Object Oriented Metrics Evaluation

S. Pasupathy[1] and R. Bhavani[2], Ph.D

[1.] Associate Professor, Dept. of CSE, FEAT, Annamalai University, Tamil Nadu, India.

[2.] Associate Professor, Dept. of CSE, FEAT, Annamalai University, Tamil Nadu, India.

ABSTRACT
This paper presents the results derived from the survey on metrics used in object oriented environments. The survey includes a small set of the most well known and commonly applied traditional software metrics which could be applied to object oriented programming and a set of object oriented metrics. In short, the metrics based assessment of a software system and measures taken to improve its design differ considerably from tool to tool. To support the case, we conducted an experiment with a number of commercial and free metrics tools. We calculated metrics values using the same set of standard metrics for three software systems of different sizes. These metrics were evaluated using object oriented metrics tools for the purpose of analyzing quality of the product, encapsulation, inheritance, message passing, polymorphism, reusability and complexity measurement. It defines a ranking of the classes that are most vital note down and maintainability. The results can be of great assistance to quality engineers in selecting the proper set metrics for their software projects and to calculate the metrics, which was developed using a sequential object oriented life cycle process.

Index: Software development, Object oriented programming, Object oriented metrics tool.

1. INTRODUCTION

Object-oriented design and development is becoming very popular in today's software development environment. Object oriented development requires not only a different approach to design and implementation, it requires a different approach to software metrics. Since object oriented technology uses objects and not algorithms as its fundamental building blocks, the approach to software metrics for object oriented programs must be different from the standard metrics set. Some metrics, such as lines of code and cyclomatic complexity, have become accepted as "standard" for traditional functional/ procedural programs, but for object oriented, there are many proposed object oriented metrics in the literature [1]. The question is, "Which object oriented metrics should a project use, and can any of the traditional metrics". This paper presents the possibility of using object-oriented software metrics for the automatic detection of a set of design problems.

It will illustrate the efficiency of this approach by discussing the conclusions of an experimental study that uses a set of three metrics for problem detection and applies them to three projects. These three metrics are "touching" three main aspects of object-oriented design, aspects that have an important impact on the quality of the systems – i.e. maintenance effort, class hierarchy layout and cohesion [2]. For each of these metrics we will present the design flaw that it may detect together with some of our experimental observations – and a possible redesign solution for that problem. Object-oriented (OO) metrics are measurements on OO applications used to determine the success or failure of a process or person, and to quantify improvements throughout the software process. These metrics can be used to reinforce good OO programming techniques, which leads to more reliable code.

One metric alone is not enough to determine any information about an application under development. Several metrics must be used in tandem to gain insight into improvements during a software process [3]. There are several software packages that can be used to determine the metrics on a software applications.

2. TYPES OF METRICS

The metrics presented hereinafter have been selected from metrics proposed specifically for object–oriented measurements and cannot be applied to another programming style. This is a small fraction of the most well-known metrics analyzed in various real time applications. The categories chosen to present the metrics are not defining a metrics classification but used simply to ease the presentation and sometimes a metric may fall in more than one category. The metrics presented are: class related metrics, method related metrics, encapsulation metrics, cyclometic complexity measuremetnt,inheritance metrics, metrics measure coupling and metrics measure general (system) software production [4,5]. The types of metrics are shown in table 1.

Table 1: Types of Metrics

S.NO	Metrics Types
01	Size Metrics
02	Class Metrics
03	Encapsulation Metrics
04	Complexity Metrics
05	Inheritance Metrics
06	Polymorphism Metrics
07	Message Passing Metrics
08	Coupling Metrics
09	Reuse Metrics
10	Quality Measurement

2.1 SIZE METRICS
This metrics is used to evaluate overall program size and specify the module wide metrics. Each metrics has different factors. The size oriented metrics are,

Lines of code (LOC): This measure provides a count of total number of lines in the module. It includes source lines, blank lines, comment lines.

Physical lines of code: This measure provides a count of total number of source lines in the module.

Number of statements: This measure indicates total number of statements in the module. It includes if, else, switch, case, while, do while, for statements.

