Quantitative Methods in Object-Oriented Software Engineering

Fernando Brito e Abreu
ISEG (Lisbon Technical University) / INESC
INESC, Rua Alves Redol, 9, 1000 Lisboa, Portugal
+351 1 3100226
fba@inesc.pt

ABSTRACT

This paper includes a brief description of the author's doctoral research work in Quantitative Methods applied to the Object-Oriented Software Engineering field. Previous, current and future research work are outlined. An overview of related work is also included.

Keywords

Object Oriented Design; Software Metrics; Effort and Reliability Prediction Models; Software Engineering.

INTRODUCTION

The adoption of the Object-Oriented paradigm is expected to help produce better and cheaper software. The main structural mechanisms of this paradigm, namely, *inheritance, encapsulation, information hiding and polymorphism*, are the keys to foster reuse and achieve easier maintainability. However, the use of language constructs that support those mechanisms can be more or less intensive, depending mostly on the designer's ability. In fact, the analysis to design transition is an activity which offers several degrees of liberty.

Long before the OO languages became widespread, it was possible to build software with an OO flavor, using conventional 3rd generation languages. Conversely, by simply using an OO language that supports those mechanisms we are not automatically favored with an increase in software quality and development productivity, because its effective use relies on the designer's ability. Decisions on best alternatives are usually fuzzy and mostly based on expert judgment. Novice designers in particular, are exposed to a myriad of design decisions that surely affect the final outcome. We can then expect rather different quality products to emerge, as well as different productivity gains. Advances in quality and productivity need to be correlated with the use of those constructs. We then need to evaluate this use quantitatively (using design metrics) to guide the OO design process, for instance by means of design heuristics.

Metrics can also help software managers in the scheduling, costing, staffing and controlling activities by allowing to build effort, schedule and reliability models. Quantitative approaches in software engineering are now fully recognized and included in standards like [18, 19, 20, 21].

RELATED WORK

Since the early days of computer science many approaches to quantify the internal structure of procedural software systems have emerged [28]. Some of those "traditional" metrics can still be used with the object-oriented paradigm, specially at the method level. However, the need to quantify the distinctive features of this paradigm gave birth to new metric sets in recent years. Most of those metrics haven't been experimentally validated yet. This validation step usually consists of correlation studies between internal (design) attributes and external (quality) attributes. A brief survey of known efforts in this area follows.

Several research works in the OO design metrics arena were produced in recent years [8, 9, 11, 13, 17, 26]. However, there is a lack of experimental validation. Worse than that, there is scarce information on how the proposed metrics should be used. A better scenario can be found on the field of OO reuse metrics, where experimental studies like [22, 24] were a step ahead.

The MOOSE metrics proposed in [11] were validated in [7]. Besides discussing the metrics' advantages and drawbacks, the authors claim that several of them appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. Nevertheless, some criticism on the MOOSE metrics' imprecise and ambiguous definition (lack of language bindings) were raised in [12].

In [23] the authors used an extension of the MOOSE set to build a regression model that is said to be adequate to predict changeability (effort of correcting or enhancing classes). The model was validated with data from two systems built with an object-oriented dialect of Ada.

A metric proposed in [1], derived from the design information captured in class definitions for measuring the

number and strength of the object interactions, is claimed to be useful for predicting experts' design preferences. To validate the metric the authors used 9 sets of 2 or 3 design alternatives and compared the evaluations suggested by both the proposed metric and a panel of object-oriented design experts. They found out that the preferred alternatives were coincident in 80% of the cases.

Module and system level metrics for information hiding are described in [25]. A validation experiment based on a system with approximately one million lines of Ada code (an *object-based* language according to [27]) is described. Results showed that those metrics were able to "discriminate between packages that are or are not likely to undergo significant changes". On the other hand the authors recognize that the same experiment showed that there is no linear correlation between their information-hiding metric and change.

There is also an increasing interest from OO CASE tool makers in design metrics. Output from the ROSE tool (which supports the Booch method), for instance, is being used at Rational [14] to derive object-oriented metrics.

PREVIOUS WORK

The MOOD metrics

The participant has proposed the MOOD (Metrics for Object Oriented Design) set [2] which includes the *Method and Attribute Hiding Factors (MHF and AHF), the Method and Attribute Inheritance Factors (MIF and AIF), the Polymorphism Factor (POF) and the Coupling Factor (COF).*

These metrics are defined at the system or subsystem[1] level while in other approaches, such as the well know set proposed in [11], the metrics are defined at the class level.

