HAL
open science

Using Software Metrics in the Evaluation of a Conceptual Component Model

Éric Céret, Sophie Dupuy-Chessa, Guillaume Godet-Bar

To cite this version:

Éric Céret, Sophie Dupuy-Chessa, Guillaume Godet-Bar. Using Software Metrics in the Evaluation of a Conceptual Component Model. 4th Int. Conf. on Research Challenge in Information Science (RCIS'2010), 2010, Nice, France. pp.507-514. hal-00953311

HAL Id: hal-00953311

https://inria.hal.science/hal-00953311v1

Submitted on 28 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using Software Metrics in the Evaluation of a Conceptual Component Model

E. Ceret, S. Dupuy-Chessa and G. Godet-Bar
LIG Laboratory, CNRS University of Grenoble.
385 rue de la Bibliothèque
38041 Grenoble Cedex 9, FRANCE
{prenom.nom}@imag.fr

Abstract— **Every interactive system has a functional part and an interactive part. However the software engineering and the human-computer-interaction communities work separately in terms of methods, models and tools, which induces a work overhead for integrating the results of these efforts, as well as increased inconsistency risks. We endeavour to treat this problem by proposing a design method, which couples the functional kernel and the interaction design. In particular, this method proposes a specific way of structuring the interaction and the business spaces. The structure is based on components called Symphony Objects. In this article, we attempt to evaluate the technical aspect of a Symphony Object model issued from the method by measuring its implementations with software metrics.**

Information Systems; Human-Computer Interaction; Design Method; conceptual component; Quality; Metrics

I. INTRODUCTION

The evolution of computer technologies, in terms of communication (wireless networking) and interaction device (visualization headsets, tactile gloves) deeply alter the classical, implicit perception of Human-Computer Interaction (HCI). The users are now expecting adaptable and user-friendly interfaces. However, the methods are not all ready to design the interactional components of such HCI. Symphony is one of those methods, initially very strongly focused on business concerns, and augmented in the very last years for considering rich user interfaces [1].

The augmented version of Symphony was realized following some principles, mainly the use of reusable components early in the design process and a clear separation of concerns between the business and the human-computer interaction aspects.

The latter principle has been applied for describing several software architectures, such as the Seeheim model [20] or MVC [1]. Following this principle, the augmented version of Symphony features a separation of the design of the information system and that of its human-computer interface as early as the specifications phase. Therefore, the method specificies two distinct sub-processes for describing business and interaction conceptual components, which are respectively called Business Objects [3] and Interactional Objects [4]. During the following development phases, these concepts (which we gather around the term "Symphony Objects") are progressively refined into concrete components.

Symphony Objects are a major contribution of the augmented Symphony method: they were imagined to provide the designer with simple, consistent, structured and reusable constructs and consequently to encourage the production of good quality software.

But first, we needed to validate this extension of Symphony. Proving the quality of a design method is a hard work, but D.L. Moody [5] has proposed assessing a method by the evaluation of several of its aspects, from effectiveness to actual usage. Augmented Symphony was not ready for an actual use, so we decided to follow the first ones of his recommendations. As our method proposes a new way of structuring business and interaction spaces, we first needed to evaluate it in terms of its technical effectiveness (the extent to which the method improves the quality of the results): we evaluated if the method helps designers to produce good quality Symphony Objects models.

Many aspects of a model can be studied to determine its quality. The most immediate criteria are legibility, expressivity conciseness. In this work, we consider that a model is good if it gives rise to good quality in its implementations. Using this definition, our research question can be formulated as: **does a structure in terms of Business and Interactional Objects obtained from the use of the Symphony method give rise to good quality in the resulting software?** We try to answer this question with an experiment that we describe in this article.

Before answering this question, we will study related work about model and software evaluation. We will describe the Symphony method, with its extensions for HCI aspects, and particularly the concept of Symphony Object. Then section IV describes an experiment to evaluate Symphony Objects, based on different implementation solutions. Finally, we analyze the results of this experiment and conclude on the contribution of this approach.

II. RELATED WORK

A. Evaluation of Conceptual Models

Considering the Symphony Object model as a conceptual model, we study related works on the quality of models. There is little agreement among experts to define what makes a "good" model. If we consider the definition of quality given by ISO 9000, Moody [6] proposes to define the model quality as

"the totality of features and characteristics of a conceptual model that bear on its ability to satisfy stated and implied needs".