Comment lines: This measure indicates total number of comment lines in a module.

Blank lines: This measure indicates total number of blank lines in a module.

Non-comment Non-blank (NCNB): This measure provides count of all lines that are not comments and not blanks.

Executable Statements (EXEC): This measure provides a count of executable statements regardless of number of physical lines of code.

2.2 CLASS ORIENTED METRICS

Classes, which are the central points of every object oriented language implement methods and define attributes. The class metrics address thus this aspect: their complexity can be expressed through methods and attributes and the way these entities behave. HNL Hierarchy nesting level also called depth of inheritance tree. The number of classes in superclass chain of class. In case of multiple inheritances, count the number of classes in the longest chain. Summarizes the overall class metrics.

2.2.1 Number of Class Measurement

- NA *Number of accessors*, the number of get/set - methods in a class.
- NAM *Number of abstract methods*.
- NC *Number of constructors*.
- NCV *Number of class variables*.
- NIA *Number of inherited attributes*, the number of attributes defined in all superclasses of the subject class.
- NIV *Number of instance variables*.
- NMA *Number of methods added*, the number of methods defined in the subject class but not in its superclass.
- NME *Number of methods extended*, the number of methods redefined in subject class by invoking the same method on a superclass.
- NMI *Number of methods inherited*, i.e. defined in superclass and inherited unmodified.
- NMO *Number of methods overridden*, i.e. redefined in subject class.
- NOC *Number of immediate children of a class*.
- NOM *Number of methods*, each method counts as 1. NOM = NMA + NME + NMO.
- NOMP *Number of method protocols*. This is Smalltalk - specific: methods can be grouped into method protocols.
- PriA *Number of private attributes*.
- PriM *Number of private methods*.
- ProA *Number of protected attributes.*
- ProM *Number of protected methods*.
- PubA *Number of public attributes*.
- PubM *Number of public methods*.
- WLOC *Lines of code*, sum of all lines of code in all method bodies of the class.
- WMSG *Number of message sends*, sum of number of message sends in all method bodies of class.
- WMCX *Sum of method complexities*.
- WNAA *Number of times all attributes defined in the class are accessed*.
- WNI *Number of method invocations*, i.e. in all method bodies of all methods.
- WNMAA *Number of all accesses on attributes*.
- WNOC *Number of all descendants*, i.e. sum of all direct and indirect children of a class.
- WNOS *Number of statements*, sum of statements in all method bodies of class.

2.2.2 Methods present in the class:

Methods can be seen as a flow of instructions which take input through parameters and which produce output. Methods can invoke other methods or access attributes. The method metrics are defined in this context.

- LOC *Lines of code* in method body.
- MHNL *Hierarchy nesting level* of class in which method is implemented.
- MSG *Number of message sends* in method body.
- NI *Number of invocations* of other methods in method body.
- NMAA *Number of accesses on attributes* in method body.
- NOP *Number of parameters* which the method takes.
- NOS *Number of statements* in method body.
- NTIG *Number of times invoked by methods non-local to its class*, i.e. from methods implemented in other classes.
- NTIL *Number of times invoked by methods local to its class*, i.e. from methods implemented in the same class.

2.2.3 Attributes present in the class*:*

Attributes are properties to classes. Their main function is to return their value when accessed by methods. The attribute metrics are defined in such a context.

- AHNL *Hierarchy nesting level* of class in which attribute is defined.
- NAA *Number of times accessed*. NAA = NGA + NLA.
- NCM *Number of classes having methods that access it.*
- NGA *Number of times accessed by methods non-local* to its class.
- NLA *Number of times accessed by methods local* to its class.
- NM *Number of methods accessing it.*

2.3 ENCAPSULATION METRIC

The encapsulation metrics evolves packaging (or binding together) of a collection of items.

- Low-level examples of encapsulation include records and arrays.
- Subprograms (e.g., procedures, functions, subroutines, and paragraphs) are mid-level mechanisms for encapsulation.
- In object-oriented (and object-based) programming languages, there are still larger encapsulating mechanisms, e.g., C++'s classes, Ada's packages, and Modula 3's modules. [Figure 6] Summarizes the Encapsulation metrics.

