Wesley, 1996.

[HIT 96] M. HITZ AND B. MONTAZERI, "Measuring Coupling in Object-Oriented Systems", Object Currents, April, 1996.

[HUT 85] D. H. HUTCHENS AND V. R. BASILI, "System Structure Analysis: Clustering with Data Bindings", IEEE Transactions on Software Engineering, vol.11,n.8,pp.749-757, August, 1985.

[ISO 9126] ISO9126, "Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics", Geneva, Switzerland: ISO.

[JAC 92] I. JACOBSON, M. CHRISTERSON, P. JONSSON, AND G. ÖVERGAARD, *Object-Oriented Software Engineering- A Use Case Driven Approach*. Reading, MA, USA / Wokingham, England: Addison-Wesley / ACM Press, 1992.

[KAU 90] L. KAUFMAN AND P. J. ROUSSEEUW, *Finding Groups In Data: An Introduction To Cluster Analysis*: John Wiley and Sons, 1990.

[KUN 96] T. KUNZ, "Evaluating Process Clusters to Support Automatic Program Understanding", presented at the 4th Workshop on Program Comprehension, 1996.

[LOU 97] H. LOUNIS, H. SAHRAOUI, AND W. MELO, "Defining, Measuring and Using Coupling Metrics in OO Environment", presented at the OOPSLA Workshop on Object Oriented Product Metrics, Atlanta, USA, 1997.

[MEY 95] B. MEYER, *Object Success - A Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process*: Prentice Hall, 1995.

[MEY 97] B. MEYER, *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[PAG 95] M. PAGE-JONES, *What Every Programmer Should Know about Object-Oriented Design.* New York, USA: Dorset House, 1995.

[PER 98] G. M. PEREIRA, “Reengenharia da Modularidade de Sistemas de Informação", in Departamento de Engenharia Electrotécnica e Computadores. Lisboa, Portugal: master thesis, IST / UTL, 1998.

[POE 98] G. POELS, “Evaluating the Modularity of Model-Driven Object-Oriented Software Architectures”, presented at the ECOOP Workshop on Techniques, Tools and Formalisms for Capturing and Assessing Architectural Quality in Object-Oriented Software, Brussels, Belgium, 1998.

[PRE 98] P. PREDONZANI, G. SUCCI, AND A. VALERIO, “Object-oriented frameworks: architecture adaptability”, presented at the ECOOP Workshop on Techniques, Tools and Formalisms for Capturing and Assessing Architectural Quality in Object-Oriented Software, Brussels, Belgium, 1998.

[PRE 00] R. S. PRESSMAN, *Software Engineering: A Practitioner's Approach (European Adaptation),* 5th ed: McGraw-Hill Book Company, 2000.

[ROM 90] H. C. ROMESBURG, *Cluster Analysis for Researchers.* Malabar, Florida, USA: Robert E. Krieger Publishing Company, 1990.

[RUM 91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN, *Object-Oriented Modelling and Design.* Englewood Cliffs, NJ, EUA: Prentice Hall, 1991.

[SCH 96] S. R. SCHACH, *Classical and Object-Oriented Software Engineering*, 3rd ed. Burr Ridge, Illinois, USA: Richard D. Irwin, 1996.

[SCH 91] R. W. SCHWANKE, “An Intelligent Tool For Re-enginnering Software Modularity”, presented at the 13th International Conference on Software Engineering (ICSE'91), 1991.

[SNO 98] M. SNOECK AND G. DEDENE, “Existence Dependency: The Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types”, IEEE Transactions on Software Engineering, vol. 24, n. 4, April, 1998.

[SOM 00] I. SOMMERVILLE, *Software Engineering*, 6th ed: Addison-Wesley Longman, 2000.

[SOU 98] D. F. D'SOUZA AND A. C. WILLS, *Objects, Components and Frameworks with UML: The Catalysis Approach.* Reading, Massachussets: Addison Wesley Longman, 1998.