The criteria used in the MOOD set definition were: *(1) coverage* of the basic structural mechanisms of the object-oriented paradigm as *encapsulation*, *inheritance*, *polymorphism* and *message-passing, (2) formal definition* to avoid subjectivity of measurement and thus allow replicability, *(3) size independence* to allow inter-project comparison, thus fostering cumulative knowledge and *(4) language independence* to broad the applicability of this metric set by allowing comparison of heterogeneous system implementations.

Each MOOD metric is associated with such basic structural mechanisms of the object-oriented paradigm as *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphism* (POF) or *message-passing* and *association* (COF). The mathematical definition of each MOOD metric will be introduced after the underlying basic concepts are made clear. Each metric is expressed as a quotient where the numerator represents the actual use of one of those mechanisms for a given design. The denominator, acting as a normalizer, represents the hypothetical maximum achievable use for the same mechanism within the same universe of discourse that is, considering the same classes and inheritance relations. As a consequence, these metrics are expressed as percentages, ranging from 0% (no use) to 100% (maximum use) and thus are *dimensionless.* This avoids the misleading, subjective or "artificial" units that are often found in the metrics literature.

Being *formally defined,* the MOOD metrics avoid subjectivity of measurement and thus allow replicability. In other words, different people at different times or places can yield the same values when measuring the same systems.

The MOOD metrics are also system *size independent.* This property allows inter-project comparison, thus fostering cumulative knowledge.

The MOOD metrics definitions make no reference to specific language constructs. The *language (in)dependence* will broaden the applicability of this set of metrics by allowing comparison of heterogeneous system implementations.

The metrics definitions, along with their underlying concepts are included in next sections.

Methods

The MOOD concept of **method** is that of a wrapped piece of procedural code (the body) whose execution as a whole is triggered by some agent. This is done through an interface that is identified by a unique name (within a certain range) and which may contain some mechanism of interchange with the calling agent (such as parameters or returned values). Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects.

Attributes

The MOOD concept of **attribute** is one of an independently identified data structure, either static or dynamic, transient or persistent, atomic or structured (e.g. record or array), which is used to store constants or variables. Attributes are used, among other things, to represent the status of each object in the system.

Methods and Attributes Visibility

The MOOD concept of **visibility**, associated with what is often referred to as the *range* or *scope* of an identifier, is related to the use of information hiding mechanisms. Each feature (method M or attribute A) is either visible or hidden from a given class C. If a feature is visible to a class C, then C can use that feature. Therefore, we can define the following logic function:

$$is\_visible(C_i.M_\alpha, C_j) = \begin{cases} 1 \quad iff \quad \begin{cases} j \neq i \\ C_j \ may\ call\ M_\alpha \end{cases} \\ 0 \quad otherwise \end{cases}$$

[1] - Collection of classes organized in some way to offer a given functionality as a whole.

$$is\_visible(C_i.A_\delta, C_j) = \begin{cases} 1 & iff \quad \begin{cases} j \neq i \\ C_j \ may\ reference\ A_\delta \end{cases} \\ 0 & otherwise \end{cases}$$

The *visibility of a feature* is defined as the percentage of the system classes, *other than the one where it was defined*, for which that feature is visible. Supposing TC is the total number of classes in the system under consideration, then the visibilities of method M_α and attribute A_δ, both defined in class C_i, are given by:

$$V(C_i.M_\alpha) = \frac{\sum_{j=1}^{TC} is\_visible(C_i.M_\alpha, C_j)}{TC-1}$$

$$V(C_i.A_\delta) = \frac{\sum_{j=1}^{TC} is\_visible(C_i.A_\delta, C_j)}{TC-1}$$

The denominator is the number of all classes except the one where the feature is defined. Function V may range from zero (the feature is hidden from all classes) to one (the feature is visible to all classes).

For the purpose of MOOD measurement, changes in the visibility of inherited features are accounted for in the class where they were initially defined. In other words, changes of visibility in any descendent class will eventually increase the number of classes that can potentially use the feature. A similar situation arises when we have feature name clashing in **multiple inheritance.** The resulting visibility of a feature inherited from two or more classes, which had different visibilities in each ascending class, will be the union of the corresponding visibilities.