2.3.1 Objects Encapsulate

- knowledge of state, whether statically maintained, calculated upon demand, or otherwise,
- advertised capabilities (sometimes called operations, method interfaces, method selectors, or method interfaces), and the corresponding algorithms used to accomplish these capabilities (often referred to simply as methods),
- [in the case of composite objects] other objects,
- [optionally] exceptions,
- [optionally] constants, and
- [Most importantly] concepts.

In many object-oriented programming languages, encapsulation of objects (e.g., classes and their instances) is syntactically and semantically supported by the language. In others, the concept of encapsulation is supported conceptually, but not physically.

The types of complexity metrics are shown in table 2.

Encapsulation has two major impacts on metrics:

- the basic unit will no longer be the subprogram, but rather the object, and
- we will have to modify our thinking on characterizing and estimating systems.

2.3.2 Information Hiding is the suppression (or hiding) of details.

- The general idea is that we show only that information which is necessary to accomplish our immediate goals.
- There are degrees of information hiding, ranging from partially restricted visibility to total invisibility.
- Encapsulation and information hiding are not the same thing, e.g., an item can be encapsulated but may still be totally visible.

Information hiding plays a direct role in such metrics as object coupling and the degree of information hiding.

2.4 COMPLEXITY METRICS

Complexity is everywhere in the software life cycle: requirements, analysis, design, and of course, implementation is usually an undesired property of software because complexity makes software harder to read and understand, and therefore harder to change; also, it is believed to be one cause of the presence of defects. Summarizes the Complexity measurement metrics. In a use net debate surrounding Intelligent Design, the issue of measuring complexity kept coming up. Are there any good objective metrics for "complexity"? The complexity measured output shown in Figure 2.

METRIC	OBJECTIVE
Cyclomatic Complexity	Low
Lines of Code/Executable Statements	Low
Comment Percentage	~ 20 – 30 %
Weighted Methods per Class	Low
Response for a Class	Low
Lack of Cohesion of Methods Cohesion of Methods	Low High
Coupling Between Objects	Low
Depth of Inheritance	Low (trade-off)
Number of Children	Low (trade-off)

Table 2: Types Of Complexity Metrics.

All Methods In System

Name	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
AcceptClient...	1	0	9	64	30	1712.95	0.10	6	6	1	
action(java....	5	0	11	110	43	5026.93	0.20	4	7	1	
chatClient(ja...	1	0	14	98	39	2606.03	0.17	9	11	0	
chatServer()...	2	0	7	45	22	1089.38	0.07	5	4	0	
main(java.la...	1	0	3	25	19	707.99	0.04	3	2	0	
main(java.la...	1	0	2	18	16	370.29	0.02	3	1	0	
run() (JHawk...	3	0	5	36	22	778.00	0.05	1	3	0	
run() (JHawk...	11	0	38	236	60	29274.55	0.46	6	11	0	
setup() (JHa...	1	0	10	53	23	1100.02	0.08	2	3	4	

