Published in Software Engineering an International Journal (SEIJ). Vol. 1, No. 1, Sept. 2011 which should be cited to refer to this work

Assessing Legacy Software Architecture with the Autonomy Ratio Metric

Philippe Dugerdil
Geneva School of Business Administration, Univ. of Applied Sciences of Western Switzerland,
7 route de Drize CH-1227 Geneva, Switzerland
philippe.dugerdil@hesge.ch

***Abstract* - Among the software quality metrics, coupling and cohesion play an important role since they provide a clue about the structuring of the classes of the system. They are, therefore, computed at the level of the classes. However, when analyzing the architecture of a system, we are not only interested in the class level, but also in the higher levels of the system structure, for example the packages and components. This is especially true when we need to assess the quality of this structuring on the viewpoint of system understanding. In this paper, we first present the motivation for the definition of two new coupling and cohesion metrics that are applicable to higher structuring levels than classes. We then present our main metric: the autonomy ratio that measures the "functional structuring" of a system that we believe is essential to system understanding. Although, traditionally, the coupling and cohesion metrics are computed based on static analysis (i.e. source code analysis to find the potential calls among the elements), we rely on dynamic analysis and present the way the metrics are computed. Finally, we present a case study of the assessment of a large industrial system based on our metrics and the findings we drew from this experiment. We conclude the paper with a discussion of the results and present the future work. The key contribution of the paper is the definition of the autonomy ratio metrics for software architecture assessment on the viewpoint of system understanding.**

***Keywords* – Software metrics, system understanding, software architecture assessment, dynamic analysis.**

1. INTRODUCTION

Software metrics have long been used in the assessment of the quality of software systems and especially object oriented systems [15][31][30]. However, all of these works focus on the lowest structuring level of the software, i.e. the methods and classes. This is enough for low level software restructuring (i.e. what package the classes should be located to) and impact analysis purposes [14]. When we need to assess the quality of the architecture of some legacy systems on the viewpoint of system understanding, this is, however, not enough. We also need to make sense of the large-grain components of the software architecture represented, roughly speaking, by the different levels of the system substructure (syntactic grouping of elements such as package, modules, components, etc.) and to study the communication links between these substructures. Traditionally, software metrics are computed based on the static analysis of the software code. However, dynamic techniques, i.e. the analysis of the running of the code, have grown in importance especially in the software reengineering domain (see, for example, [23] [17][36]). In particular, they have been used to analyze the coupling between classes [1][3][27]. As far as software architecture assessment is concerned, there is no consensus on any single metric [12], since this process is related to the software quality attribute considered (QA) [26]. Therefore, any research on architecture quality assessment should make clear the quality attribute considered. In this paper, we clearly focus on system understanding (or the "understandability" QA which is strongly linked to the "maintainability" QA). In Section 2, we present the motivation for our work and the rationale for the definition of two new cohesion and coupling metrics: hierarchical functional coupling and hierarchical functional cohesion. From the latter, we can define, in Section 3, our main metric: the autonomy ratio of the substructures of the systems. In Section 4, we present the way they are computed based on the execution trace of the system. Section 5 shows a case study and Section 6 presents the related work. Finally, Section 7 concludes the paper and presents the future work.

2. PROGRAM UNDERSTANDING AND COMPONENTS COUPLING

It is well known that understanding takes the lion's share of the maintenance cost. Some studies say that code comprehension amounts to 50% of the maintenance effort [34], while others suggest that the ratio could be situated between 50%-80% [4]or even more: 50%-90% [11]. However, the system's understandability which is key to code maintainability has different meaning depending on the authors.

One widely-accepted definition is given by Biggerstaff and Mitbander [8]: "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."

In principle, any explanation on some phenomenon rests on the relationships we can make with something we already know. Explaining software is no different [7]. In particular, this is what we think was intended by the expression: "…*in terms that are qualitatively different…*" in the definition by Biggerstaff and Mitbander. However, since software is a formal system, it could be analyzed from the syntactic as well as semantic viewpoints. The syntactical structure of a program can be described at several granularity levels (i.e. from the level of a single program statement to the level of large structures like components or even subsystems) that are themselves associated with semantic interpretations. Therefore, any explanation of the software should address some or all of these granularity levels (Figure 1).

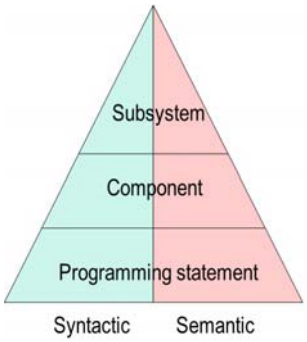

Figure 1: granularity levels of program structure and semantics

On the syntactic viewpoint, "understanding" means being able to relate the program constructs to the operational semantics of the programming language. This operational semantics must, of course, be already known by the software engineer for such an explanation to be useful. On the semantic viewpoint, "understanding" means being able to relate the program constructs to some external corpus of knowledge. The latter is represented by the knowledge of the purpose of the program in the business domain (the business function), plus some general computer science knowledge about good programming practices. However, the explanations of the program could be targeted to the different syntactical levels: single program statement, set of statements, components, packages and subsystems (i.e. sets of components). The granularity levels considered depend on the "explanation level" needed, which means the required level of detail on the working of the program. This depends on the software engineer's maintenance tasks and activities to perform on the system [34]. Sometimes, a broad overview on the subsystem's roles and responsibilities would be enough while, in other cases, the detailed understanding of some program statements would be required. Depending on the task, program understanding may work bottom-up through the syntactical levels (from the program statements up to the subsystems) or top down of both [34]. Now, if the task of an engineer is to understand the software system *globally,* for the purpose of assessing the way the program has been implemented, the engineer must usually work top-down starting from the whole application level (syntactic viewpoint) mapped to its purpose in the application domain (corresponding semantics). Next, he will decompose the system gradually according to the syntactical clues found in the program and map each program s*ubstructure* to the corresponding pieces of knowledge in the application domain.
In this study, we find it more convenient to speak about substructure than component because the latter is often related to behavior. Therefore, there is always an ambiguity about the viewpoint adopted: syntactic or semantic. Thus, we have:

Definition 1: a s*ubstructure* of a program or system is a source code declaration that:

- represents the syntactical grouping of program elements at any level above the method/procedure/function declarations;

and

- obeys a containment relationship.

For example, in Java, the substructures are the classes and all the packages that group the packages and/or the classes at any level of containment. Figure 2 presents the two ways containment could be represented in UML. The picture on the right part is very interesting in our context since it represents

containment as a graph where the nodes are the substructures and the edges represent the containment relationship (called "membership" in UML). We will refer to this graph as the *containment graph*.

