A Systematic Literatures Review of Impact of Quality Measurement of Object Oriented Software Design Environments

Shailaja M.[1], Dr. Vaibhav Bansal[2]

[1]Research Scholar of Sri Satya Sai University
[2]Research Supervisor of Sri Satya Sai University

Abstract

This paper describes the results of a study where the impact of Object-Oriented design on software quality characteristics is experimentally evaluated. A suite of metrics for OO design called MOOD was adopted to measure the use of OO design mechanisms. Data collected on the development of eight small-sized information management systems based on identical requirements was used to assess the referred impact. The results obtained in this experiment show how OO design mechanisms like inheritance, polymorphism, information hiding and coupling influence quality characteristics such as defect density and rework. Predictive models based on OO design metrics built in this study are also presented. Software errors are removed during different stages in the software development. Software metrics are used to measure the performance of software like cohesion, coupling, polymorphism, inheritance etc. These parameters help to provide the quality measurement, software maintain, fault detect prediction of the software. In the past, the static metrics were measured to predict the size and complexity of the software and it was successful but it didn't provide the run time parameter value. So, the dynamic metrics are executed to predict the cost estimation, defect prediction, etc. Fault prediction helps developers to reduce the error effectively in less time.

Keywords: *Metrics for Object-Oriented Design; Software Quality Characteristics; Error Prediction Model; Effort Prediction Model; Rework On Object-Oriented Software Development.*

1. INTRODUCTION

Software metrics help managers in several activities of the development life cycle such as scheduling, costing, staffing and controlling and therefore contribute to the overall objective of software quality. Their need is fully recognized by the software engineering community and included in standards like [IEEE1061, 1990], [ISO9000/3, 1991], [ISO9126, 1991] and [ISO14598, 1995]. Besides process factors, also product factors are expected to influence the resulting software quality. Among them is the design. The analysis to design transition is an activity where a skeleton for a computable implementation supporting the defined system requirements is defined. This transition often offers several degrees of liberty. Decisions on best alternatives are usually fuzzy and mostly based on expert judgment. By other words, cumulative knowledge plays a very important part in the design phase. The intensive use of patterns, frameworks and other reusable components is expected to ease this open problem, but current practice is still far from widespread adoption. Novice designers are therefore exposed to a myriad of design decisions that surely affect the final outcome. Being able to predict some of its features is one of our great motivations. This availability will allow to guide the designing process for instance by means of heuristics. Perhaps the most well known heuristic for object-oriented design is the Law of Demeter [Lieberherr et al., 1989]. This "law" restricts the message sending structure of methods in order to organize and reduce dependencies between classes. The authors say "...We believe that the Law of Demeter promotes maintainability and comprehensibility, but to prove this in absolute terms would require a large experiment with a statistical evaluation. ...". Unfortunately, to the extent of our knowledge, that hasn't been done yet. The main goal of this paper is to evaluate the impact of OO design on software quality characteristics such as defect density and rework by mean of experimental validation. In order to measure the OO design characteristics, a suite of metrics called MOOD [Abreu et al., 1994] was adopted. Motivations behind 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. The outline of this paper is the following: section 2 presents the MOOD metrics suite for OO design; section 3 describes an experiment where process and product metrics were collected; section 4 includes the statistical analysis on the collected data, discusses the validation of the adopted metrics set and finally proposes derived predictive models; section 5 includes an overview of related research works; finally, section 6 concludes the paper by presenting lessons learned and future work.

2. THE SUITE OF METRICS FOR OBJECT-ORIENTED DESIGN

The MOOD (Metrics for Object Oriented Design) set includes the following metrics:

• Method Hiding Factor (MHF)

• Attribute Hiding Factor (AHF)

• Method Inheritance Factor (MIF)

• Attribute Inheritance Factor (AIF)

• Polymorphism Factor (POF)

• Coupling Factor (COF)

Each of these metrics refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (POF) and message passing (COF). The MOOD metrics definitions make no reference to specific language constructs. However, since each language has its own constructs that allow for implementation of OO mechanisms in more or less detail, an abstracted binding for two OO languages, C++ [Stroustrup, 1991] and Eiffel [Meyer, 1992], is included ahead1. In the remainder of this section an overview of those metrics will be provided. Readers familiarized with MOOD can skip this section.

