Volume 4, Issue 1, January 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Object Oriented Metrics Estimation and Performance Matching

S.Pasupathy
Associate Professor, Dept.of CSE, FEAT ,
Annamalai University,
Tamil Nadu, India,

R.Bhavani, PhD.
Professor, Dept.of CSE,FEAT,
Annamalai University,
Tamil Nadu, India

Abstract: 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. The process provides a practical, systematic, start-to-finish method of selecting, designing and implementing software metrics. 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 chronological object oriented life cycle process.

Index: Object Oriented Programming Concepts, Reusability, Performance Evaluation, Matching.

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. 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. We 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 [1,2,3]. 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. There are several software packages that can be used to determine the metrics on a software applications.

2. TYPES OF METRICS

The first rule of quantitative software evaluation is that if we collect or compute numbers we must have a specific intent related to understanding, controlling or improving software and its production.
This implies that there are two broad kinds of metrics: product metrics that measure properties of the software products; and process metrics that measure properties of the process used to obtained these products [3,4].

Product metrics include two categories: external product metrics cover properties visible to the users of a product; internal product metrics cover properties visible only to the development team. External product metrics include:

- Product non-reliability metrics, assessing the number of remaining defects.
- Functionality metrics, assessing how much useful functionality the product provides.
- Performance metrics, assessing a product's use of available resources: computation speed, space occupancy.
- Usability metrics, assessing a product's ease of learning and ease of use.
- Cost metrics, assessing the cost of purchasing and using a product.

© 2014, IJARCSSE All Rights Reserved

2.1 Internal Product Metrics Include

- Size metrics, providing measures of how big a product is internally.
- Complexity metrics (closely related to size), assessing how complex a product is.
- Style metrics, assessing adherence to writing guidelines for product components (programs and documents).

2.2 Process Metrics Include

- Cost metrics, measuring the cost of a project, or of some project activities (for example original development, maintenance, documentation).
- Effort metrics (a subcategory of cost metrics), estimating the human part of the cost and typically measured in person-days or person-months.
- Advancement metrics, estimating the degree of completion of a product under construction.
- Process non-reliability metrics, assessing the number of defects uncovered so far.
- Reuse metrics, assessing how much of a development benefited from earlier developments.

2.3 Internal And External Metrics

The second rule is that internal and product metrics should be designed to mirror relevant external metrics as closely as possible. Clearly, the only metrics of interest in the long run are external metrics, which assess the result of our work as perceptible by our market. Internal metrics and product metrics help us improve this product and the process of producing it. They should always be designed so as to be eventually relevant to external metrics.

Object technology is particularly useful here because of its seamlessness properties, which reduces the gap between problem structure and program structure (the "Direct Mapping" property). In particular, one may argue that in an object-oriented context the notion of function point, a widely accepted measure of functionality, can be replaced by a much more objective measure: the number of exported features (operations) of relevant classes, which requires no human decision and can be measured trivially by a simple parsing tool [7,8].

2.4 Designing Metrics

The third rule is that any metric applied to a product or project should be justified by a clear theory of what property the metric is intended to help estimate. The set of things we can measure is infinite, and most of them are not interesting. For example I can write a tool to compute the sum of all ASCII character codes in any program, modulo 53, but this is unlikely to yield anything of interest to product developers, product users, or project managers.

A simple example is a set of measurements that we performed some time ago on the public-domain EiffelBase library of fundamental data structures and algorithms, reported in the book Reusable Software. One of the things we counted was the number of arguments to a feature (attribute or routine) over 150 classes and 1850 features, and found an average of 0.4 and a maximum of three, with 97% of the features having two or less. We were not measuring this particular property in the blind: it was connected to a very precise hypothesis that the simplicity of such interfaces is a key component of the ease of use and learning (and hence the potential success) of a reusable component library. These figures show a huge decrease as compared to the average number of arguments for typical non-O-O subroutine libraries, often 5 or more, sometimes as much as 10. (Note that a C or Fortran subroutine has one more argument than the corresponding O-O feature.)