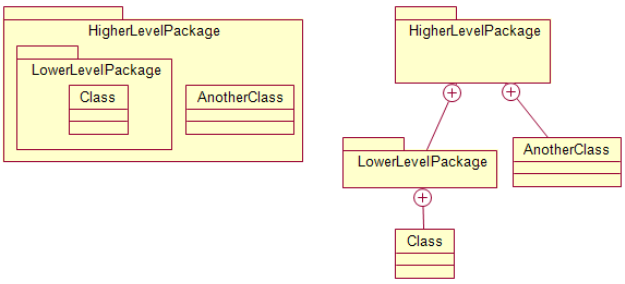

Figure 2: alternative UML containment representation

Whatever the substructure level considered, understanding will be facilitated if the system exhibits the "tree-like hierarchical quasi-decomposability" property [33]. In this case, the substructures could be studied in isolation; whatever the decomposition level, their purpose and working could be analyzed without needing to understand the other substructures of the system. This property means, in the software engineering world, that the substructures should exhibit low coupling and high cohesion. As far as system understandability is concerned, however, low coupling and high cohesion for a substructure is not enough for it to be understandable. It should also be mappable to some well-defined piece of knowledge (or "concept") in the external corpus of knowledge (application domain model). This idea is illustrated in Figure 3. When Counsell et al. empirically studied the intuitive meaning of class cohesion among a set of software engineers, they realized that the very notion of cohesion was subjective. In one paper they remarked: "In one sense, we could easily replace the word 'cohesion' in this paper with the word 'comprehension' " [18]. In another paper they strengthened their analysis and said: "Finally, if the research supports the view that cohesion is a subjective concept reflecting a cognitive combination of class features, then cohesion is also a surrogate for class comprehension" [19]. This is close to our view, but we extended this vision to substructure cohesion and coupling since we need to comprehend software at higher (coarser) granularity levels.

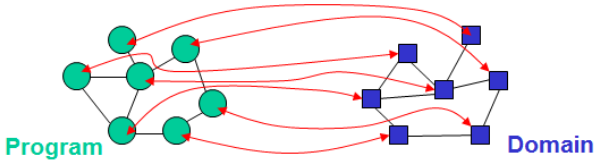

Figure 3: relationship between program elements and domain elements

In summary, when facing the task of assessing the architecture of some legacy software system from the viewpoint (quality attribute) of maintainability, understandability of the system is key. The latter is related to the way the system is split into substructures at different levels of granularity and to the links we may draw between these substructures and the concepts in the application domain. Therefore, the interpretation we give to cohesion and coupling are:

1. Cohesion of a substructure: strength of the *functional relatedness* of the elements within the substructure.
2. Coupling of a substructure: strength of the *functional dependencies* of the substructure with other substructures.

It is worth mentioning that Chidamber and Kemerer, in their seminal paper, justified the "Coupling Between Object Classes" (CBO) metric based on reuse and testing arguments only [15]. Moreover, the cohesion metric (or rather the lack of, since their metric is called "Lack of Cohesion in Methods," LCOM) claimed to be an indicator of class complexity. In contrast, our work rests on program understanding arguments based on the business purpose of the software. Of course, the relationship with complexity does exist, but it is not straightforward since the same substructure can be involved in several business functions.

3. DEFINING SUBSTRUCTURE COUPLING AND COHESION

3.1 Introduction

Cohesion and coupling could be measured based on static information as well as dynamic information. However, dynamic information (i.e. information on program execution) has the advantage of actually showing the coupling resulting from the actual processing of a scenario of interactions. Since there is an infinite number of scenarios that could be run on the system, we will focus on those that have value in the business i.e. corresponding to instances of the use-cases. Therefore, what we will study is the *dynamic functional architecture* of the system, i.e. the ways the substructures dynamically interact to implement some well-defined *business functions* (represented by the scenarios). By reference to the 4+1 views of Kruchten [28] this architecture belongs to the logical view, at the level of components or packages. In fact, the programming elements involved in the scenarios can be related to the substructures they belong to. Figure 4 illustrates the static structure of a program made up of packages and classes on top of which the grayed "cloud" represents the program elements involved in a scenario execution (when the "cloud" partly covers an element, this means that only a subset of its methods are involved). If the elements in the "cloud" are tightly coupled, they form what we call a *functional component* i.e. a set of program elements that closely work together to implement a subtask of a business function [21].

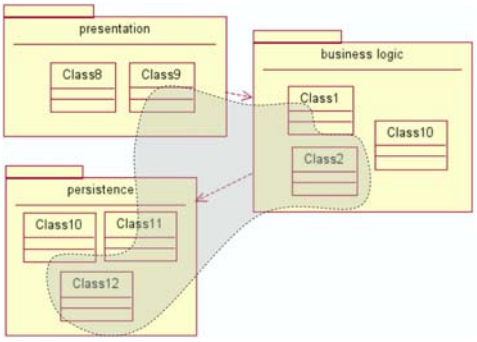

Figure 4: program substructures and functional components

Therefore, one possible architectural assessment of a system is to observe the extent to which the functional components map to the program substructures. If the latter map well to the functional components and if they are cohesive and weakly coupled, then we could assign them a meaning from the functional components (i.e. from the business function these components implement). Because there are generally fewer interactions between program substructures at coarser granularity levels than at finer ones, we expect both the cohesion and coupling metrics to be lower at coarser levels. This is illustrated in Figure 5 where the thickness of the links represents the "level" of interactions (number of messages sent between each other).

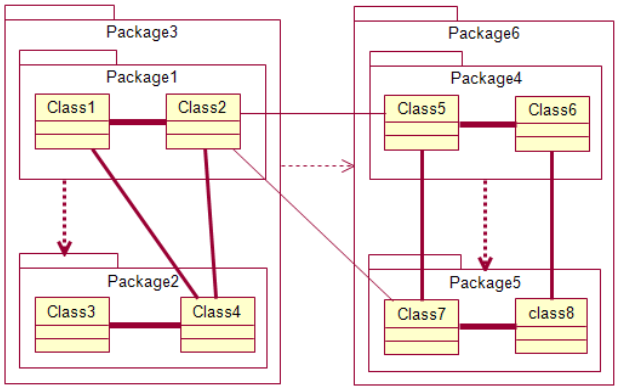

Figure 5: coupling and cohesion among substructure levels

As illustrated in Figure 5, the level of interactions between the classes in the same package are expected to be higher than the level of interactions among classes belonging to different packages which themselves are expected to be higher than the level of interaction among classes from different enclosing packages (packages 3 and 6 in the figure). This has some important consequences for the measurement of the coupling: the "benchmark" (i.e. the value against which we compare the metrics

[30]) will depend on the granularity level of the substructure considered. However, to be able to measure these metrics, we must define what coupling means at these different levels.