Defined Features

Features **defined** in a class are the ones whose declaration lies within that class. That includes the ones that are not implemented (deferred or external features). We then define the following functions for any given class C_i:

$M_d(C_i)$ = methods defined in C_i

$A_d(C_i)$ = attributes defined in C_i

We are now able to introduce the Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) as:

$$MHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{M_d(C_i)}(1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

$$AHF = \frac{\sum_{i=1}^{TC}\sum_{m=1}^{A_d(C_i)}(1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

New and Overriding Features

A defined feature can be either a **new** or an **overriding** version of an inherited one. **New** features are the ones whose interface (name and/or parameters) is different from any inherited feature and thus do not override them. **Overriding** features are the ones that change the definition of inherited features. The following functions are then defined, for any given class C_i:

$M_d(C_i) = M_n(C_i) + M_o(C_i)$= methods defined in C_i

$A_d(C_i) = A_n(C_i) + A_o(C_i)$ = attributes defined in C_i

where:

$M_n(C_i)$ = new methods in C_i

$M_o(C_i)$ = overriding methods in C_i

$A_n(C_i)$= new attributes in C_i

$A_o(C_i)$= overriding attributes in C_i

Polymorphic Features

An important characteristic of the object-oriented paradigm is polymorphism[2]. This characteristic is such that a given message sent to class C_i can be bound (statically or dynamically) to a named method implementation in C_i or one of its descendants. Thus, the message recipient can have as many distinct implementations as the number of times this same method is overridden in C_i descendants. We then define the Polymorphism Factor (POF) as:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_n(C_i) \times DC(C_i)}$$

The numerator represents the *actual number of possible different polymorphic situations*. The denominator represents the *maximum number of possible distinct polymorphic situations* for class C_i. This would be the case where all new methods defined in C_i would be overridden in all its descendants.

Inherited Features

Inherited features in a class C_i are those which are inherited

[2] - From the ancient Greek "poly" (several) and "morphos" (shapes).

and not overridden in that class. An inheritance relation, for instance C_d inheriting from C_a, is represented by $C_d \rightarrow C_a$. We then define the following functions:

$M_i(C_i)$ = methods inherited in C_i

$A_i(C_i)$ = attributes inherited in C_i

Available Features

Available features (methods or attributes) in a class C are the ones that can be used in association with C. Available features are the defined plus the inherited ones. We then define the following functions:

$M_a(C_i) = M_d(C_i) + M_i(C_i)$ = available methods in C_i

$A_a(C_i) = A_d(C_i) + A_i(C_i)$ = available attributes in C_i

Now we can introduce the Method Inheritance Factor (MIF) and the Attribute Inheritance Factor (AIF):

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \qquad AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Coupling

Coupling is due to the representation of associations between classes (static coupling) and message exchanges between their instances (dynamic coupling). It can be identified by the existence of several kinds of references. In MOOD a class C_c is said to be a *client* of another class C_s (the *supplier*) and is represented by $C_c \Rightarrow C_s$ if C_c contains at least one non-inheritance reference to C_s. A reference can be made in an attribute or method argument type, a local method type (returned value) or even a call to a method belonging to the supplier class.

Clientele is represented by the *is_client* logic function. For the sake of simplicity, clientele shape and strength (number of references made to the client class) are not considered. Therefore we have:

$$is\_client(C_c, C_s) = \begin{cases} 1 & iff \quad C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & otherwise \end{cases}$$

The Coupling Factor (COF) is then defined as:

$$COF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is\_client(C_i, C_j)}{TC^2 - TC}$$

The numerator represents the *actual number of couplings not imputable to inheritance.* The denominator is the *maximum possible number of non-inheritance couplings* in a system with TC classes.

Metrics Extraction

MOODKIT G1, a tool to extract MOOD metrics from C++ or Eiffel source code was built and is being or was used by research teams in Portugal, UK (University of Southampton) and USA (University of Maryland). It uses specific "stubs", based on language parsers, for extracting the design metrics directly from those OO languages (by reverse engineering). As previously mentioned the MOOD metrics are supposed to be fairly implementation language independent. To achieve this independence the adopted approach was to develop bindings between MOOD and each specific language (e.g.: C++ [3] and Eiffel [5]). These bindings include (1) a mapping of concepts and terminology between MOOD and the language under consideration and (2) a description of how basic measures needed to compute MOOD metrics can be extracted from code written in that language.

One of the research goals was to conduct experimental validation on MOOD metrics by evaluating the impact of OO design on software quality characteristics such as *reliability and maintainability.* Using MOODKIT G1 (v1.0), the candidate made an extensive evaluation of available systems (C++ class libraries) and derived some design heuristics using a filter metaphor [3].