2.5 Calibrating Metrics

More precisely, the fourth rule is that most measurements are only meaningful after calibration and comparison to earlier results.This is particularly true of cost and reliability metrics. A sophisticated cost model such as COCOMO will become more and more useful as you apply it to successive projects and use the results to calibrate the model's parameters to your own context. As you move on to new projects, you can use the model with more and more confidence based on comparisons with other projects.

Similarly, many internal product metrics are particularly useful when taken relatively. Presented with an average argument count measure of 4 for your newest library, you will not necessarily know what it means -- good, bad, irrelevant? Assessed against published measures of goodness, or against measures for previous projects in your team, it will become more meaningful. Particularly significant are outlying points: if the average value for a certain property is 5 with a standard deviation of 2, and you measure 10 for a new development, it's probably worth checking further, assuming of course (rule 2) that there is some theory to support the assumption that the measure is relevant. This is where tools such as the Monash suite can be particularly useful.

3. METRICS AND THE MEASURING PROCESS

The fifth rule is that the benefits of a metrics program lie in the measuring process as well as in its results. The software metrics literature often describes complex models purporting to help predict various properties of software products and processes by measuring other properties. It also contains lots of controversy about the value of the models and their predictions. But even if we remain theoretically skeptical of some of the models, we shouldn't throw away the corresponding measurements [10,11]. The very process of collecting these measurements leads (as long as we confine ourselves to measurements that are meaningful, at least by some informal criteria) to a better organization of the software process and a better understanding of what we are doing. This idea explains the attraction and usefulness of process

© 2014, IJARCSSE All Rights Reserved

guidelines such as the Software Engineering Institute's Capability Maturity Model, which encourage organizations to monitor their processes and make them repeatable, in part through measurement.

3.1 Metrics Description
3.1.1 Project Wide 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.

3.1.2 Module Wide Metrics

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.

3.1.3 C.K. Metrics Object Oriented Metrics
Weighted Methods per Class: It is an average of the number of methods within the classes of the software system for which the metrics are being collected. Average Number of Methods per Class reflects the degree of responsibility attributed to a class i.e., it is a predictor of how much time and effort is required to develop and maintain the class. A large number of methods mean a greater potential impact on its derived class, since the derived class inherits all the method of the base class. A class in a system should not be overloaded with large number of functions.

Depth of Inheritance Tree (DIT): It is a class level design metric. The DIT for a particular class calculates the length of the inheritance chain from the root to the level of this class. Deep trees promote reuse because of method inheritance. High DIT increase faults. Most fault-prone classes are the ones in the middle of the tree. A recommended DIT is five or less.
Number of Children: It indicates the number of immediate subclasses of a class. NOC is counted via the inherited statement. NOC is equal to number of immediate child classes derived from a class via the inherited statement. It measures the breadth of a class hierarchy. High NOC
indicates high reuse, fewer faults.

Coupling between object Classes: It is the number of classes to which a class is coupled. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. High CBO is undesirable. High coupling indicates fault-proneness. Only methods an have children in a design. variable references are counted. Other types of reference, such as use of constants, calls to API declared, handling of events, use of user defined types, and object instantiations are ignored. If a method call is polymorphic (either because of overrides or overload) all the classes to which the call can go are included in the coupled class.

Response for a Class (RFC): It is a class level design metric. Response for a class is a set of methods that can potentially be executed in response to a message received by an object of that class. Response for a class defines the number of methods available in the class. The number of available methods in the class is the sum of the number of local

© 2014, IJARCSSE All Rights Reserved

methods in the class and the number of methods called by the local methods. Larger values for this metric imply that more testing is required for that class because of the increased coupling issues.

Lack of Cohesion (LCOM): It is a class level design metric. It measures the number of disjoint sets of local methods. Disjoint sets are a collection of sets that do not intersect with each other. Any two methods in one disjoint set access at least one common local instance variable.