3.2 Computing dynamic coupling and cohesion among substructures

Dynamic coupling and cohesion metrics are computed based on the interactions (i.e. method calls or message sent) between program elements. Several studies on the dynamic coupling of object and classes and the comparison to static coupling have been published [2][3]. However, for a software system architecture assessment, we need to go beyond the class level. Indeed, we need to define dynamic coupling and cohesion at a higher (coarser) granularity level than the classes because this is the level at which the business functions are implemented. Therefore, highly cohesive and weakly coupled substructures could be the sign of a good functional decomposition of the system, hence a good "understandability." Among the authors having published on dynamic coupling we highlight the work of Arisholm et al. [3] because they defined a framework we refer to (see the annex, Section 10, where the approach of Arisholm is explained in some detail). However, Arisholm concentrates on the class level: the granularity levels above the classes are only shallowly dealt with (see the related work section). Since our definitions of coupling and cohesion are specific, we will use different names. Substructure coupling will be called *hierarchical functional coupling* or *hf_coupling*. Substructure cohesion will be called *hierarchical functional cohesion* or *hf_cohesion*. This will avoid any confusion with the common semantics for these metrics (although there does not seem to be a widely accepted one [17]). Nowadays, the software systems are mostly written by reusing open-source or commercial frameworks and components. We call the software system's *specific classes* the classes specifically written in the context of this system. The other classes could come from the reused commercial and open-source libraries and frameworks. Therefore, the scope of the metrics measurement could be the specific classes only or all the classes (the specific and the reused ones). Since our goal is to assess the quality of the software system's architecture, we only take the *specific* classes into account. Otherwise, we would assess the architecture of the framework, as well. Here are some definitions we need in order to specify our metrics.

Definition 2:

- *S*: set of all the program substructures in the software system that complies with Definition 1 in Section 2.
- *SC*: set of all possible scenarios (instances of use-cases) that can be executed on the system.
- *Distinct messages*: messages whose identification by the 4-uple [C1,m1, C2, m2] is unique. In other words, there is no pair of messages among all messages considered whose identification is the same. The interpretation of the 4-uple is: the method m1 in some instance o1 of class C1 calls the method m2 in some instance o2 of class C2. m1 and m2 are identified by their signature and C1,C2 by their fully qualified name.

When computing our metrics, we will only take the distinct messages into account. For example, if a given method in an instance of some class is called several times by the same method in an instance of another class, we count this event only once. In particular, if there are loops in the caller method then the repetitions of the same call are ignored. Referring to the definitions of Arisholm et al. [3], we count the "object-level distinct method import coupling metric" (IC-OM). The rationale for these choices is that we need to assess the architecture of the system from the point of view of the program understanding QA. Therefore, we need to know the "tasks" that some substructures delegate to another substructure. This task delegation means that the responsibility for the processing is shared among several substructures. Usually, the larger the variety of the delegated tasks by some substructure, the harder it is to understand the responsibilities of this substructure. (Remark: in these definitions, the classes considered are the actual classes of the instances that send and receive messages. This is because we want to know where the action really happens, not where it is defined.)

Definition 3: *hf_coupling* (*hierarchical functional coupling*) is a binary function on the set of program *substructures* S and the set of *scenarios* SC defined as:

$$\textbf{hf\_coupling: S x SC} \rightarrow \textbf{int} \qquad (1)$$

It is computed by counting the number of *distinct messages* sent by all the instances of the classes recursively contained by the substructure referenced by the first parameter to all of the instances of the classes located outside this substructure when executing some specific scenario referenced by the second parameter.

Figure 6, which is based on Figure 5, presents the elements involved in the computation of the hf_coupling of Class2, where the lines represent the messages sent by the instances of Class2 to the instances of the other classes when running some well-defined scenario. In this example, the metric value would be similar to the *import coupling* of Arisholm et al. [3], because we are dealing with the level of a single class.

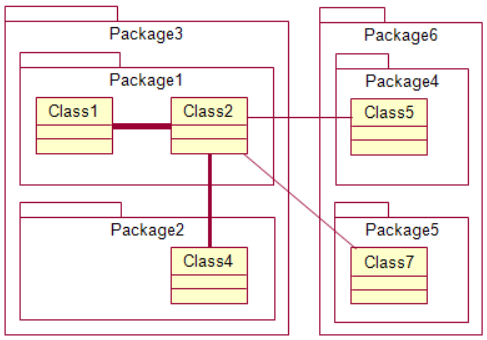

Figure 6: elements involved in the computation of the hf_coupling of Class2

Starting from Figure 6, Figure 7 presents the element involved in the computation of the hf_coupling of Package1. Again, the lines represent the messages sent by the instances of the classes in Package1 to the instances of the classes outside Package1 when running the scenario. This coupling measurement has no equivalent in the work of Arisholm et al. because we are now at the granularity level of a package.

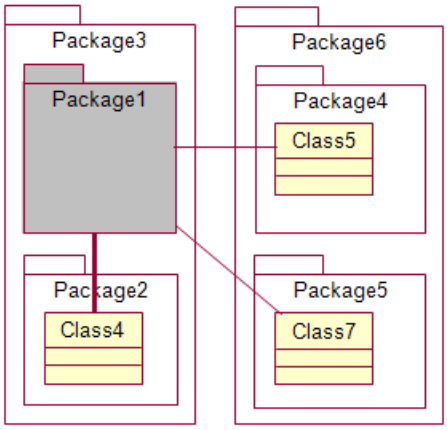

Figure 7: elements involved in the computation of the hf_coupling of Package1

Finally, starting from Figure 7, Figure 8 presents the element involved in the computation of the hf_coupling of Package3. The lines now represent the messages sent by the instances of the classes in Package3 to the instances of the classes outside Package3 when running the scenario.

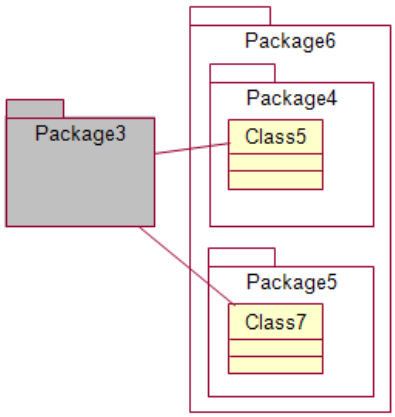

Figure 8: elements involved in the computation of the hf_coupling of Package3

In this computation, we can say that the substructure referenced by the first parameter of the hf_coupling function is considered to be a black box; we ignore what is going on inside the black box and we observe the message traffic between this black box and the rest of the system.

Definition 4: *hf*_cohesion (*hierarchical functional cohesion*) is a binary function on the set of program *substructures* S and the set of *scenarios* SC defined as:

$$\textbf{hf\_cohesion: S x SC} \rightarrow \textbf{int} \qquad (2)$$

It is computed by counting the number of *distinct messages* sent **among the direct children** of the substructure referenced by the first parameter in the containment graph, when executing some specific scenario referenced by the second parameter. However, we ignore the messages sent *inside* the direct children themselves which are considered black boxes. We only observe the message traffic between these black boxes.