An experiment conducted at the University of Maryland [4] evaluated the impact of object-oriented design (expressed by the MOOD metrics) on resulting software quality attributes (defect density and rework). The results achieved so far allow to infer that in fact the design alternatives may have a strong influence on resulting quality. Quantifying this influence can help to train novice designers by means of heuristics [3] embedded in design tools. Being able to predict the resulting reliability and maintainability is very important to project managers during the resource allocation (planning) process. This work was a small step toward the understanding of how software designs affect resulting quality.

CURRENT WORK

MOODKIT G2 has a different architecture from its predecessor. It extracts the MOOD metrics from an OO design language also proposed by the candidate [6]. This language, named GOODLY (a Generic Object Oriented Design Language? Yes!), embodies information about class features (state and operations), relations between classes (inheritance, static coupling and dynamic coupling) and features scope. Parsers for extracting design information (GOODLY specifications) from several widely used OO languages such as Smalltalk, C++, Java and Eiffel are being developed.

This new architecture allows to redefine and extend the

design metrics set without the burden of changing the several OO language parsers (drawback in MOODKIT G1).

Further empirical validation experiments with a larger sample of projects and using MOODKIT G2 with GOODLY are planed for the near future.

A MOOD2 metrics set is currently being defined as a result of the opinions gathered and limitations perceived while using the original MOOD set. Among other things the candidate is trying to apply theoretical validation to the MOOD metrics by using Measurement Theory [15, 28]. This research action intends to answer questions such as:

- What kinds of *operations* can we perform with MOOD metrics?
- Can MOOD metrics be combined to express complexity in some way?
- What kind of *meaningful* statements can be made about a system who has a certain value for a given MOOD metric?
- Can we build *valid relationships* between the MOOD metrics (based on internal measurements) and external quality characteristics like reliability, maintainability or testability?

FUTURE WORK

I) MOOD Metrics in the Analysis Phase

Metrics should be collected and used to identify possible flaws as early as possible in the life-cycle, before too much work is spent based on them. It is a well known fact that the effort of correcting and recovering from those defects increases non-linearly with elapsed project progress since they were committed. Looking at the analysis instead of design would then be a step forward towards cost-effectiveness. The object-oriented paradigm is supposed, at least theoretically, to allow a seamless analysis-design-coding transition. Many analysis and design methods have emerged [10] in the past few years, with their own diagrammatic representations of differently named abstractions representing not-so-different basic concepts. This plethora gave birth to tools, such as ParadigmPlus or ObjectMaker, supporting multiple analysis and design methods. These tools map the information extracted from the distinct diagrams used by those different methods into a common repository, thus allowing diagrammatic con-versions. From those kind of repositories the candidate plans to generate GOODLY specifications (forward engineering).

II) Design Patterns Complexity

Object-oriented design patterns [16] are currently a very active research field. They seem to be the yellow brick road to the promised reuse-land. Substantial increases in quality and productivity are expected to happen if software developers really start using these new "bricks". However, the patterns' adoption greatly depends on their complexity, adaptability, functionality and reliability. All those characteristics must be quantitatively evaluated in order to define acceptance criteria, assess reuse potential and risk or compare different pattern implementations for similar functionalities.

If a pattern has an high complexity, potential users will not understand it and their adoption will be jeopardized. Measuring and establishing reasonable limits for a pattern's complexity seems to be a must. A generic OO complexity metric is expected to be built upon a combination of the MOOD metrics.

The functionality offered by a pattern represents its power to solve a certain category of problems. Some patterns have a much wider coverage than others in the sense that they can solve a given problem in many different contexts.

Patterns are not supposed to be used "as is" (verbatim reuse). Instead, they are supposed to be somehow configured or adapted (leverage reuse) to solve a particular problem of the system under construction. Therefore, their degree of adaptability should also be quantified. A reduced configuration capability would degrade the pattern's desired generality. Too much flexibility, on the other hand, would surely depend on several compromises which would sacrifice efficiency and memory usage optimization, provoke inadmissible increase in complexity and eventually produce undesirable side-effects.

The unreliability of a system that was built using an adopted pattern can be originated in the pattern itself or outside of it. Testing different systems with embedded patterns and selecting only the faults which depend on the patterns inclusion, should allow us to correlate them with the pattern's complexity. From there we can build predictive models for reliability or/and redesign patterns for an increased reliability.

REFERENCES

[1] D. H. Abbott., T. D. Korson and J. D. McGregor, "A proposed design complexity measure for object-oriented development", Clemson University, *TR94-105,* April 1994.

[2] F.B. Abreu and R. Carapuça, "Object-Oriented Software Engineering: Measuring and Controlling the Development Process", *Proceedings of the 4th International Conference on Software Quality,* McLean, Virginia, USA, October 1994.