Structural Complexity: Complexity is calculated by number of predicated node or decision plus one. Decision include conditional statements like if, else, select, for, do loop, while and end loop.

Hierarchical Model: Hierarchical Model is used to determine the overall quality of Object Oriented system by evaluating the structural and the behavioral design properties of classes and their relationship. Design properties include encapsulation, polymorphism, modularity, coupling,
cohesion, etc.,.

Abstraction: Abstraction is a measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.

Cohesion: Assesses the relatedness of methods and attributes in a class, strong overlap in the method parameters and attributes types in an indication of strong cohesion.

Complexity: Complexity is a measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Composition: Measures the "part-of", "has","consist- of" or "part-whole" relationship, which are aggregation relationships in an object –oriented design.

Coupling: Define the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in
order for that to function correctly.

Design Size: A measure of the number of classes used in a design.

Encapsulation: Defined as enclosing of data and behavior within a single construct .In Object Oriented design the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.

Hierarchies: Hierarchies are used to represent different generalization –specialization concepts in a design.It is the count of the number of non-inherited classes that have children in a design.

Inheritance: A measure of "is –a" relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.

Messaging: A count of number of public methods that is available as services to other classes. This is a measure of services that a class provides.

Polymorphism: Polymorphism refers to the ability to substitute objects whose interfaces match for one another at run time. It is a measure of services that are dynamically determined at run time in an object.

3.1.4 Quality Attributes

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).

© 2014, IJARCSSE All Rights Reserved

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).

4. PERFORMANCE MATCHING

4.1 Object Oriented Metrics in Software Engineering Approach

"Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area.This demand has spurred the provision of a number of new and/or improved approaches to softwaredevelopment, with perhaps the most prominent being object-orientation (OO). In addition, the focus onprocess improvement has increased the demand for software measures, or metrics".

4.2 Applying And Interpreting Object Oriented Metrics

"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."

4.3 Design Principles and Design Patterns

"What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications1. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. "

4.4 Using Educational Tools For Teaching Object Oriented Design And Programming

"The development of software systems is a complex process which requires a diverse set of skills and expertise. The Object Oriented programming paradigm has been proven to better organize the inherent complexity of software systems, than the traditional procedural paradigm. Hence, Object Oriented (OO) is becoming the dominant paradigm in the recent years. The software industry is placing increasing emphasis on newer, object-oriented programming languages and tools, such as Java. It is highly interested for software engineers capable to analyze and develop systems using the OO programming paradigm."

4.5 Quality Metrics Tool For Object Oriented Programming

"Metrics measure certain properties of a software system by mapping them to numbers (or to other symbols) according to well-defined, objective measurement rules. Design Metrics are measurements of the static state of the project's design and also used for assessing the size and in some cases the quality and complexity of software. Assessing the Object Oriented Design (OOD) metrics is to predict potentially fault-prone classes and components in advances"

4.6 Message Creation Overhead And Performance

Since all messages and parameters must possess particular meanings to be consumed (i.e., result in intended logical flow within the receiver), they must be created with a particular meaning. Creating any sort of message requires overhead in either CPU or memory usage. Creating a single integer value message (which might be a reference to a string, array or data structure) requires less overhead than creating a complicated message such as a SOAP message. Longer messages require more CPU and memory to produce. To optimize runtime performance, message length must be minimized and message meaning must be maximized.

4.7 Message Transmission Overhead And Performance

Since a message must be transmitted in full to retain its complete meaning, message transmission must be optimized. Longer messages require more CPU and memory to transmit and receive. Also, when necessary, receivers must reassemble a message into its original state to completely receive it. Hence, to optimize runtime performance, message length must be minimized and message meaning must be maximized.

