Object-Oriented Metrics: An Overview*

David Bellin, Ph.D.
Manish Tyagi
Maurice Tyler

Computer Science Department

North Carolina A & T State University
Greensboro, NC 27411-0002
910-334-7245
Fax: 910-334-7244
dbellin@ncat.edu

Abstract

Object-Oriented Analysis and Design (OOAD) techniques appear to be at the forefront of software engineering technologies. Nevertheless, as with the introduction of any relatively new technique, there is a tendency for people to attempt to maximize efficiency without always having a corresponding factual basis for their actions. This paper discusses important "bullets of measure" that should be taken into consideration during and after the development of an Object-Oriented System, particularly as it pertains to the static analysis of OO source code. The proposed metrics are consistent with the suggestions of many individuals who are well known for their experience.

OOPSLA CATEGORY: Research

OOPSLA TOPIC AREAS: language design & implementation; software engineering practices; principles and theory.

*Research supported by Grant from IBM SWS, Cary Laboratory.

1. Introduction

In designing a complex software system, Object-Oriented Analysis and Design techniques provide many benefits. These benefits include, but are not limited to, the following: reusability, decomposition of a problem into easily understood objects, and the facility of future modifications and enhancements. Despite all of its benefits, the OOAD software development life cycle is by no means less difficult than the typical procedural approach. In fact, it has become even more complicated. Many software engineers are finding it troublesome to make the transition into this new way of thinking. In a variety of ways it is orthogonal to the traditional procedural approach. Therefore, it is necessary to provide dependable guidelines that one may follow to help ensure good OO programming practices and furnish reliable code. One of the most difficult problems in the software engineering realm is producing software that is within budget, complete, on time, and that meets the requirements of the end user [1]. This paper presents a compilation of Object-Oriented programming metrics that we think should be considered. In this instance, we mean metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system. Metrics are a useful means for monitoring progress, attaining more accurate estimations of project milestones, and developing a software system that contains minimal faults. Software metrics also help to quantify the problem complexity, design complexity, and the program or product complexity [4].

Our particular interest is in metrics which can be applied to analyze source code as an indicator of quality. The source code could be Smalltalk, C++, Eiffel, or any other OO language. We are engaged in research which will result in validating metrics in the Smalltalk environment.

2. Problem Statement

One of the classical problems of software development is figuring out how to predict the resources necessary to complete a project while at the same time maximizing the quality and efficiency of the effort. The development of metrics is a procedure that has been used for previous generations of software technology. Since OOAD is a relatively new technology as compared to structured analysis and design, most work in this area is relatively new [2]. However, the metrics which were useful for evaluating procedural programs may not necessarily apply to software developed using Object-Oriented languages. For example, the Lines of Code (LOC) metric which may have been applicable in algorithmic-based programming code is not so useful in determining the quality of Object-Oriented code. In addition, metrics such as Lines of Code used on conventional source code are generally criticized for being without solid theoretical basis [5]. As a result, there exists no set of rules or standards against which today's Object-Oriented software engineer can measure the development of an OO-based system. As a result, a new set of metrics must be cultivated in order to satisfy the needs of OO Analysis and Design practices.

3. Suggested Solution

This paper presents an overview of a compilation of Object-Oriented programming metrics that we think should be considered throughout the OO development life cycle. We have divided the set of metrics into three main groups. *Group A* consists of the statistical aspects of OO design captured via a static analysis of the source code. *Group B* focuses on code reuse and how it may effect the OO development process. *Group C* investigates the "human" aspect of OO programming.

3.1 Group A

This group contains metrics that count the important structures in Object-Oriented code, such as methods, objects, and global variables. The main purpose of group A is to assess the relationships amongst the various building blocks of an Object-Oriented software system. Some of the metrics that should be collected in this group are:

Number of Methods: This metric could be important in judging the complexity of each object. It may be concluded that the more methods a class contains, the higher the complexity of the class. In addition, collecting and studying this metric allows determination of the ideal complexity of a class and establishment of a standard.

Number of Classes: This metric may give an idea of the overall complexity of a system. It could be important to determine, after enough data are collected from various completed projects, a relationship between the complexity of the problem domain and the number of classes.