Figure 1: All Methods Available in Java Program and calculate Metrics Factor

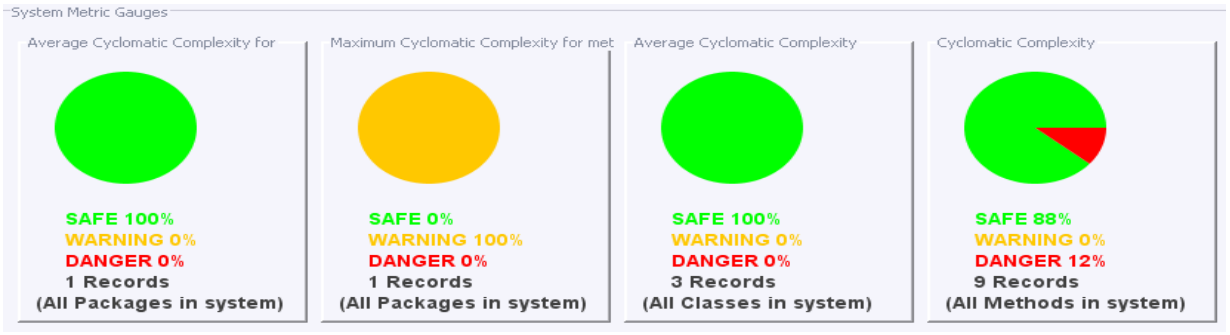

Figure 2: Complexity Measurement for Java program

2.5 INHERITANCE METRICS

The mechanism supports the class hierarchy design and captures the IS-A relationship between a super class and its subclass.

2.5.1 Types Available For Corresponding Metrics

- Dynamic inheritance
- Multiple inheritance

2.5.2 Types of Internal Metrics

- Average Degree of Understandability (AU) Metric
- Average Degree of Modifiability (AM) Metric
- Average Inheritance Depth (AID)
- Derive Base Ratio Metric (DBRM)
- Average Number of Direct Child (ANDC) Metric
- Average Number of Indirect Child (ANIC) Metric

2.6 MESSAGE PASSING METRICS

Message passing describes the act of communication between two or more computer processes (in the form of "messages"). A metric is a numerical value computed from a collection of data. Message Passing metrics deal with the measurement of Number of Message passing Iterations involved in software product or a process by which it is developed. A software product can be viewed as an abstract object that evolves from an initial statement of need to a finished software system, including source and executable code and the various forms of documentation produced during development. Ordinarily, the measurements of the software products and processes are studied and developed for use in modeling the development process [6]. In this Work two algorithms for estimation and Measurement of Messing passing, their metrics are then used to estimate/predict product costs and schedules and to measure productivity and product quality. Information gained from metrics can then be used in the management and control of the development process in order to improve results. Summarizes the overall method metrics.

2.7 REUSE METRICS

Reuse Ratio (U):

The reuse ratio (U) is given by U=number of super class/total number of class.

Specialization Ratio(S):

This ratio measures the extent to which a super class has captured abstraction. S=number of subclass/number of super class.

Average Inheritance Depth:
The inheritance structure can be measured in terms of depth of each class with in its hierarchy. Average inheritance depth =sum of depth of each class/number of class. Figure 3 and 4 represents class oriented metrics also specifies each selected class metrics.

2.8 QUALITY METRICS

Reusability: Reusability means reflects the presence of OO Design characteristics that allow a design to be reapplied to new problem without significant. Reusability formula= (-0.25*coupling) + (0.25*cohesion) + (0.5*messaging) + (0.5*design size).

Flexibility: Characteristics that allow the incorporation of change in a design. The ability of a design to be adapted to provide Functionality related capabilities. Flexibility formula= (0.25*encapsulation)-
(0.25*coupling)+(0.5*composition)+(0.5*polymorphism).

Understandability: The properties of the design that enable it to be easily learned and comprehend. Understandability formula= (-0.33*abstraction) + (0.33*encapsulation)-(0.33*coupling) + (0.33*cohesion)-(0.33*polymorphism)-(0.33*complexity)-(0.33*design size).

Functionality: The responsibilities assigned to the classes of design, which are made available by the classes through their public interfaces. Functionality formula= (0.12*cohesion) + (0.22*polymorphism) + (0.22*messaging)+ (0.22*design size) + (0.22*hierarchies)

Extendibility: It refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design. Extendibility formula = (0.5*Abstraction)-(0.5*coupling) + (0.5*inheritance) + (0.5*polymorphism).

Effectiveness: It refers to a design's ability to achieve the desired functionality and behavior using OO Design concepts. Effectiveness formula= (0.2*abstraction) +(0.2*encapsulation) + (0.2*composition) + (0.2*inheritance) + (0.2*polymorphism).