4.8 Message Translation Overhead And Performance

© 2014, IJARCSSE All Rights Reserved

Message protocols and messages themselves often contain extra information (i.e., packet, structure, definition and language information). Hence, the receiver often needs to translate a message into a more refined form by removing extra characters and structure information and/or by converting values from one type to another. Any sort of translation increases CPU and/or memory overhead. To optimize runtime performance, message form and content must be reduced and refined to maximize its meaning and reduce translation.

4.9 Message Interpretation Overhead And Performance

All messages must be interpreted by the receiver. Simple messages such as integers might not require additional processing to be interpreted. However, complex messages such as SOAP messages require a parser and a string transformer for them to exhibit intended meanings. To optimize runtime performance, messages must be refined and reduced to minimize interpretation overhead.

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. The representational theory of measurement should be the base of any kind of measurement, included that of software. It defines the measure as a mapping between an empirical relation system and a formal one and it gives the representation condition as a necessary and sufficient condition for a measure to be valid. Using measurement theory, Fenton has demonstrated that a single measure for complexity cannot exist and that axioms are inconsistent. We have described in detail six metrics, chosen among the ones most widely known and used. They are relative to different phases of software development. In the requirements phase, we can use Function Points to measure the functionality starting from the user requirements. In the high-level design phase, the suite of metrics can be used: we have concept of measures for cohesion and coupling, which are important attributes of design.

A number of object oriented metrics have been proposed in the literature for measuring the design attributes such as inheritance, polymorphism, message passing, complexity, Hiding Factor, coupling, cohesion, reusability etc,. In this paper, A metrics program that is based on the goals of an organization will help communicate, measure progress towards, and eventually attain those goals. People will work to accomplish what they believe to be important. Well-designed metrics with documented objectives can help an organization obtain the information it needs to continue to improve its software products, processes, and services while maintaining a focus on what is important. A practical, systematic, start-to-finish method of selecting, designing, and implementing software metrics is a valuable aid. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class. This metrics set can be applied on various projects and evaluate and compare the performance of the code using object oriented paradigm. While in the past the focus in research was on inventing new metrics, now the focus is more on measurement theory, in particular on the definition of new validation frameworks or of new set of axioms.

REFERENCES

[1] Kaur Amandeep, Singh Satwinder, K. Kahl. “Evaluation and Metrication of Object Oriented System”, International Multi Conference of Engineers and Cmputer Scientists, 2009 vol. 1.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

© 2014, IJARCSSE All Rights Reserved

[10] L. Briand, S. Morasca, V. Basili, Property-Based Software Engineering Measurement, IEEE Trans. Software Eng. 22(1), 2000, pp. 68-85.
[11] 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.
[12] Boehm, Barry W., as quoted by Ware Myers, "Software Pivotal to Strategic Defense," IEEE Computer, January 2001.
[13] Campbell, Luke and Brian Koster, "Software Metrics: Adding Engineering Rigor to a Currently Ephemeral Process," briefing presented to the McGrummwell F/A-24 CDR course, 2003.
[14] V. Basili, Qualitative Software Complexity Models: a Summary, in Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos, CA, 2004.
[15] E. Weyuker, Evaluating Software Complexity Measures, IEEE Trans. Software Eng., 14(9), 2002, pp. 1357-1365.
[16] H. Zuse, Software Complexity: Measures and Methods, Walter de Gruyter, Berlin, 2006.
[17] Ada and C++: A Business Case Analysis, Office of the Deputy Assistant Secretary of the Air Force, Washington, DC, June 1999.
[18] Albrecht, A.J., "Measuring Application Development Productivity," Proceedings of the IBM Applications Development Symposium, Monterey, California, October 2005.
[19] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 2006.
[20] M. Xenos, D.Stavrinoudis, K.Zikouli and D. Christodoulakis, "Object Oriented Metrics – A Survey", Proceeding of the FESMA 2000, Federation of European Software Measurement Association, Madrid. Spain, 2006.

© 2014, IJARCSSE All Rights Reserved