These needs can be defined with characteristics, which are decomposed into sub-characteristics following the ISO approach. For example, according to Lange & Chaudron [7], the first level of the model quality is the primary use of models, either development or maintenance. The primary uses are decomposed into purposes: for the development primary use, the purposes are communication, analysis, prediction, implementation and code generation. For each purpose, the required characteristics are specified. For instance, communication requires evaluating complexity, self-descriptiveness, conciseness and aesthetics. On another hand, complexity is defined as the effort required for understanding a model. After selecting some quality characteristics, a set of metrics is identified to measure them. Complexity metrics are the depth of inheritance tree or the number of classes per Use Case and so on.

Another approach is to integrate needs into a quality framework like the semiotic one proposed by Lindland [8]. In [8], quality is detailed into syntactic, semantic and pragmatic aspects. The syntactic quality verifies how well a model corresponds to its language constructs without considering the meaning. The semantic quality indicates the link of a model to its domain or to the knowledge of the domain specialists. Finally the pragmatic quality relates to the interpretation by the model audience.

Finally there is a reverse inference approach [6] where we work backwards from the quality characteristics of the final system to the characteristics of the model. Even if this approach is referenced in [6], we found no existing work using it. We choosed this approach to evaluate Symphony Objects because we hypothesised the existence of a causal relationship between the characteristics of our conceptual model (the Symphony Objects model) and the characteristics of the code made following its recommandations. So, we try to evaluate the Symphony Objects model through the quality of several of its implementations. We then focus on the technical effectiveness as defined in [3]. We hope that this will help us in defining some quality properties such as those proposed by the ISO approach. Then these properties can be used to understand the pragmatic quality of a Symphony Objects model.

B. *Software quality*

Our problem is now refined into the evaluation of implementations resulting from a Symphony Objects model. For evaluating an implementation, we can refer to software quality that has been studied for a long time in the software engineering domain. Software quality has given rise to standards such as the ISO/IEC 9126. The ISO/IEC 9126 software quality standard is one of the most widespread quality standard available in the software engineering community. It fixes six top-level characteristics, which are refined into twenty-seven sub-characteristics, which are in turn decomposed into properties that the software products belonging to the domain of interest should exhibit. For instance, software maintainability is the characteristic that refers to the effort needed to make specific modifications. Its sub-characteristics are stability, analyzability, changeability and testability. Some of these characteristics are measurable with metrics, like the number of lines of comments in a program. In our work, software metrics are used to measure software obtained from a Symphony Object model.

III. THE AUGMENTED SYMPHONY METHOD

A. *Presentation*

Originally developed by the UMANIS Company, Symphony is a method focused on business components. It has been extended, first to increase components' reusability **[3]**, and then to include the design of rich user interfaces [1].

Symphony is based on the separation of two purposes represented in a Y-lifecycle, as shown on Figure 1. The whole lifecycle is applied for each functional unit of the system under development.

The **functional (left) branch** corresponds to the traditional task of domain and user requirements modelling, independently from technical aspects,

The **technical (right) branch** allows developers to design both the technical and applicative architectures. It also federates all the constraints and technical choices with relation to security, pervasiveness, load balancing…

The **central branch** integrates the technical and functional

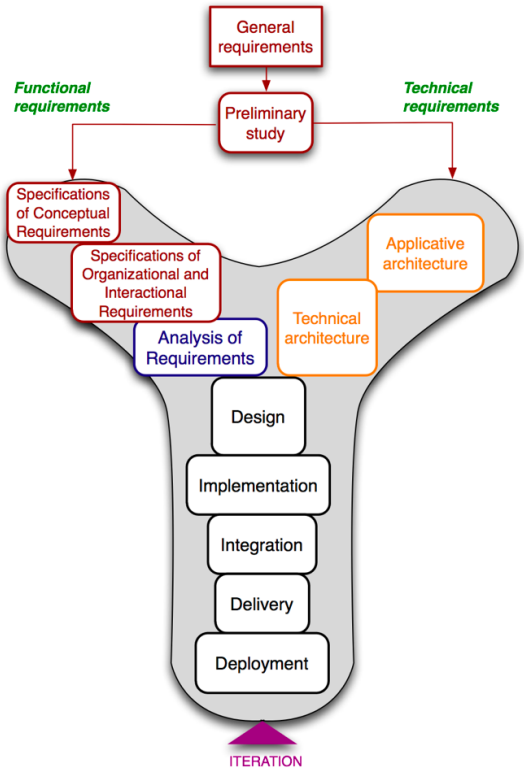

Figure 1. Symphony's development cycle

branches into the design model, which merges the analysis model with the applicative architecture and details traceable components.