3. METRICS DEFINITION AND LANGUAGE BINDINGS

Method Hiding Factor:

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

where:

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC}$$

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & iff \quad C_j \quad can \quad call \quad M_{mi} \\ 0 & otherwise \end{cases}$$

MOOD		***C++***	***Eiffel***
TC	***total classes***	*total number of classes*	*same as for C++*
	methods	*constructors; destructors; function members2; operator definitions*	*class features with implementation (**do** clause) or without it (**deferred** clause); **external** functions; constants with **once** clause*
Md(Ci)	***methods defined** (not inherited)*	*all methods declared in the class including **virtual** (deferred) ones*	*all methods declared in the class, even if declared **obsolete**;*
V(Mmi)	***visibility** - % of the total classes from which the method Mmi is visible*	*=1 for methods in **public** clause; =1/TC for methods in **private** clause; =(1+DC(Ci))/TC for methods in **protected** clause (note: DC(Ci)descendants of Ci*	*= 1 by omission or if **ANY** is mentioned; = (1 + DC(Ci))/TC if **NONE** or empty brackets { } are mentioned; else = (1+ DC(Ci)+ number of classes between brackets {...} + their descendants + exports) / TC*

Attribute Hiding Factor :

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

where:

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC}$$

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 & iff \quad C_j \quad can \quad reference \quad A_{mi} \\ 0 & otherwise \end{cases}$$

MOOD		***C++***	***Eiffel***
Ad(Ci)	***attributes defined** (not inherited)*	*data members*	*class features without implementation; simple typed constants (integer, boolean, character)*
V(Am i)	***visibility** - % of the total classes from which Ami is visible*	*= 1 for attributes in **public** clause; = 1/TC for attributes in **private** clause; =(1+DC(Ci))/TC for attributes in **protected** clause note: DC(Ci)= descendants of Ci;*	*= 1 by omission or if **ANY** is mentioned; = (1 + DC(Ci))/TC if **NONE** or empty brackets { } are mentioned; else =(1+DC(Ci)+ number of classes between brackets {...} + their descendants + exports) / TC*

Attribute Inheritance Factor :

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

where:

$$A_a(C_i) = A_d(C_i) + A_i(C_i)$$

	MOOD	C++	Eiffel
$A_a(C_i)$	available attributes	data members that can be invoked associated with Ci	similar to $M_a(C_i)$
$A_d(C_i)$	attributes defined	data members declared within Ci	similar to $M_d(C_i)$
$A_i(C_i)$	inherited attributes	data members inherited (and not overridden) in Ci	similar to $M_i(C_i)$

Polymorphism Factor:

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

where:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

	MOOD	C++	Eiffel
DC(Ci)	descendants count	number of classes descending from Ci	number of classes descending from Ci
$M_n(C_i)$	new methods	function members declared within Ci that do not override inherited ones	features declared within Ci that do not override inherited ones
$M_o(C_i)$	overriding methods	function members declared within Ci that override (redefine) inherited ones	features in **redefine** and **undefine** clauses; deferred features which were inherited and implemented in Ci

The numerator represents the actual number of possible different polymorphic situations. Indeed, a given message sent to class C i can be bound (statically or dynamically) to a named method implementation, which can have as many shapes ("morphos" in ancient Greek) as the number of times this same method is overridden (in C i descendants). 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 their all derived classes.