2.9 COUPLING METRICS

Coupling in software has been linked with maintainability and existing metrics are used as predictors of external software quality attributes such as fault-proneness, impact analysis, ripple effects of changes, changeability, etc. Many coupling measures for object-oriented (OO) software have been proposed, each of them capturing specific dimensions of coupling. This paper presents a new set of coupling measures for OO systems – named conceptual coupling, based on the semantic information obtained from the source code, encoded in identifiers and comments. A case study on open source software systems is performed to compare the new measures with existing structural coupling measures. The case study shows that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures; hence it can be used to complement the existing metrics.

2.9.1 Object Oriented Programming Coupling
Coupling between objects (CBO)

1) coupling = class x is coupled to class y iff x uses y's methods or instance variables (includes inheritance related coupling).
2) CBO for a class is a count of the number of other classes to which it is coupled.
3) High coupling between classes means modules depend on each other too much.
4) Independent classes are easier to reuse and extend.
5) High coupling decreases understandability and increases complexity.
6) High coupling makes maintenance more difficult since changes in a class might propagate to other parts of software.
7) Coupling should be kept low, but some coupling is necessary for a functional system.

2.9.2 COUPLING VERSUS COHESION

Coupling and Cohesion are the two terms which very frequently occur together. Together they talk about the quality a module should have. Coupling talks about the inter dependencies between the various modules while cohesion describes how related functions within a module are. Low cohesion implies that module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

3. SOFTWARE ENGINEERING METRICS ESTIMATION

In software estimation process involved various domains. Here are some observations [7,8,9]:

- A single software engineering metric in isolation is seldom useful. However, for a particular process, product, or person, 3 to 5 well-chosen metrics seems to be a practical upper limit, i.e., additional metrics (above 5) do not usually provide a significant return on investment.
- Although multiple metrics must be gathered, the most useful set of metrics for a given person, process, or product may not be known ahead of time. This implies that, when it is first begin to study some aspect of software engineering, or a specific software project, we will probably have to use a large (e.g., 20 to 30, or more) number of different metrics. Later, analysis should point out the most useful metrics.
- Metrics are almost always interrelated. Specifically, attempts to influence one metric usually have an impact on other metrics for the same person, process, or product[10].
- To be useful, metrics must be gathered systematically and regularly -- preferably in an automated manner.
- Metrics must be correlated with reality. This correlation must take place before meaningful decisions, based on the metrics, can be made[11].
- Faulty analysis (statistical or otherwise) of metrics can render metrics useless, or even harmful.
- To make meaningful metrics-based comparisons, both the similarities and dissimilarities of the

people, processes, or products being compared must be known.

- Those gathering metrics must be aware of the items that may influence the metrics they are gathering. For example, there are the "terrible H's," i.e., the Heisenberg effect and the Hawthorne effect.
- Metrics can be harmful. More properly, metrics can be misused.

3.1 Object-oriented software engineering metrics are units of measurement that are used to characterize:

- object-oriented software engineering products, e.g., designs, source code, and test cases,
- object-oriented software engineering processes, e.g., the activities of analysis, designing, and coding, and
- Object-oriented software engineering people, e.g., the efficiency of an individual tester, or the productivity of an individual designer. Summarizes the overall Performance Evaluation.

Classes

Name	No. Me...	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC
chatServer	2	0.00	1.50	13	0.11	1641.28	4	2	3	7	1	118.93	0	22
chatServ...	2	1.00	6.00	51	0.59	31116.56	5	3	3	16	0	85.43	0	78
chatClient	5	0.40	2.20	53	0.62	10927.62	15	10	4	32	1	107.22	0	86

Figure. 3: Methods Metrics for Selected Class

Methods for selected Class

Name	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
chatServer() ...	2	0	7	45	22	1089.38	0.07	5	4	0	11
main(java.la...	1	0	2	18	16	370.29	0.02	3	1	0	4