B. *Reusable components in Symphony*

The left branch of Symphony consists in studying functional and interactional solutions by refining models previously outlined. Software Engineering (SE) and HCI-oriented activities are realized in parallel, by design actors specialized either in SE or HCI. Both conclude their analysis with conceptual components.

The original version of Symphony is based on the iterative identification and description of business components. For example, considering a tool for managing team members, two types of objects may be identified (Fig. 2): Business Object Entities (BOE) e.g. a person, and Business Object Processes (BOP), e.g. the applicative rules for adding or removing team members, for managing mentorships etc. Entities represent domain concepts, and processes describe the applicative logic of the system.

Our extension of Symphony is expected to improve the integration of rich user interface design, in order to gain modular, reusable and independent interactional components. It notably analyzes HCI concerns similarly to and in parallel with the more classical business analysis, thus allowing the definition of interactional entities and processes. An Interactional Object Entity (IOE) is an essential concept of the interactional domain, e.g. the graphical representation of a team (using tree-graphs, for instance). The Interaction Object Process (IOP) describes the logic of the interactional domain, e.g. the interaction techniques for adding or removing branches from a tree graph.

Figure 2 presents a part of a Symphony Objects model. Each object features an interface, which defines its contract with the outside world and an implementation of the contract, which includes a master class (for describing the main concern of the object) and part classes (for organizing the minor concerns the object depends on). Dependencies with other Symphony Objects are managed using role classes. These classes notably allow reusing existing Symphony Objects (for instance, the "Person" object) by adapting it to an applicative concern (for instance, the concept of team member, which is a person with an office and a position in the team hierarchy).

C. *Technical choices*

The right branch of the Symphony method generally recommends structuring software into a classical three-tier architecture, with the topmost tier describing the interaction devices (e.g., digital tables, OpenGL rendering components etc.), the middle tiers integrating the Symphony Process and Entity Objects and the lowest tier defining the database components.

Following common design practices, Symphony recommends the use of design patterns. In particular, all objects are designed with a "Factory", a class that is used to instantiate the objects [9]. Factories are not strictly recommended by the method, but using design patterns is generally advisable for obtaining good code quality.

IV. EVALUATING THE CONTRIBUTIONS OF THE SYMPHONY

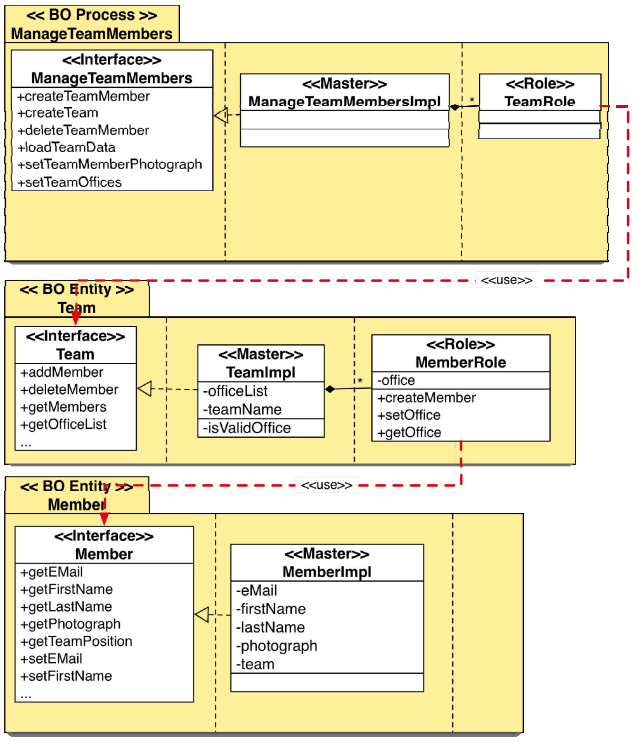

Figure 2. Symphony tripartite objects

METHOD

A. *Research methodology*

As we previously mentioned in the introduction, our goal is to evaluate the quality of a Symphony Object model obtained from the use of the augmented Symphony method. In particular, we want to evaluate if our model, that contains business and interactional objects, give rise to good quality software. In the produced software, we concentrate on the components that are systematically obtained from the design and implementation rules associated with Symphony Business and Interactional Objects, because we expect that the properties of those technical Symphony Objects to be induced by the properties of the corresponding analysis-level Symphony Objects model.