Coupling Factor:

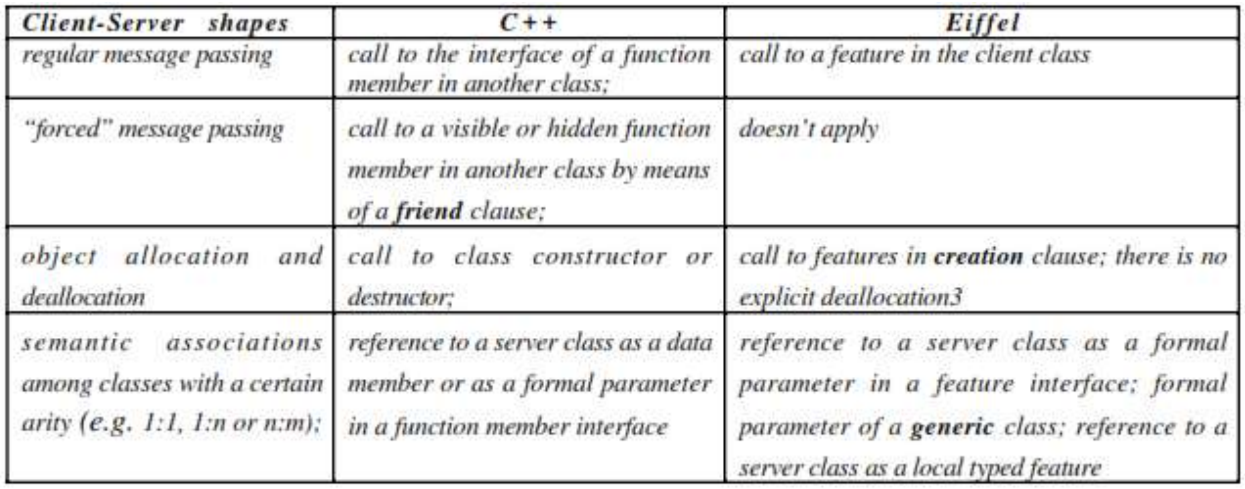
$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC}$$

where:

$TC^2 - TC$ = maximum number of couplings in a system with TC classes

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

The client-server relation (C C c s ⇒) means that Cc (client class) contains at least one noninheritance reference to a feature (method or attribute) of class Cs (supplier class). The numerator then represents the actual number of couplings not imputable to inheritance. The denominator stands for the maximum possible number of couplings in a system with TC classes. Client-server relations can have several shapes:



Client-Server shapes	*C++*	*Eiffel*
regular message passing	*call to the interface of a function member in another class;*	*call to a feature in the client class*
"forced" message passing	*call to a visible or hidden function member in another class by means of a **friend** clause;*	*doesn't apply*
object allocation and deallocation	*call to class constructor or destructor;*	*call to features in **creation** clause; there is no explicit deallocation3*
semantic associations among classes with a certain arity (e.g. 1:1, 1:n or n:m);	*reference to a server class as a data member or as a formal parameter in a function member interface*	*reference to a server class as a formal parameter in a feature interface; formal parameter of a **generic** class; reference to a server class as a local typed feature*

4. OBJECT-ORIENTED SPECIFIC METRICS

As discussed, many different metrics have been proposed for object-oriented systems. The object-oriented metrics that were chosen by the SATC measure principle structures that, if improperly designed, negatively affect the design and code quality attributes. The selected object-oriented metrics are primarily applied to the concepts of classes, coupling, and inheritance. For some of the object-oriented metrics discussed here, multiple definitions are given, since researchers and practitioners have not reached a common definition or counting methodology. In some cases, the counting method for a metric is determined by the software analysis package being used to collect the metrics.

A Class

A class is a template from which objects can be created. This set of objects share a common structure and a common behavior manifested by the set of methods. Three class metrics described here measure the complexity of a class using the class's methods, messages and cohesion.

A.1 Method

A method is an operation upon an object and is defined in the class declaration.

• METRIC 4: Weighted Methods per Class (WMC) The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement since not all methods are accessible within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children since children inherit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This metric measures Understandability, Maintainability, and Reusability [1,4,5,7].