Since the direct children of the substructure referenced by the first parameter may not be classes, we need to explain how this is computed. We say that some direct child of a substructure s_1 sends a message to another direct child of s_1 if some instance of a child (if it is a class) or some instance of a class recursively contained in the child (if it is not a class, for example a package) sends a message to an instance of another child (if it is a class) or to an instance of a class recursively contained in another child (if it is not a class). As an example, Figure 9, which is based on Figure 5, presents the elements involved in the computation of the hf_cohesion of Package1. The line represents the messages sent by the instances of Class1 to the instances of Class2 and vice-versa when running some well-defined scenario. For each class in the figure, we ignore the messages sent among the instances of this class and the messages from an instance to itself, because the class is considered a black box at this level.

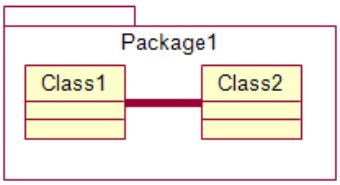

Figure 9: elements involved in the computation of the hf_cohesion of package1

Figure 10, based on Figure 5, presents the elements involved in the computation of the hf_cohesion of Package3. For each package in the figure, we ignore the messages sent among the instances of the classes recursively contained in the package because it is considered a black box. The lines represent the messages sent from some instance of some class recursively contained in Package1 to some instance of some class recursively contained in Package2 and vice-versa.

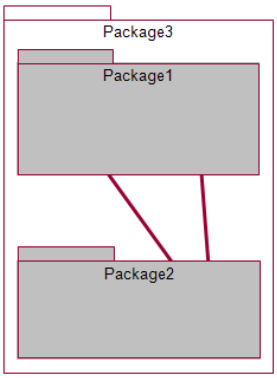

Figure 10: elements involved in the computation of the hf_cohesion of package3

Because of our very definition, the hf_coupling function exhibits the following property (whose demonstration based on the set algebra is trivial):

$$\forall\ \mathbf{A, B} \in \mathbf{S, A = father(B)} \Rightarrow \mathbf{hf\_coupling(A,s_1) \leq hf\_coupling(B,s_1)} \qquad (3)$$

Where father(x) returns the substructure that is the father of x in the containment graph. Interpretation: the value of the hf_coupling function is either stable or decreasing while moving to some higher level substructures.

As can be seen in the definitions, the metrics must always be computed for some specific scenario, usually an instance of a use-case of the system. Indeed, the analysis of the functional components of the system requires the business functions of the software to be explicitly known. In UML, the latter are expressed through scenarios. Finally, we notice that the above metrics are expressed in absolute numbers. This raises the well-known problem of the definition of a benchmark for the metrics [30]. In fact, we use the two metrics above to define the *autonomy ratio* of a substructure which is expressed in percent and whose interpretation is much easier.

Definition 5: *autonomy ratio* is a partial binary function on the set of program *substructures* S and the set of *scenarios* SC defined as:

$$\textbf{autonomy\_ratio: S x SC} \rightarrow \textbf{[0..100] (\%)} \quad (4)$$

If $hf\_cohesion(s_1,sc_1) + hf\_coupling(s_1,sc_1) \neq 0$, autonomy_ratio is computed as:

$$autonomy\_ratio(s_1, sc_1) = \frac{hf\_cohesion(s_1, sc_1)}{hf\_cohesion(s_1, sc_1) + hf\_coupling(s_1, sc_1)} * 100 \quad (5)$$

If $hf\_cohesion(s_1,sc_1) + hf\_coupling(s_1,sc_1) = 0$, autonomy_ratio is undefined (no value).

Interpretation: this ratio represents the "autonomy" of the substructure s_1 in the implementation of some functional component involved in scenario sc_1. The higher that the ratio is, the higher the autonomy of the substructure s_1 in implementing its duties in the scenario sc_1, and the lower the collaboration with other substructures. In other words, when running a scenario, we will observe several substructures that interact to implement the steps of the processing. If the autonomy ratio is high for a given substructure, this means that it autonomously implements some task in the processing. The limit values are:

- $autonomy\_ratio(s_1,sc_1) = 100$ means that s_1 is perfectly autonomous: the subparts of s_1 are not coupled to any other substructure outside s_1 to implement the responsibilities of s_1 with respect to the scenario sc_1. All of the services required by the subparts of s_1 are implemented by some other subparts of s_1.
- $autonomy\_ratio(s_1,sc_1) = 0$ means that s_1 is not autonomous at all: the services required by the subparts of s_1 to implement the responsibilities of s_1 with respect to the scenario sc_1 are all implemented outside s_1 and not by any other subpart of s_1.
- $autonomy\_ratio(s_1,sc_1)$ undefined: this is the case where the subparts of s_1 do not interact with each other or with any other substructure outside s_1. In other words, s_1 is only a container of pure server subparts that are not functionally connected with each other. In this case, the subparts of s_1 are not put in s_1 based on a "functional relatedness" but on some other criteria.

3.3 Autonomy_ratio as a way to assess system understandability

When trying to assess the quality of a software architecture from the viewpoint of the understandability QA, we compute the autonomy_ratio for all of the substructures of the system (in the case of Java, for all the packages) and for all the relevant scenarios. Then, we observe the distribution of the autonomy_ratio among the substructures. If each substructure gets a high autonomy_ratio ($hf\_cohesion(s,sc) >> hf\_coupling(s,sc)$), this means that we can study them almost in isolation and assign them some specific functional responsibility. In this case, the substructures exhibit the "quasi-decomposability" property [33]: to understand the working of each substructure we do not need to understand the other substructures outside them. At the other extreme, if the substructures have a low autonomy_ratio ($hf\_cohesion(s,sc) << hf\_coupling(s,sc)$) they cannot be understood in isolation since they delegate the work to the substructures outside them. We must, therefore, understand the latter to understand the working of the former. In summary, the higher the autonomy ratio of a substructure, the easier to assign it a role (i.e. a responsibility) in the implementation of some business function.

This shows why the autonomy_ratio is a good indicator of the functional decomposability of a system and therefore of its understandability (Section 2). We believe the autonomy ratio to be more appropriate as an indicator of system understandability than cohesion or coupling alone because:

1. It applies to all of the levels of the system's syntactical decomposition. This is required since the understanding of a system does not mean assigning a functional meaning to its lowest level elements only, but to all the relevant granularity levels (Section 2). The autonomy ratio indicates

the extent to which each level of the syntactical decomposition of a system could be assigned a functional meaning.