Considering the characteristics of the augmented Symphony method (i.e. the reusability of components and a clear separation of concerns between the HCI and the business parts), we would like to obtain the following code properties:

1. **The technical Symphony Objects must be modular.** Tripartite conceptual Symphony Objects have role classes, which are in charge of representing the external objects used by the object. So a technical Symphony Object must have all its external links concentrated in its role classes. As Symphony tries to separate interactional and business concerns, the dependencies between implemented Business

Objects and Interactional Objects should be very low, as to say nonexistent.

2. The technical Business Object Entities and technical Interactional Object Entities must be **reusable** at least in terms of a low complexity of their implementation classes and, all instability factors being concentrated into process Objet, they should have **a high functional stability**. As a matter of fact, they must be independent from the application logic, which is implemented in Process Objects.

B. Experiment

1) Evaluation Process

The previously mentioned properties being identified, we think about a way to evaluate them. As they are related to code quality, we study existing software engineering metrics. [15] identifies 225 metrics which can be classified into 2 categories:

- Metrics measuring directly programmers' work like the ratio of comments or the size of methods;
- The others evaluating the implementation recommendations of the method like the coupling between components.

We chose to focus on the method dependent metrics that are more relevant to our hypotheses.

Then we built an experiment to evaluate our hypotheses:

1. We chose a case study (controlled variable) which is a software to implement;
2. We designed the conceptual models according to the augmented Symphony method (controlled variable);
3. Several versions of the same software were developed by different programmers with different techniques (independent variables);
4. Then each of them have been evaluated with software metrics (dependent variables).

The details of this experiment are described below.

2) Controlled variables

Our case study is based on the results of a software engineering project, called "EDEMOI"[1], that deals with civil aviation security. The EDEMOI project originally assessed the conformance of a given security policy with the security standards, by automatically building and running test cases. However, the tests cases and their results were presented as lists of events and system states, which were difficult to understand for security specialists. Therefore, we were asked to design a user-friendly interface for visualizing these tests. The actual implementation features a rich graphical human-computer interface, developed in Java Swing [10] and Apache Project's Batik [11] library for manipulating SVG files. Persistency is handled using XML files. So this application offers the advantages of having a reasonable size with a complex user interface while its business logic is reasonably complex.

All our implementations are based on the same study of the EDEMOI project following the augmented Symphony method. The analysis document with its UML specifications, in particular its Symphony Objects model, and its technical choices (section 3.3) will not change during the experiment. They are our controlled variables.

3) Independent variables

The independent variables are those that we varied in order to obtain a significant experiment. Our independent variables are: 1) programmer experience; 2) some design choices (i.e. communication choices between Symphony Objects and their organization).

The first independent variable of our experiment is programmer experience. This criterion allows us to check whether good code quality is only due to a programmer's skills or to the method guidelines (i.e., average and inexperienced programmers should be able to produce satisfactory code, especially in terms of structure). Three implementations have been realized with two programmers. The first one was made by a PhD student, who is experienced in object-oriented programming. He used Java aspect-oriented programming to implement EDEMOI. The two other implementations were developed in JavaBeans by an experienced developer, who had a small experience in object-oriented programming. No constraint was given about the programming environment or the time required to produce code as our goal was to evaluate the code and not the way to achieve it.

The second independent variable concerns technical choices (Symphony's right branch). The first implementation uses a software framework whereas the two other implementations follow the MVC pattern [12].

Changing the type of technology allows checking whether a good quality is dependent on a very specific type of technology or is more generic.

The third variable carries on the design step (Symphony central branch). The main question was about the communication mechanism between technical objects. MVC can be implemented by using an Observer design pattern, but a more recent solution, proposed by R. Eckstein [13] proposes a solution based on a propertyChange event, as shown on figure 3. It permits a complete independence between View and Model objects. The role of the Controller is augmented, as it manages the events and their impact on each object. Therefore, it matches well with the processing applied by Process Objects and allows us to obtain a complete independence between the implemented Interactional and Business Entity objects.