[STE 74] W. P. STEVENS, G. J. MYERS, AND L. L. CONSTANTINE, “Structured Design”, IBM Systems Journal, vol. 13, n. 2, pp. 115-139, 1974.

[STR 97] B. STROUSTRUP, *The C++ Programming Language*, 3rd ed. Reading, Massachusetts, USA: Addison-Wesley Publishing Company, 1997.

[SZY 98] C. SZYPERSKI, *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press / Addison-Wesley, 1998.

Appendix A

Specification	Type	Formalism	Classes	Total couplings	MMF_{cur}	MMF_{opt}
Allegro	**APP**	**Eiffel**	**33**	**3534**	**0.061**	**0.535**
Port of the Allegro game programming library to the SmallEiffel Eiffel Compiler.						
Bast	**APP**	**SmallTalk**	**38**	**587**	**0.102**	**0.380**
Object-oriented framework for building fault-tolerant distributed applications.						
Blox	**APP**	**SmallTalk**	**42**	**1264**	**0.048**	**0.365**
GUI building block tool kit. It is an abstraction on top of the platform's native GUI toolkit that is common across all platforms.						
BoochApp	**APP**	**Eiffel**	**18**	**115**	**0.000**	**0.634**
Small application to test the BoochTower library.						
BoochTower	**LIB**	**Eiffel**	**130**	**2894**	**0.014**	**0.449**
Library of structure components such as bags, graphs, lists, stacks, strings, trees...						
Canfield	**APP**	**Eiffel**	**9**	**47**	**0.000**	**0.439**
A solitaire game.						
Cxtnsn	**APP**	**SmallTalk**	**20**	**26**	**0.154**	**0.364**
C based extensions to GNU Smalltalk.						
dinamico	**LIB**	**Delphi**	**12**	**52**	**0.018**	**0.373**
Abstract Data Types library.						
Ems	**APP**	**Eiffel**	**111**	**3370**	**0.022**	**0.436**
Lexical analyzer that takes a list of classes in input and returns the analysis to the standard output.						
funcao3d	**APP**	**Delphi**	**18**	**146**	**0.164**	**0.377**
3D functions viewer.						
Gma	**LIB**	**Delphi**	**46**	**139**	**0.231**	**0.407**
Graphics package developed in Delphi.						
GNU_SmallTalk	**ENV**	**SmallTalk**	**246**	**6934**	**0.013**	**0.408**
Implementation of Smalltalk-80 under GNU's public license.						
GPCardGames	**APP**	**SmallTalk**	**20**	**362**	**0.063**	**0.422**
Implementation of card games.						
Gobo	**LIB**	**Eiffel**	**119**	**4563**	**0.034**	**0.467**
Eiffel libraries portable across various Eiffel compilers including: a Kernel Library, a Structure Library and an Utility Library.						
SIG_Container	**LIB**	**Eiffel**	**47**	**1215**	**0.042**	**0.396**
Containers library for SIG Eiffel.						

SIG_DateTime	**LIB**	**Eiffel**	**20**	**929**	**0.153**	**0.549**
Date and Time library for SIG Eiffel.						
SIG_Eiffel	**ENV**	**Eiffel**	**79**	**7531**	**0.030**	**0.478**
SIG Eiffel programming environment.						
SIG_Libraries	**LIB**	**Eiffel**	**34**	**2509**	**0.035**	**0.452**
Libraries for SIG Eiffel.						
Stix	**APP**	**SmallTalk**	**110**	**4119**	**0.018**	**0.414**
SmallTalk Interface to the X protocol layer that underlies all of X Windows.						
Structure	**LIB**	**Eiffel**	**14**	**713**	**0.098**	**0.525**
A data structure library based on circular-linked-lists.						
Yoocc	**LIB**	**Eiffel**	**58**	**457**	**0.054**	**0.461**
Compiler-compiler that uses an extended parse library which derives from the ISE Eiffel parse library.						

Legend:

$\mathbf{MMF_{cur}}$ – Current (observed) MMF
$\mathbf{MMF_{opt}}$ – Optimal MMF