[3] F.B. Abreu, M. Goulão and R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems", *Proceedings of the 5th International Conference on Software Quality,* Austin, Texas, USA, October 1995.

[4] F.B. Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality", *Proceedings of the Third International Software Metrics Symposium,* IEEE, Berlin, March 1996.

[5] F.B. Abreu, R. Esteves and M. Goulão, "The

Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics", *Proceedings of the TOOLS USA Conference,* Santa Barbara, July 1996.

[6] F.B. Abreu, Ochoa, L. and M. Goulão, "The GOODLY Design Language for MOOD Metrics Collection", *INESC/ISEG internal report (available at http://albertina.inesc.pt/ftp/pub/esw/mood)*, March 1997.

[7] V. Basili, L. Briand, W. Melo, "A Validation of Object-Oriented Design Metrics", *Technical Report CS-TR-3343,* University of Maryland, Department of Computer Science, May. 1995.

[8] M. Campanai, P. Nesi, "Supporting OO Design with Metrics", *Proceedings of TOOLS Europe 94*, France, 1994.

[9] S.N. Cant, Brian Henderson-Sellers, D.R. Jeffery, "Application of cognitive complexity metrics to object-oriented programs", *Journal of Object-Oriented Programming*, pp. 52-63, July-August 1994.

[10] Dennis de Champeaux, Penelope Faure, "A Comparative Study of Object-Oriented Analysis Methods", *Journal of Object-Oriented Programming*, vol. 4, n. 10, pp. 21-33, March / April 1992.

[11] S. Chidamber, C. Kemmerer, "A metrics suite for object oriented design", Center of Information Systems Research (MIT), WP No. 249, July 1993 , also published in *IEEE TSE* Vol. 20 (6), pp. 476-493, June 1994.

[12] N. I. Churcher, M. J. Shepperd, "Comments on 'A metrics suite for object oriented design' ", *IEEE TSE*, Vol. 21 (3), pp. 263-265, 1995.

[13] Reiner Dumke, "A Measurement Framework for Object-Oriented Software Development", submitted to the *Annals of Software Engineering*, Vol. 1, 1995

[14] Bill Fay, Jim Hamilton, Viktor Ohnjec, "Position/Experience Report", *Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics,* OOPSLA'94, Portland, USA, October 1994.

[15] Norman E. Fenton (editor), *Software Metrics: A Rigorous Approach*, Chapman & Hall (UK) or Van Nostrand Reinhold (USA), 1991.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[17] Trevor P. Hopkins, "Complexity metrics for quality assessment of object-oriented design", SQM´94, Edinburgh, July 1994, proceedings published as *Software Quality Management II, vol. 2: Building Quality into Software*, pp. 467-481, Computational Mechanics Press, 1994.

[18] Institute of Electrical and Electronic Engineers, *"ANSI/IEEE P-1061/D21 - Standard for a Software Quality Metrics Methodology*", 1990.

[19] International Organization for Standardization, *"ISO/IEC 14598 - Information Technology - Software Product Evaluation",* ISO JTC1/SC7, 1995.

[20] International Organization for Standardization, *"ISO/IEC 9000 / Part 3 - Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software",* ISO JTC1/SC7, 1991.

[21] International Organization for Standardization, "*ISO/IEC 9126 - Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their use",* ISO JTC1/SC7, 1991.

[22] John Lewis, Sallie Henry, Dennis Kafura, "An Empirical Study of the Object-Oriented Paradigm and Software Reuse", *Proceedings of OOPSLA'91,* ACM, pp. 184-196, 1991.

[23] W. Li, Sallie Henry, "Object-oriented metrics that predict maintainability", *Journal of Systems and Software,* Vol. 23 (2), pp. 111-122, 1994.

[24] Walcélio Melo, Lionel Briand, Victor R. Basili, "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems", *Technical Report CS-TR-3395,* University of Maryland, Dep. of Computer Science, January 1995.

[25] L. Rising, F. Calliss, "An information hiding metric", *Journal of Systems and Software,* Vol. 26, pp. 211-220, 1994.

[26] Brian Henderson-Sellers, "Identifying internal and external characteristics of classes likely to be useful as structural complexity metrics", *Proceedings of OOIS'94,* London, Dec.1994, Springer-Verlag, pp. 227-230, 1995.

[27] Peter Wegner, "Dimensions of Object-Oriented Design", *Proceedings of the OOPSLA'87 Conference,* pp. 168-182, October 1987.

[28] Horst Zuse, *Software Complexity: Measures and Methods*, Walter de Gruyer, 1991.