[1] http://www-lsr.imag.fr/EDEMOI/

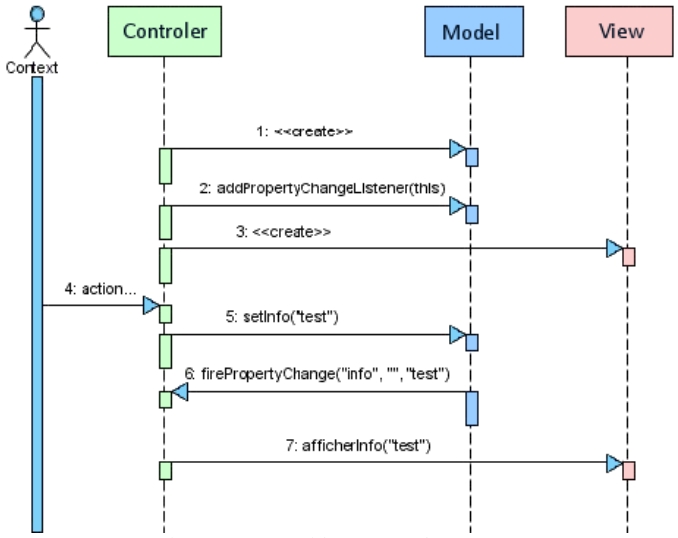

Figure 3. MVC with propertyChange event

Finally the last variable is the internal organization of objects. This criterion evaluates the impact of different implementation strategies for the Symphony Objects' "Master" and "Part" classes (i.e., finely grained classes or dense, complex ones) on the overall code quality. As a matter of fact, the code can be cut out in several ways in order to achieve the same goal: a large Master class can manage the whole problem of the object or a smaller Master class can be coupled with several Part classes to which parts of the problem are delegated. In the same way, inside a class, methods can be more or less important.

To summarize, we introduced variations at different levels of the design:

1. Programmer experience;
2. Communication management between technical Symphony objects, which gathers our 2nd and 3rd independent variables.
3. Organization of technical Symphony Objects.

For time reasons, we were not able to realize all the possible combinations of those independent variables. However we developed 3 different versions of the EDEMOI project with a specific goal for each one of them (Table. 1):

- A version using an original software framework (whose description is outside of the scope of this paper) that facilitates the implementation of Symphony Objects and of the connections between Business and Interactional Objects.
- The second version can be qualified as standard: it can be developed by any programmer adopting standard design choices.
- The last version could be realized by any programmer trying to make optimal choices. In particular, this version was developed by taking into account the results of software metrics tools run on our code.

TABLE I. INDEPENDENT VARIABLES OF THE EXPERIMENT.

	Programmer experience	Communi-cation between objects	Object Organization
Implem. #1 (Framework)	Experienced	Managed by the framework	Minimal sizes
Implem. #2 (Standard)	Intermediary	MVC with Observer	No care to size
Implem. #3 (Optimized)	Intermediary	MVC according to Eckstein	Minimal sizes

4) Dependent variables

The dependent variables are those that allow us to evaluate our hypothesis. In our case, they are the metrics measured on our three implementations. The metrics will give us some quantitative results without having to make any inference. We simply need to define their "good" values according to the literature.

We looked for software metrics that would permit to measure modularity and reusability of our technical components. Classic metrics such as the Lines Of Code / Lines Of Comments ratio can only be used to assess the programmers' coding practice, whereas we need to focus on the metrics related to the structure of the application. We focused on application architecture metrics, particularly those proposed by Chidamber and Kemerer [14] (coupling between objects, object complexity, depth of inheritance tree…). Among all the application metrics, we chose three of them that seem to be the most significant to measure modularity and reusability: coupling, complexity and instability. As these metrics are correlated, we cannot envisage to realize some meta-analysis on their results.

a) Coupling

Coupling indicates the level of interaction between several software components. Afferent (Ca) and Efferent Coupling (Ce) count respectively the number of ingoing and outgoing links between software entities, which are identified by Chidamber and Kemerer [14] as indicators of their modularity and reusability. As a matter of fact, the more a software entity is autonomous, the more it is easy to separate it from its original application; therefore it is reusable.

G. Booch [15] proposed the notion of "category", that is, a set of software classes that are designed to work together and be tightly coupled. Measuring the coupling of the classes within a category would make no real sense, therefore a relevant measure of coupling needs to take into account the software's architecture design. Concerning our implementations, Java packages correspond both to Symphony Objects and to categories. Therefore, we measure the coupling between Java packages.