2. Cohesion is an indicator of the functional relatedness of the components of a substructure. But what if each of these components delegates some of their work to remote substructures? In this case, we must also understand these remote substructures in order to understand the working of the components. This shows that cohesion alone is not enough as an indicator of the "understandability" of a substructure.
3. Coupling is an indicator of the functional dependencies among substructures. Then, low coupling means low delegation of work to other substructures and we could hope to be able to understand a weakly coupled substructure in isolation. But what if the components of such a substructure never interact? This would means that the substructure cannot be assigned a unique functional role, but rather, a set of possibly unrelated roles. Therefore, coupling alone is not enough as an indicator of the functional role of a substructure.

4. MEASURING HF_COUPLING, HF_COHESION AND AUTONOMY_RATIO

4.1 Introduction

Since we collect the metrics based on the business function of a system, we must first know the use-case of this system. In the case of legacy systems it is often the case that no reliable documentation exists. Then, we must observe the actual users and abstract out their interaction with the system to redocument the use-cases. Next, we instrument the source code of the system to be able to generate a trace of the system's execution. The latter represents the set of the methods called during the execution. Finally, we run the system according to each relevant scenario (instance of use-cases) and record the corresponding execution trace. From this information, we will compute the metrics associated with each scenario. It is worth mentioning that in the case of legacy systems, we redocument the use-cases corresponding to the actual system usage. We ignore the scenarios that nobody uses. Since the computation of the autonomy_ratio depends on the scenario considered, the result gives us a measure of the system understandability *with respect to the actual system usage*. This is exactly what we need when facing system maintenance.

4.2 Format of the execution trace

To be able to reproduce the method call hierarchy, we record an event when a method is entered and when it is exited. The execution trace is therefore represented by a set of events having the following format:

1. [SP] [SC] [DP] [DC] '[' [TN] ']' [MS] [RT] '[' [TS] ']' [PV]

or

2. 'END' [SP] [SC] [DP] [DC] '[' [TN] ']' [MS] [RT] '[' [TS] ']'

Where:

- [SP] : full package name of [SC] ("static" package)
- [SC] : class where the called method is defined ("static" class)
- [DP] : full package name of [DC] ("dynamic" package)
- [DC] : class of the instance that received the message ("dynamic" class)
- [TN] : thread number
- [MS] : signature of the called method
- [RT] : returned type of the called method
- [TS] : time stamp of the call
- [PV] : parameter values of the called method (printable parameters only)

The first format of the trace event is generated when a method is entered for execution. The second format with the prefix keyword 'END' is generated when the corresponding method is exited (end of a call).
For example, let us have an instance "a" of Class11 in Package11 whose method m1() defined in Class1 of Package1calls the method m2() of some instance "b" of Class21 in Package21, m2() being defined in Class2 of Package2. Class1 is therefore a superclass of Class11 and Class2 a superclass of Class 21. This method call situation is represented in the sequence diagram of Figure 11.

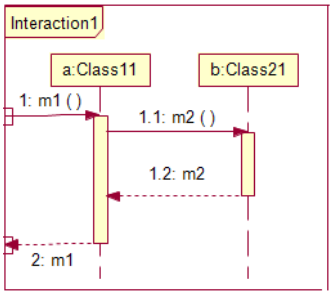

Figure 11: calls between 2 instances

In this case, we would find the following events in the trace file (if all method run in thread 1). In this example, the timestamp is arbitrary:

Package1 Class1 Package11 Class11 [1] m1() int [2345]
Package2 Class2 Package21 Class21 [1] m2() void [2346]
END Package2 Class2 Package21 Class21 [1] m2() void [2346]
END Package1 Class1 Package11 Class11 [1] m1() int [2347]

Here is an example of real trace events:

- pas.evi.cumulus.od iOdImage pas.evi.cumulus.od iOdImage [36] setAssetName(java.lang.String) void [12323324] "ffff_gggg_dffer_dfr_2009_3.jpg"
- END pas.evi.cumulus.od iOdImage pas.evi.cumulus.od iOdImage [36] setAssetName(java.lang.String) void [12323324]

The instrumentation of the source code is performed by an instrumentor we developed using the javaCC parser generator [25]. The AST of the parsed program is then visited (using the Visitor pattern [22]) to decorate the nodes corresponding to the method entry and to exit with the instrumentation code. The latter represents calls to an external program that actually writes the events to the trace file. Using such an external program minimizes the impact of the instrumentation to the original source file of the system to analyze. When the decoration of the AST is completed, the source code of the program is re-generated from the AST and recompiled. When the instrumented program is installed on a machine, we must also install the library that contains the runtime program that writes the events to the trace file. We decided not to write the trace events directly to a database for performance reasons. In this way, the impact of the instrumentation on the execution speed of the analyzed system is minimal. Figure 12 presents the tools workflow we have implemented to record the execution trace and process it. There is nothing new in such a workflow since several authors have used the same kind of technique (see, for example, [5]). However, because the volume of the execution trace we generate is quite huge (on the order of 10^6 events), we need to load the trace in a database before being able to process it.

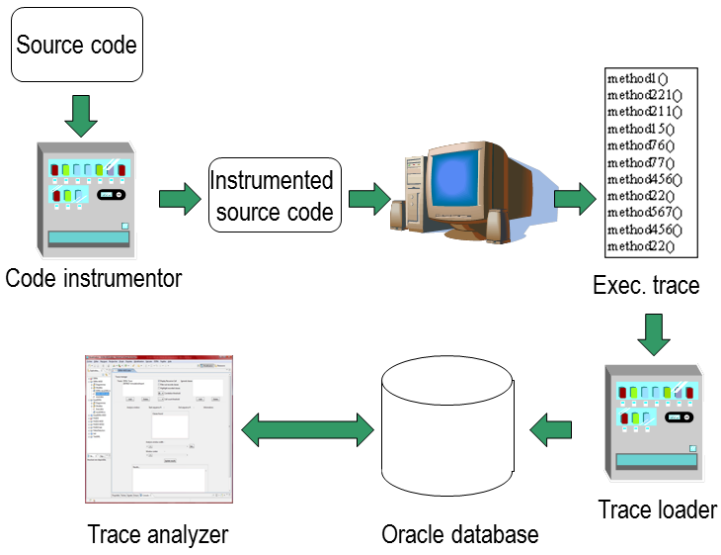

Figure 12: tools workflow for trace generation and analysis

4.3 Computing the metrics

Since we record the full package names in the events, it is easy to locate the classes in the containment hierarchy and to identify all of the substructures of the program. Then, the algorithm we use to compute the hf_cohesion and hf_coupling is simple: we just need to decompose the full package name to select the relevant level of substructure to analyze. For example, starting from the real trace events presented in §4.2, if we wanted to compute the autonomy ratio of the "pas.evi" substructure, we would consider all of the events having this prefix in their full package name. Then, we would first select the events representing the calls among its direct subparts to compute the hf_cohesion metric. Secondly, we would select the events representing the calls to substructures outside it (i.e. whose package name would not start with "pas.evi") to compute the hf_coupling metric.