Number of Messages: By counting the number of messages a class sends, one can determine the communication behavior of a class. This will help in analyzing the idea of coupling in classes.

If a class can respond to a wide variety of messages from a number of other classes, then it is possible to conclude that this particular class has tight coupling with respect to other classes. A tightly coupled class design is not a favorable feature as far as code reuse, testing, or debugging is concerned.

Number of Receiving Classes (Servers): If there are many server classes in a system then it is probable that many of the classes are performing very small functions within the system. This could be a hint that the system might have been constructed using many classes, which is generally favorable.

Number of Sender Classes (Actors): Using too many sender classes can have the same effect as having too many receiver classes. An ideal ratio of actors to servers should be derived. However, this ratio may vary for differing application domains.

Number of Agent Classes: Agent classes contain the properties of both actors and servers. Therefore, if a system contains an abundance of agent classes, there may be too much message passing amongst the various classes. This will result in a tightly coupled system and drastically reduce the amount of reusable code. In addition, the system could be more difficult to debug should a problem arise.

Number of Global Variables in each Class: This count would be helpful in determining the inheritance relationship shared among different classes. If there are a large number of global variables, the classes lower in the hierarchy can inherit many characteristics of the classes that are "higher up" in the hierarchy. However, too many global variables can destroy the independence of a class. Thus, a reasonable figure for this metric is also an important determinant in Object-Oriented code design quality.

Number of Levels in the Class Hierarchy Tree: By measuring the depth of the hierarchy tree, one will be able to ascertain the level of class abstraction. This metric may be useful when one is trying to decide upon the proper level of specialization and dependence amongst similar classes in a system.

Number of Leaves in the Class Hierarchy Tree: Measuring the breadth of a class hierarchy tree is as important as knowing the depth. This metric may shed additional light on the inheritance relationship between superclasses and subclasses. It may be a significant indicator of overall complexity.

Ratio Between Depth and Breadth: The ratio between the depth and breadth of a class hierarchy tree may be more important than measuring the individual number of each. Through developmental experience, one may derive a suitable ratio between the two.

Ratio of Methods/Class: By measuring the number of methods per class, one will be able to determine the complexity of each class. For example, if a particular class contains many methods, then it can be concluded that the class is complex.

Ratio of Private/Public Methods: This metric is helpful in determining the "trade-off" between data abstraction and the inheritance. This "trade-off" figure can be adjusted to comply with various OO programming languages and application domains.

Ratio of Abstract/Instantiated Classes: By noting the ratio of abstract classes to instantiated classes one may derive an idea of the hierarchical relationships within a system. For instance, having many abstract classes versus instantiated classes may indicate a hierarchical structure rich in inheritance.

Ratio of Lines of Code/Method: This metric might be important in determining the complexity of each method. It is generally desirable to keep the number of lines per method to a relatively small number. Thus far, the optimum number for a specific language is a matter of speculation.

Ratio of Lines of Code/Comment: This ratio may vary drastically depending upon the designer(s) of the system. It is a good programming practice to document well all aspects of a system. In addition, commenting facilitates the reuse and modification of code. However, too much commenting may actually result in a decrease in utility.

3.2 Group B

This group of metrics focuses on the reusability. Code reuse is one of the most desirable attributes of the Object-Oriented paradigm. It yields versatility and contributes to an efficient, cost-effective means of producing viable software systems. It has been determined that the more reused classes a system contains, the more reliable the finalized system will be. Subsequently, the classes reused have already been fully tested, hence there will be a decrease in the need for program maintenance. Unfortunately, the creation of new classes does not provide such benefits. Two important metrics that can be covered by this group are:

Number of Classes Reused: This consists simply of keeping track of the number of unmodified, pre-coded classes that are used by the “parasite” system. In general, if a system contains an abundance of reused classes, then it is likely that the system will be more reliable. This is because prefabricated classes, often contained in a reuse library, are thoroughly tested before being made readily accessible. As a result, the number of faults contained within the resulting system is minimized.