In many metrics calculation tools, the value of the efferent coupling Ce did not match our expectations. To measure a package coupling, classic tools count the number of its classes that import other classes or packages. So, a package containing

only one class that imports 3 classes of other components will be counted as Ce = 1 (i.e. one class with dependencies). This didn't seem to measure actual dependencies between our components : we expected to have Ce=3 in this case.

This is the reason why we decided to define another efferent coupling: Ce^2, which counts the number of effective efferent links between packages, without considering standard language classes (which seldom evolve). This definition of efferent coupling enabled us to refine our perception of dependency, for it was the exact level of risk for a package to be impacted by the modification of one of the external objects it uses.

When considering a whole implementation, each ingoing link is also an outgoing link from the used object, thus the number of ingoing links is equal to the number of outgoing links. This is why we focused only on efferent coupling.

b) Complexity

Cyclomatic and Npath Complexity [18] measure the number of paths through a source code. They represent the number of unit tests required to cover the whole code. So, it's a vision on code testability, maintainability and granularity. Those qualities are necessary for guaranteeing the reusability of developed components.

c) Instability

Instability is defined as a ratio between afferent and efferent coupling: Ca / (Ca + Ce). It is usually interpreted as an indicator of the object resilience to change. Within Symphony, an object is expected to be either a service client or a service provider, i.e. that its instability should be close to either 1 or 0. When the component is distant from either mark, it has an unclearly defined role.

V. RESULTS

A. Hypotheses based on metrics measures

Our approach is not based on statistical analysis, we simply want to evaluate if the implementation of Symphony Objects give rise to acceptable results in terms of code quality. So considering the chosen metrics, we need to translate our high-level hypotheses into expected metrics values (Table 2):

1. We expect a low coupling value between the objects of our implementation. The literature on SE metrics recommends a measure of Ce such that $Ce \leq 4$. From our experience, we generally have $Ce^2 \approx 2 * Ce$, therefore we expect that $Ce^2 \leq 8$,
2. If coupling exists between two Symphony objects, we expect it to be concentrated on the dependency relationships between role classes and the interface class of the collaborating object (see Section 3.2), As a corollary, we expect the instability of our Symphony Objects to be concentrated into the role classes. So instability must be close to 1 for Symphony Objects and close to 0 for other objects.
3. We expect an overall low complexity, in order to guarantee a finer granularity and modularity of classes and components. We base our thresholds on those introduced by Watson and MacCabe [16] while defining the cyclomatic complexity (CC):
   - $1 \leq CC \leq 4$ – low complexity;
   - $4 < CC \leq 7$ – moderate complexity;
   - $7 < CC \leq 10$ – high complexity;
   - $10 < CC$ – very high complexity.

TABLE II. SUMMARY OF HYPOTHESES

	Hypotheses
Coupling metrics	$Ce^2 \bullet 8$
Instability metrics	I close to 0 or 1
Complexity metrics	CC • 4

B. Coupling

We expect a low coupling between our components i.e. $\mathbf{Ce^2 \leq 8}$. Moreover if coupling exists between two Symphony objects, we expect it to be concentrated on the dependency relationships between role classes and the interface class of the collaborating object. Table 3 presents the values for coupling. Ce^2_{tot} is the total coupling in a project, the total number of links between packages.

TABLE III. COUPLING IN OUR IMPLEMENTATIONS

	Ce^2 tot	Ce^2 max	Ce^2 avg
Optimized	21	5	1.82
Standard	46	7	3.00
Framework	13	2	1.00

Efferent coupling is lower than the values set in our hypotheses, for all our implementations. By looking at the code elements that are coupled, we note that they are the collaborative parts of tripartite technical Symphony Objects: the interface parts are used by the other components and the role classes use external components. Interfaces and role classes are easy to adapt as they are abstract classes. We can conclude than hypotheses 1 and 2 are verified.

C. Complexity

We expect an overall low complexity so as to guarantee a small granularity and a high modularity of classes and components. We measured Cyclomatic (CC) and Npath (NC) Complexity.

Our projects have very low CC values (between 1.46 and 1.79). But average values don't to be very significant, as they hide peaks. Looking at maximum values tells us a bit more: we expected to obtain values lower than 10. Our implementations, optimized, standard and with framework, are respectively evaluated to 5, 23 and 11, which are satisfactory results, above

all because the project evaluated at 23 is the less optimized one: its value is high, but acceptable considering that it does not exceed of more than an order of magnitude the values set in our hypotheses.