5. CASE STUDY

The autonomy ratio metric has been specifically developed to perform the assessment of the architecture of the legacy software system in the perspective of the "understandability" QA. We were recently approached by the CIO of a company which, several years ago, bought a system from another company. However, the CIO always struggled to get this system properly maintained by the other company. To solve the situation, the CIO hesitated to buy the source code of the system to maintain it in his company. Before doing this transaction, he asked us to assess the architecture of the system and provide a possible explanation for the complexity of the maintenance. The system had on the order of 5300 Java (J2EE) classes distributed in about 600 packages with a high level of nesting.

First, we designed two dozen scenarios that represented about 80% of the common usage of the system. The corresponding executed code would then represent the implementation of most of the business functions used by the actual users of the system. In the case of maintenance, it would therefore be very likely that the problem would be located in this very code. Next, we instrumented the source code (JSPs, Servlets and Plain Java Objects) and installed the code on the server (Tomcat). When we executed the instrumented system based on the scenarios we observed that about 120 packages were involved in about each of the scenarios (counting all the levels of package nesting up to the root of the system structure). Finally, we analyzed the whole substructure (package) hierarchy to measure the autonomy ratio of all packages. We quickly realized that the autonomy ratio was weak for almost all of the packages. The results are presented in Table 1 below for the first 20 scenarios. To synthesize these results and show the distribution of the autonomy ratio (AR), we created three categories. We present the number of packages whose AR was bigger than 50%, those whose AR was bigger than 30% and finally those whose AR was bigger than 0.

TABLE 1
Results of the execution of 20 scenarios

Scenario #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
NB pack.	126	128	117	115	114	114	114	127	129	130	131	130	126	128	116	114	114	119	129	130
AR ≥ 50	7	7	6	5	5	5	5	5	7	7	8	8	7	7	6	5	5	5	6	7
AR ≥ 30	13	13	12	11	10	13	10	12	14	14	15	15	13	13	12	10	12	10	12	14
AR > 0	55	55	52	50	51	51	52	56	56	55	57	57	54	55	51	51	51	52	55	56

- Scenario # : scenario number
- NB pack. : total number of packages involved in the scenario, taking all levels of nesting into account up to the root package.
- AR ≥ 50: number of packages involved in the scenario for which the autonomy_ratio was bigger than or equal to 50.
- AR ≥ 30: number of packages involved in the scenario for which the autonomy_ratio was bigger than or equal to 30.
- AR > 0: number of packages involved in the scenario for which the autonomy_ratio was bigger than 0.

Analysis of the results:

1. The number of packages involved in all of the scenarios is remarkably similar, as is the number of packages for which the autonomy ratio (AR) is greater or equal to 50, greater or equal to 30 or

greater than 0. This tends to demonstrate that the functional architecture supporting all of the scenarios is rather similar.

2. The autonomy ratio of the packages is generally weak. About 5% of the packages have AR ≥ 50, about 10% have AR ≥ 30 and less than half of the packages have AR > 0.
3. The weak number of packages with AR ≥ 50 tends to demonstrate that only a few packages may actually implement functional components. In fact, the implementation of most of the functions seems to be distributed over a lot of packages. Therefore, in the case of maintenance, if some function must be modified, there will likely be several packages involved.
4. A detailed analysis of the packages with AR ≥ 50 showed that they are generally the same packages for all of the scenarios. This strengthened our feeling that the functional architecture is similar among all scenarios.
5. The packages with AR = 0 (more than half of the packages involved in the scenarios) are packages whose direct children are not functionally related. They represent substructures that do not collaborate when the business-related scenarios are executed. These components, therefore, do not have any functional cohesion (although their subcomponent may have some).

The interpretation of these results was:

- The architecture of the system is likely not to be based on functional components.
- The architecture of the system seems to be centered on a very small set of core components (server components) that are accessed by a lot of substructures at different levels of granularity (client components).
- The absence of a functional component architecture means that most of the maintenance tasks will likely involve several packages.
- The lack of functional component architecture could explain the relative difficulty to maintain the system (due to its low understandability).

Therefore, we advised the CIO not to try to maintain the system in his company because of the likely complexity of the system. Later in the project, we had a chance to talk to the development manager of the system provider who confirmed our analysis of the architecture of the system. Indeed, it is centered on a set of core components that are heavily parameterized and called from a set of smaller components that implemented the code specific to each customer. In a sense, each specific application was built as a composition of services called in the core components.

6. RELATED WORK

First of all, it is worth mentioning that the notion of cohesion does not have a well-accepted definition in the OO community [17], while coupling has been more widely studied [9]. Despite this fact, many papers have dealt with coupling and cohesion, but at the level of the classes. The seminal work of Chidamber and Kemerer [15] was about the first to propose a formal definition of the coupling metric (Coupling Between Object, CBO) and cohesion metric (or, rather, the lack of cohesion in methods, LCOM). However, it did not address the problem of larger program substructures other than the classes. Recently, Abreu at al. published the MOOD metric set among which the COF coupling factor computes a global value for the coupling of systems of classes. This metrics is based on the binary coupling among pairs of classes: if one class references the attributes and/or methods of another class, the coupling value is 1, 0 otherwise. The COF is the normalized sum of the binary couplings of all of the pairs of classes in the system. Again, the coupling is computed at the level of classes only. Later Briand, et al. redefined the LCOM metrics of Chidamber and Kemerer to remove some ambiguity by presenting five variants [9], but they stayed at the level of the classes. The work of Arsholm et al. [2][3] proposed a precise definition of class coupling by distinguishing the object level and class level metrics as well as the elements that are counted: the messages, the methods or the classes. However, this work concentrated on the class level. For the levels above, the classes they simply proposed to aggregate the results computed at the class level to the next granularity level. However, this treatment of a higher (coarser) level of granularity is not appropriate for the assessment of software architecture on the viewpoint of the understandability QA. In fact, coupling must measure the need for a substructure to rely on some other substructures to implement some business function. The proposal of Arisholm et al. does not, however, comply with this intuition since the value for coupling will monotonically increase when computed at coarser levels of granularity. Therefore, it basically indicates how tightly a set of classes is linked to the rest of the system's classes, irrespective of the *encapsulation* of the classes in higher level substructures. Counsell et al. introduced the normalized Hamming distance metric (NHD) [16] to compute class cohesion. It is based on the measurement of the similarity