Figure. 4: Methods Metrics for Each Selected Class

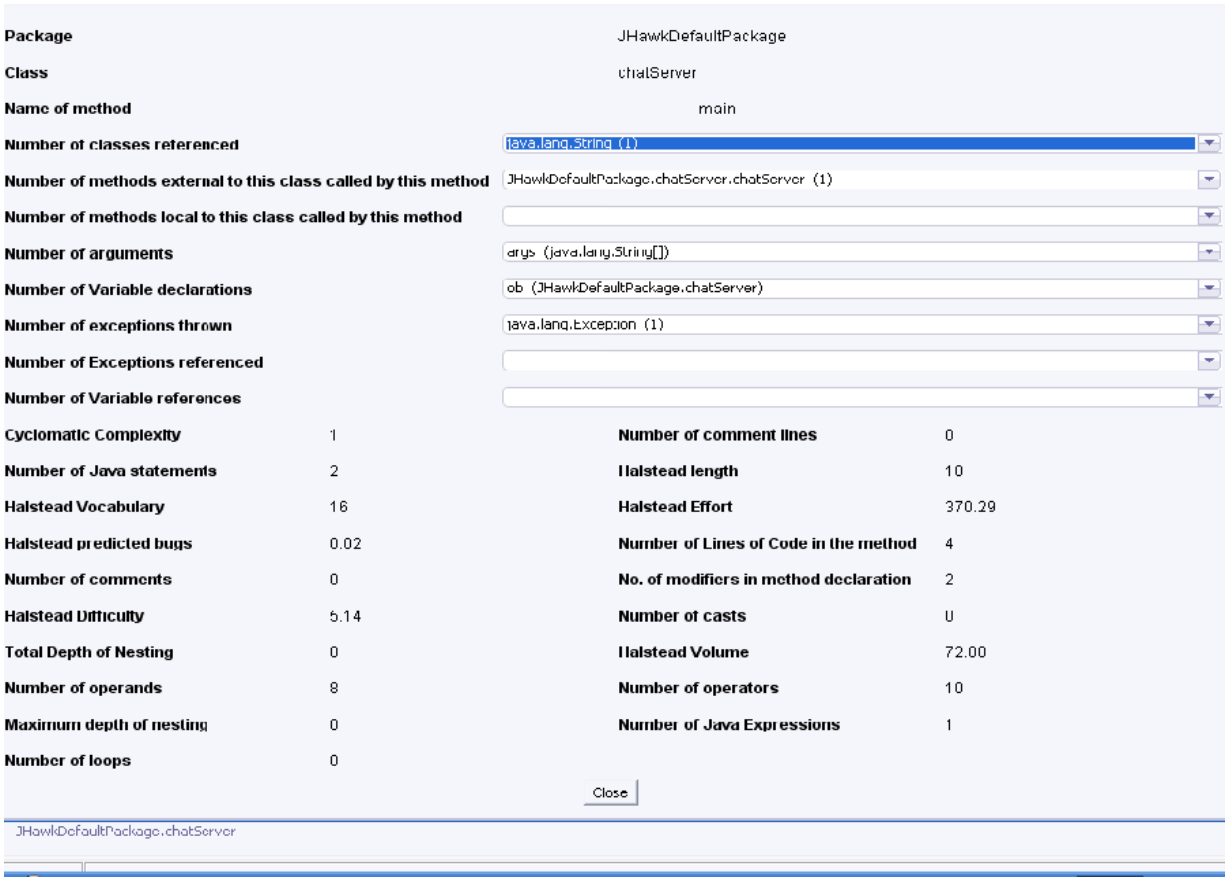

Figure. 5: Method Metrics for Different Parameters.