The situation is very similar with Npath complexity: we expected values lower than 50 [17]. Our implementations are in the expected range. Our optimized version has the lowest value (1.61), and Standard the highest (50).

But here too the maximum value gives a different vision: all projects, except Optimized have lots of method with high values, reaching up to 182000. Optimized has a maximum of 16, and Standard 13000. This can be explained by the fact that for the first one, we have been watching complexity all over the development cycle, splitting methods as soon as CC or NC was too high. This is not a Symphony specific problem, but the way our objects were designed made it simple to apply: the clear specialization of each object, the well identified role of every component facilitate the restructuring when needed. Those results tend to validate our fourth hypothesis.

D. Instability

Tripartite objects recommended by Symphony have by construction two roles: the contract part is managing the provider role of the object, while the collaboration part corresponds to its client role. This architecture should lead to bad instability values.

But Symphony also recommends splitting objects into entities and processes. Entities represent dense, consistent and relatively independent concepts. On the other hand, processes describe the applicative logic and use the entities they are considering. So, processes are strongly in a client role, while entities are mostly service providers.

For our three implementations, the values of I are particularly eloquent if we consider separately the entity and process objects. In fact, the interactional and business entities feature an instability of I=0 (no incoming link), while the process objects (some of these assuming the role of MVC controllers) have a value close to 1, at least greater than 0.75, meaning few outgoing links. These values match the distribution of applicative and entity roles proposed by the Symphony method, as detailed in Section III.B.

By construction, role classes are the only ones of our tripartite objects having an efferent coupling: interfaces, master and part classes have no incoming links. And, if we check it, efferent coupling is actually concentrated into role classes. For example, the optimized version, with its 11 packages, has 17 role classes, representing 95% of all efferent coupling. Those classes are only used within the object in wich they embedded, so none of them can have afferent coupling: cumulated afferent coupling in Optimized, Standard and Framework versions is actually 0 for role classes. Given that instability is defined as Ce / (Ce + Ca), all those classes have an instability of 1. Conversely, interfaces are measured with a null instability, and master and part classes are evaluated at I=0.14. This matches our third hypothesis: 96% of instability is concentrated into role classes. Moreover this analysis confirms the one realized on the coupling. So the validation of the 2nd hypothesis is strengthened.

E. Discussion

The Symphony Objects were designed to reduce the coupling between business and interaction elements. The experiments we conducted tend to validate the contribution of Symphony on this point.

However Symphony does not prevent from overly complex code when developers do not respect basic programming rules and practices. This point was not developed during these experiments because it seemed to be common sense. The point was to show that when respecting "good practices" in development, the Symphony method and its structuring objects give rise to reuse and modular components.

We could also develop the same application without the Symphony method and compare the metrics results with the ones with Symphony. In such case, we think that the results without Symphony would too heavily depend on the quality of the developer who would not be guided by a method.

Anyway evaluation is a very difficult task. At this point of our work, definitive statements would be very bold. We must, in particular, consider the instability results with precaution. As a matter of fact, we discovered during the experience that the EDEMOI project has a structural specificity: all its Entity Objects are directly linked to their Process Object. This specificity is in favour of a low efferent coupling for Entity Objects (they do not use other Entity Objects) and a high efferent coupling for Process Objects (they use the Entity Objects of the conceptual domain).

There are also several points which can influence the validity of our experiment: the size of the case study may be too small; the number of case studies is certainly not representative enough, or the fact that the sample developments were realized by two persons. Therefore, additional evaluation is required for refining our first conclusions.

VI. Conclusion

We showed how some code-oriented metrics actually help in evaluating a system model. Our way of experimenting a model is original as it is based on an under used approach, the reverse inference one, which is applied in a usual way i.e. by evaluating software metrics. Moreover it gives an insight *in fine* on the method that guided the description of this model. In this way, our experiment shows the applicability and the interest of the reverse-inference approach.

Our experiment has shown that a Symphony Object model, which represents business and interactional aspects, can give rise to implementations with modular, reusable and simple components. Thus we can conclude that 1) the augmented Symphony method can give rise to good Symphony Object models; 2) the Symphony Object model can be adequate to represent applications with quality requirements in terms of reusability.