of the method parameter types in a class. The rationale is, for the authors, that the more similar the parameter types between all of the methods, the more cohesive the class. Again, this is class metric only. The work of Kavitha and Shanmugam [27]presents a framework to compute the class coupling using the execution trace. The latter is called the "actual function call information" or AFCI. They then claim to use the "standard" coupling formulas (i.e. the one of Arisholm et al.) to compute the metric. Therefore, this work did not propose a new way to compute the metric. The work of Lui and Milanova [29] deals with the ways to combine static and dynamic analysis for the measurement of the coupling among classes. However, the coupling of higher level structures is not addressed. The paper of Washizaki et al. proposed a coupling-based complexity metrics for components. However, this paper focuses only on the EJB kind of components. In fact, they observed that the traditional coupling metrics based on the analysis of the source code of the classes is not relevant for EJB components since the environment generates classes at run time from the Home and Object interfaces. Moreover, several implementation classes may be required to implement a single EJB component. In this context, the coupling between individual implementation classes is not relevant. It must be replaced by the coupling among EJB components, i.e. among the classes that belong to different components. Then, they proposed a variant of the COF metrics of Abreu et al. using some binary coupling between the EJB components: if a class of a component references a class of another component, then the binary coupling among these components is 1, 0 otherwise. The resulting Component Coupling Factor (CCOF) is computed as the normalized sum of the binary coupling values among all of the pairs of components. This metrics represents the first step in the computation of some coupling metrics at a higher granularity level than classes. However, this work differs from ours in several ways. First, our metrics can be computed at all the different granularity levels of the system structure. It is not fixed at the first component level. Second, we compute the *strength* of the coupling between the substructures, based on the variety of the messages sent. Since we are interested in the autonomy of the substructures in their implementation of the system's functions, it is not enough to compute a binary metric for each pairs of substructures. We must know how strong the collaboration is. Third, our coupling metric is not an end in and of itself. It is used to compute the autonomy ratio which is our main metric. We believe the latter to be more informative on the point of view of system understandability.

In the context of Aspect Oriented programming, Burrows et al. [10] reviewed the metrics used to assess that maintainability of AOP software. These metrics are adapted from the OO context to account for the specificities of AOP. Of particular interest are the following metrics where the term *module* indicates either a class or an aspect:

- CAE (Coupling on Advice Execution): Number of aspects containing advices possibly triggered by the execution of operations in a given module.
- CIM (Coupling on Intercepted Modules): Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect. (This is the dual of the CAE metric.)
- CMC (Coupling on Method Call): Number of modules or interfaces declaring methods that are possibly called by a given module.
- CFA (Coupling on Field Access): Number of modules or interfaces declaring fields that are accessed by a given module.
- LCO (Lack of Cohesion in Operations): Pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative).

The first two metrics only measure the coupling brought about by the aspects. However, because of the definition of the very notion of module, the last three metrics can be applied to non-AOP programs, as well. In particular, the CMC and CFA metrics in the non-AOP program corresponds to the Chidamber and Kemerer CBO metrics (which takes both the methods and fields into account). Finally, LCO in a non-AOP program is similar to the Chidamber and Kemerer LCOM metric. Again all of these metrics are defined at the level of the classes (or aspects) only and not at higher levels of granularity of the program structure. Therefore, this work is not comparable to ours.

7. CONCLUSION AND FUTURE WORK

Our paper dealt with the formal metrics we implemented to assess the quality of the architecture of a system from the program understanding point of view (understandability QA). Since the software architecture of a program can be defined as: "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them" [6], we had to first define the structure of a program and the containment relationship of these structures. From this definition, we were able to propose new dynamic coupling and dynamic cohesion metrics applicable to whatever level of program substructure. We called these metrics hierarchical and functional because the value of the metric depends on the granularity level considered (hierarchical)

and the scenario (functional). Then, we defined our main metric: a ratio that measures the "autonomy" of a given component (substructure) to implement a task of a business function. The reference to the scenario is compulsory when computing the dynamic coupling and cohesion, since the value of these metrics will likely be different depending on the business function considered. When assessing the quality of a system's software architecture from the program understanding QA point of view, we must first identify all of the relevant scenarios. Then, we compute our metrics for each execution trace corresponding to each scenario. As a case study, we presented the results of a system architecture assessment project we did to help a CIO to take a decision about the maintenance of a system. This experiment showed that the autonomy_ratio metric was, in this situation, efficient at predicting the likely architecture of the system and provided a possible explanation about the difficulty to maintain the system. This analysis was later validated by talking to the system's provider directly.

The first contribution of this paper is to recognize that the coupling and cohesion metrics must be computed at higher levels of granularity than the classes for these metrics to be useful in the assessment of the quality of software architecture. The second contribution is the proposal of a formal definition of these metrics as well as a new ratio expressing the functional autonomy of the components.

As for future work, we intend first to calibrate our metrics with respect to different architectural styles (i.e. to show how the style of architecture influences the value of the metrics). Next, we must empirically study the relationship between system understandability and the distribution of the values of the autonomy ratio among all of the substructures (we will perform the same kind of study as reported in[18][19]).

8. ACKNOLEDGEMENTS

We would like to express our gratitude to our teaching assistant, David Sennhauser, who conducted all of the experiments that lead to the results presented in the case study. We are also grateful to the reviewers for their suggestions that helped us to improve the quality of the paper.

9. REFERENCES

[1] Abreu F., Goulão M., Esteves R. - Toward the Design Quality Evaluation of Object-Oriented Software Systems. Proc. of the 5th International Conference on Software Quality, Austin, Texas, 1995.

[2] Arisholm E.– Dynamic Coupling Measures for Object-Oriented Software. IEEE Symposium on Software Metrics (METRICS'02). 2002.

[3] Arisholm E., Briand L.C., Foyen A. – Dynamic Coupling Measurement of Object Oriented Software. IEEE Trans. on Software Engineering 30(8). 2004.

[4] Amyot D. Logrippo L. - Feature Interactions in Telecommunications and Software Systems VII, Ottawa, Canada IOS Press 2003

[5] Antoniol G., Di Penta M. - Library Miniaturization Using Static and Dynamic Information. Proceedings of the IEEE International Conference on Software Maintenance (ICSM) 2003.

[6] Bass L., Clements P., Kazman R.. *Software Architecture in Practice*, 2nd edition. Adison-Wesley Inc. 2003.

[7] Belmonte J., Dugerdil Ph. - Using Domain Ontologies in a Dynamic Analysis for Program Comprehension. 2nd International Workshop on Ontology-Driven Software Engineering. ACM Conference on Systems, Programming, Languages, and Applications (SPLASH) Reno,Nevada. 2010.

[8] Biggerstaff T.J., Mitbander B. - Program understanding and the concept assignment problem. Comm. of the ACM, 37(5) 1994.

[9] Briand L.C., Daly J.W., Wust J. - A Unified Framework for Coupling Measurement in Object-Oriented Systems. Empirical Software Eng., vol. 3, no. 1. 1998.

[10] Burrows R., Garcia A., Taïani F. - Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies. Proc. 4th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE'09) 2009 and in: Evaluation of Novel Approaches to Software Engineering, CCIS, Volume 69, Springer, 2010.

[11] Canfora G., Cimitile A. – Software Maintnance. University of Sannio, Faculty of Engineering at Benevento, Nov. 2000.