A.2 Message

A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. The next metric looks at methods and messages within a class.

• METRIC 5: Response for a Class (RFC) The RFC is the carnality of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time. This metric evaluates Understandability, Maintainability, and Testability.

A.3 Cohesion

Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behavior. Effective object-oriented designs maximize cohesion since it promotes encapsulation. The third class metrics investigates cohesion.

• **METRIC 6:** Lack of Cohesion of Methods (LCOM) LCOM measures the degree of similarity of methods by data input variables or attributes (structural properties of classes. Any measure of separateness of methods helps identify flaws in the design of classes. There are at least two different ways of measuring cohesion:

1. Calculate for each data field in a class what percentage of the methods use that data field. Average the percentages then subtract from 100%. Lower percentages mean greater cohesion of data and methods in the class.

2. Methods are more similar if they operate on the same attributes. Count the number of disjoint sets produced from the intersection of the sets of attributes used by the methods.

5. CONCLUSION

This paper presented the results of an experiment where the impact of object-oriented design on resulting software quality attributes (defect density and rework) was empirically evaluated. The MOOD set of metrics was adopted to measure the characteristics of OO design. The results achieved so far allow inferring 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 [Abreu et al, 1995] embed 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 is a small step toward the understanding of how software designs affect resulting quality. A further validation experiment with a larger sample of projects is expected to be carried out. A replication of this experiment with a sample of C++ and Ada95 large-scale projects developed at the Software Engineering Laboratory (NASA Goddard Space Center) is expected to be done next year. The impact on other quality attributes like efficiency, portability, usability and functionality must also be assessed. The public availability of a tool to collect the adopted design metrics is expected to foster further experiments throughout the academic and industrial communities.

6. REFERENCES

(1) [Abbott et al, 1994] D. H. Abbott.; T. D. Korson; J. D. McGregor. "A proposed design complexity measure for object-oriented development". Clemson University, TR 94-105, April 1994.

(2) [Abreu et al, 1993] F. B. Abreu; R. Carapuça. "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework". Proceedings of AQUIS'93 (Achieving QUality In Software), Venice, Italy, October 1993; selected for reprint in the Journal of Systems and Software, Vol. 23 (1), pp. 87-96, July 1994.

(3) [Abreu et al, 1994] F. B. Abreu; 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.

(4) [Abreu et al, 1995] F. B. Abreu; M. Goulão; R. Esteves. “Toward the Design Quality Evaluation of ObjectOriented Software Systems”. Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.

(5) [Basili et al, 1995] 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.

(6) [Chidamber et al, 1994] 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 Transactions on Software Engineering, Vol. 20 (6), pp.476-493, June 1994.

(7) [Churcher et al., 1995] N. I. Churcher; M. J. Shepperd. "Comments on 'A metrics suite for object oriented design' ". IEEE Transactions on Software Engineering, Vol. 21 (3), pp.263-265, 1995.
(8) [Gamma et al, 1995] E. Gamma; R. Helm; R. Johnson; J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1995.
(9) [IEEE1061, 1990] Institute of Electrical and Electronic Engineers. "ANSI/IEEE P-1061/D21 - Standard for a Software Quality Metrics Methodology". 1990.
(10) [ISO9000/3, 1991] 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 (currently under revision).
(11) [ISO9126, 1991] International Organization for Standardization. "ISO/IEC 9126 - Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their use". ISO JTC1/SC7, 1991 (currently under revision).
(12) [ISO14598, 1995] International Organization for Standardization. "ISO/IEC 14598 - Information Technology - Software Product Evaluation". ISO JTC1/SC7, 1995 (currently in CD stage).
(13) [Li et al, 1994] W. Li; S. Henry. "Object-oriented metrics that predict maintainability". Journal of Systems and Software, Vol. 23 (2), pp.111-122, 1994.