From this first experiment, we can now introduce several variations in a Symphony Objects model in order to identify

their impact on the implementation quality. This would give us some insights about the technical quality properties for Symphony Objects models. Those quality properties can be completed by the study of related works on UML model quality and other experimental studies.

Now considering that the augmented Symphony method can give rise to technically good solutions, we are currently realizing qualitative experiments in order to evaluate the process in itself. We investigate whether the process integrates efficiently the practices of HCI and software engineering specialists and enables designers to develop rich user interfaces. We hope that these experiments will improve our insights into the interest of the augmented Symphony method and into the process of evaluating development method.

However, even though we try to minor the evaluation bias (for instance by multiplying code variations and programmers), we would need to apply our experimental protocol on a very large number of projects to guarantee our method's quality. This is a very severe limitation, given that the Symphony method would first need to be widely adopted on very different project types in order to be adequately evaluated. Olsen discussed the same limitations concerning usability evaluation [19], and suggested that new techniques should be first and foremost be considered in terms of the "important progress" they may provide.

Nevertheless, while the use of software metrics was applied for validating a model from a single development method, we believe that this approach could be generalized for providing an alternative solution for assessing method quality through the quality of the models and the implementations that they enable to produce.

REFERENCES

[1] Dupuy-Chessa S., Godet-Bar G., Pérez-Medina J.-L., Rieu D., Juras D., A Design process integrating mixed system into information systems, chapter of the book Engineering of Mixed Reality, eds L. Nigay, P. Gray and E. Dubois, Springer (2009).

[2] Krasner G., Pope S., A description of the model-view-controller user interface paradigm in the smalltalk-80 system, Journal of Object Oriented Programming 1 (3) (1988) 26-49.

[3] Hassine, I., Rieu, D., Bounaas, F., Seghrouchni, O.: Symphony: a conceptual model based on business components. In: SMC'02, IEEE International Conference on Systems, Man, and Cybernetics. Volume 2. (2002)

[4] Godet-Bar, G., Rieu, D., Dupuy-Chessa, S., Juras, D.: Interactional objects: Hci concerns in the analysis phase of the symphony method. In: 9th International Conference on Enterprise Information Systems ICEIS'2007, Funchal, Madeira, (2007) 37-44

[5] Moody, D.L.: The Method Evaluation Model : a Theorical Model for Validating Information Systems Design Methods. In 11th European Conference on Information Systems ECIS 2003 (2003).

[6] Moody D., Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions, Data & Knowledge Engineering, Vol 55 (2005) 243-276

[7] Lange C., Chaudron M., Managing Model Quality in UML-based Software Development, Proc. Of the 13th Int. Workshop on Software Technology and Engineering Practice (STEP'05) (2005) 7-16

[8] Lindland O., Sindre G. and Solvberg A., Understanding quality in conceptual modeling. 1EEE Software, April (1994) 42-49.

[9] Gamma, E., Helm, R., Johnson, R.E., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading (1995)

[10] Sun Microsystems. Java Swing. http://java.sun.com/docs/books/tutorial/uiswing/

[11] The Apache XML Graphics Project. Batik. http://xmlgraphics.apache.org/batik/

[12] Krasner G., Pope S., A description of the model-view-controller user interface paradigm in the smalltalk-80 system, Journal of Object Oriented Programming 1 (3) (1988) 26-49.

[13] Eckstein, R.: Java SE Application Design With MVC. Sun Developer Network (2007). http://java.sun.com/developer/technicalArticles/javase/mvc/.

[14] Chidamber S.R., Kemerer C.F., A metrics suite for object oriented design, Software Engineering, IEEE Transactions, Vol 20, (1994) 476 - 493

[15] Booch, G., Object-Oriented Analysis and Design with Applications (2nd Edition). Addison-Wesley Object Technology Series, (1994).

[16] Watson A. and McCabe T., Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. s.l. : Dolores R. Wallace, 51996).

[17] Scott Janzen, D.: an Empirical Evaluation of the Impact of Test-Driven Development on Software Quality. PhD Thesis (2006)

[18] Nejmeh B. A., Npath : a measure of execution path complexity ans its applications, in Communications of the ACM, Februrary 1988.

[19] Olsen Jr., D.: Evaluating User Interface. Systems Research. Proc ACM UIST'07. ACM Press (2007).

[20] Pfaff, G., and Hagen. P., Seeheim Workshop on User Interface Management Systems, Springer-Verlag, Berlin (1985).