[12] Chastek G., Ferguson R. -Toward Measures for Software Architectures. Software Engineering Institute, Tech. Note CMU/SEI-2006-TN-013, 2006.

[13] Ceccato M., Tonnella P. - Measuring the effect of Sotware Aspectization. Proc. 1st Workshop on Aspect Reverse Engineering (WARE'04), Delft, The Netherlands. 2004.

[14] Chern R., De Volker K. - The Impact of Static-Dynamic Coupling on Remodularization. ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'08). 2008.

[15] Chidamber S.R., Kemerer C.F.– A Metrics Suite for Object Oriented Design. IEEE Trans. on Software Engineering 20(6). 1994.

[16] Counsell S., Mendes E., Swift S., Tucker A. - Evaluation of an object-oriented cohesion metric through Hamming distances. Tech. Rep. BBKCS-02-10, Birkbeck College, University of London, UK.2002.

[17] Counsell S., Swift S., Carmpton J. - The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design. ACM Transactions on Software Engineering and Methodology, Vol. 15, No. 2. 2006.

[18] Counsell S., Swift S., Tucker A. - Object-oriented cohesion as a surrogate of software comprehension: an empirical study. IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'05), 2005.

[19] Counsell S., Swift S., Tucker A. - Object-oriented Cohesion Subjectivity amongst Experienced and Novice Developers: an Empirical Study. ACM SIGSOFT Software Engineering Notes Vol 31 N° 5, 2006.

[20] Dugerdil P., Jossi S. - Reverse-Architecting Legacy Software Based on Roles: An Industrial Experiment. Communications in Computer and Information Science (CCIS) 22, pp. 114–127, Springer-Verlag . 2008.

[21] Dugerdil Ph., Repond J. - Automatic Generation of Abstract Views for Legacy Software Comprehension. 3rd Indian / ACM Conference on Software Engineering (ISEC'2010). 2010.

[22] Gamma E., Helm R., Johnson R., Vlissides J. – Design Patterns. Elements of Reusable Object Oriented Software. Addison-Wesley Inc. 1995.

[23] Hamou-Lhadj A., Braun E., Amyot D., Lethbridge T -. Recovering Behavioral Design Models from Execution Traces. *Proc IEEE CSMR*. 2005.

[24] Jacobson I., Booch G., Rumbaugh J -. *The Unified Software Development Process*. Addison-Wesley Professional. 1999.

[25] http://javacc.java.net/

[26] Kazman R., Klein M., Clements P. - ATAM: Method for architecture Evaluation. Technical Report CMU/SEI-2000-TR-004. 2000.

[27] Kavitha A., Shanmugam A. - Dynamic Coupling Measurement of Object Oriented Software Using Trace Events. IEEE Int. Symposium on Applied Machine Intelligence and Informatics (SAMI '08). 2008.

[28] Kruchten Ph. - The 4+1 View Model of Architecture. IEEE Software 12(6). 1995.

[29] Lui Y., Milanova A.– Static Analysis for Dynamic Couling Measures. Conf. of the Center for Advanced Studies on Collaborative research (CASCON '06) 2006.

[30] Lanza M., Marinescu R.– Object Oriented Metrics in Practice. Springer. 2006.

[31] Lorenz M., Kidd J.– Object-Oriented Software Metrics: A Practical G uide. Prentice Hall. 1994.

[32] Page-Jones M. 1998 -

[33] Simon H.A. - The architecture of complexity. In: The Sciences of the Artificial, MIT Press, 1969. (Reprinted in 1981).

[34] von Mayrhauser A., Vans A.M. - Program comprehension during software maintenance and evolution. Computer, 28(8), 1995.

[35] Washizaki H., Nakagawa T., Saito Y., Fukazawa Y. - A Coupling-based Complexity Metric for Remote Component-based Software Systems Toward Maintainability Estimation. Proc. 13th IEEE Asia Pacific Software Engineering Conference (APSEC06) 2006.

[36] Yan H., Aldrich J., Garlan D., Kazman R., Schmerl B. - Discovering Architectures from Running Systems: Lessons Learned. Software Engineering Institute. Tech Report CMU/SEI-2004-TR-016. 2004.

10. ANNEX: ARISHOLM FRAMEWORK

In this section we present the framework of Arisholm et al. [2][3] to compute dynamic coupling. Dynamic coupling is fundamentally based on the interactions (i.e. method calls or message sent) between program elements. Since the method calls happen between instances not classes or packages, the first problem is to link the calls between the instances to the corresponding coupling between their classes. This is the focus of the work of Arisholm et al. that distinguished *object coupling* and *class coupling*. Object coupling represents the coupling of the classes that are the actual classes of the interacting objects. Class coupling represents the coupling of the classes that *define* the calling and called methods, which could be inherited by the actual classes of the instances. Figure 13 reproduces the example presented in [3]. There are four classes A, A', B, B' where A' is a superclass of A and B' a superclass of B. When the instance "a" invokes the method mB'() of instance "b", we get two couplings :

- *object-level coupling* : the coupling between A and B because they are the actual classes of the interacting object
- *class-level coupling* : the coupling between A' and B' because this is the level at which the methods involved in the interaction are defined.

The metrics are computed by counting the number of calls between the instances while taking the following variants into account:

- *Dynamic messages*: we count only one call per distinct triple [caller class & method, called class & method, program line where the call takes place]. Then if the same method of the same class is called from different locations in the code of the calling method, it will be counted several times.
- *Distinct methods*: we count only one call per distinct pair [caller class & method, called class & method].
- *Distinct classes*: record only one call per distinct pair [caller class & method, called class].

Finally the authors distinguish between the *import* coupling of a class representing the methods called in other classes, and the *export* coupling of a class, representing the methods of this class that are called

by the other classes. The *scope* of the analysis is the set of classes taken into account in the computation of the metrics.

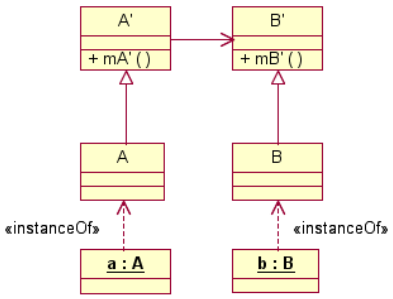

Figure 13 : Arisholm framework for the definition of class coupling

Philippe Dugerdil is Professor of Software Engineering at the Geneva School of Business Administration of the Univ. of Applied Sciences of Western Switzerland since 2002. Before, he spent 15 years in the software industry, mainly in banking environments. Philippe holds an engineer degree from the Swiss Federal Insitute of Technology in Lausanne (EPFL) Switzerland, a PhD Degree in computer science from Aix-Marseille II University, France and an MBA degree from the Institute of Management Development (IMD) in Lausanne, Switzerland