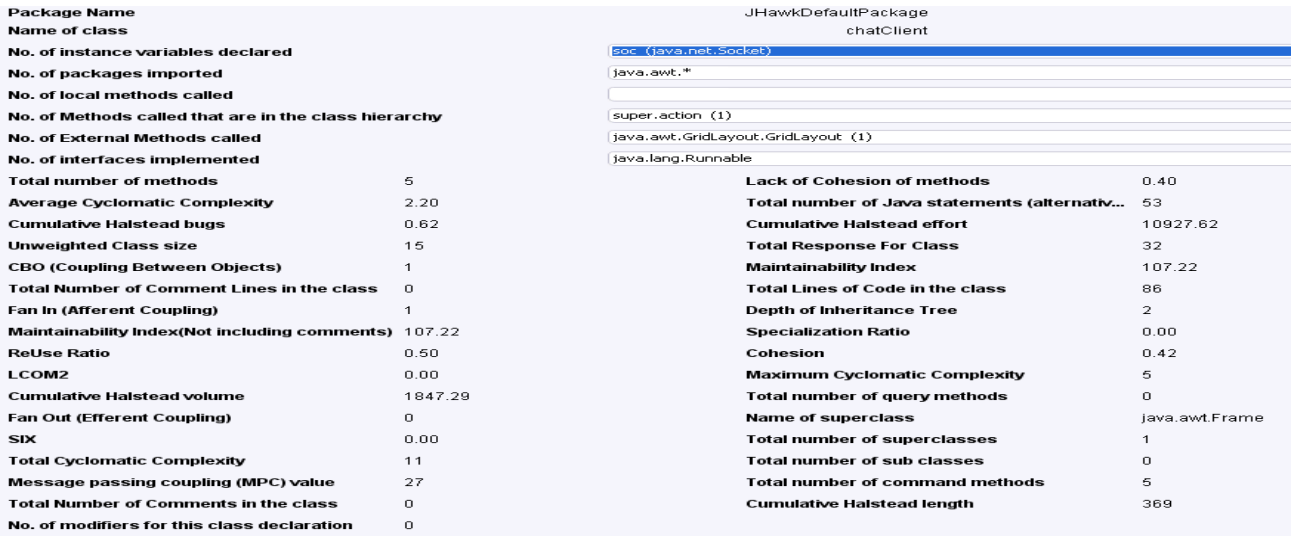

Figure.6: Encapsulation Object Oriented Metrics

5. CONCLUSION AND FUTURE SCOPE

The above results can be used in order to determine when and how each of the above metrics can be used according to quality characteristics a practitioner wants to emphasize. Make sure the software quality metrics and indicators they employ include a clear definition of component parts are accurate and readily collectible, and span the development spectrum and functional activities. Survey data indicates that most organizations are on the right track to making use of

metrics in software projects. For organizations which do not reflect "best practices", and would like to enhance their metrics capabilities, the following recommendations are suggested to Measure the "best practices" list of metrics more consistently across all projects. Focus on "easy to implement" metrics that are understood by both management and software developers, and provide demonstrated insight into software project activities.

6. REFERENCES

[1] A. Albrecht and J. Gaffney: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation; in IEEE Trans. Software Eng., 9(6), 2008, pp. 639-648.

[2] B. Bohem, Software Engineering Economics, Prentice Hall, Englewood Cliffs, 1981 [Briand et al 94] L. Briand, S. Morasca, V. Basili, Defining and Validating High-Level Design Metrics, Tech. Rep. CS TR-3301, University of Maryland, 2009.

[3] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Trans. Software Eng., 20/6), 2000, pp. 263-265.

[4] S. Morasca, Software Measurement: State of the Art and Related Issues, slides from the School of the Italian Group of Informatics Engineering, Rovereto, Italy, September 2008.

[5] J. Alghamdi, R. Rufai, and S. Khan. Oometer: A software quality assurance tool. Software Maintenance and Reengineering, 2009. CSMR 2009. 9th European Conference on, pages 190{191, 21-23}, March 2010.

[6] H. Bsar, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook, Oct. 2006.

[7] A. Albrecht: "Measuring application development productivity", in Proc. Joint SHARE/GUIDE/IBM Applications Development Symposium, Monterey, CA, 2007.

[8] L. Briand, S. Morasca, V. Basili, Property-Based Software Engineering Measurement, IEEE Trans. Software Eng. 22(1), 2000, pp. 68-85.

[9] S. Conte, H. Dunsmore, V. Shen, Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, CA.

[10] J. Stathis, D. Jeffrey, An Empirical Study of Albrecht's Function Points, in Measurement for Improved IT management, Proc. First Australian Conference on Software Metrics, ACOSM 93, Sydney, 2002, pp. 96 - 117.

[11] Boehm, Barry W., as quoted by Ware Myers, "Software Pivotal to Strategic Defense," IEEEComputer, January 2001.