Percent of Reused Classes Modified: Once a reused class is modified, it may no longer possess the trait of reliability. Therefore, it is wise not only to know the number of classes reused, but to have an idea of how many reused classes were modified. It is suggested that any modified, reused class should be subjected to additional testing before it is implemented.

3.3 Group C

The previous metrics delved into the statistical realm and do not encompass the subjective aspects of Object-Oriented Analysis and Design techniques. The "human" factor is often overlooked, yet it is one of the most important components in the design of an Object-Oriented system. The popularity of anthropomorphic techniques such as CRC cards is evidence of this side of OO development. The only dilemma is that it is very difficult to measure subjective factors. How does one go about appraising the quality of an abstraction in an OO system? How can one know if a given class or object is well designed? Booch suggests five meaningful metrics [3]:

Coupling: In an ideal system weak coupling, with regard to objects and classes breadth wise on a hierarchy tree, is preferable. However, strong coupling is desirable among superclasses and subclasses.

Cohesion: Coincidental cohesion, in which entirely unrelated abstractions are put into the same category, is least desirable. On the other hand, functional cohesion, which is just the opposite of coincidental cohesion, is most desirable.

Sufficiency: Sufficiency is needed to ensure that a class or module encompasses components that make it useful to the overall system.

Completeness: An abstraction is complete when the interface of the class or module captures all of the meaningful characteristics of the abstraction.

Primitiveness: Booch maintains that, "Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction."

Even with these metrics, quantifying abstractions still seems a little "fuzzy". The task still remains highly subjective in nature. After all, we are only human and often have differing opinions. Therefore, to help ensure the quality of an OO system, it is suggested that developer(s) consider dispersing their projects among individuals who are considered experienced in this area.

We suggest that an additional input in determining the validity of OO source code metrics will be the opinions of known domain experts. Therefore, we intend to correlate static analysis metrics against recognized domain expert analysis of the same source code.

4. Future Studies

In the near future, our research team plans to apply these metrics to various software systems and assess their importance. This will be a four-step process. First, static analysis of the source code will collect metrics along each of the parameters outlined in this overview paper. Second, cooperating vendors will provide quality data from customers using the products whose source code has been analyzed. Third, recognized coding experts will provide subjective evaluations of the source code. Finally, statistical analysis will provide correlations (both high and low) between our suggested metrics set and the “real world”. We argue that this will be a significant validation that will indicate which OO metrics contribute to product quality. Such metrics will have obvious utility to all OO developers. Our first language focus will be Smalltalk.

5. Conclusion

The metrics supplied in this paper will help in analyzing any Object-Oriented system with respect to the four major elements of the Object-Oriented Analysis and Design paradigm: Abstraction, Encapsulation, Modularity, and Hierarchy. For instance, given such measures as the number of classes and the number of objects, it is more simpler to see inherent OO ideas such as modularity and abstraction. This will assist in developing a clearer picture of a system with regard to whether the notion of OOD has been fully applied to the best of its capabilities. Moreover, they will be useful in estimating any shortcomings of a similar nature that can be rectified in future systems. In addition, these standards of measure will provide enough information to derive some mathematical relations between a good and inferior Object-Oriented software system. Likewise, these mathematical relations can be used to analyze OO code and give some insight into the quality of the different abstractions. However, all of the metrics listed may not be useful for different varieties of OO languages. Nevertheless, some should hold true for all classes of Object-Oriented languages. It should be noted that any mathematical relationship derived can only be useful if it is cross-referenced against some practical preexisting data. Once correlated and used as a guide for certain segments of the software development process, these metrics should be cross-referenced again and this feedback cycle should continue until there is a perfect correlation between these mathematical estimates and actual quality data available from the end-user. These metrics are not meant to restrict the creativity of Object-Oriented software developers. They are merely suggested measures that may increase efficiency and reliability in Object-Oriented Analysis and Design.

References

1. Barnes, G.M. and Swim, B.R. *Inheriting Software Metrics*. Journal of Object-Oriented Programming (November - December 1993), 27-34.
2. Berard, E.V. *Essays on Object-Oriented Software Engineering*. Prentice-Hall, Englewood Cliffs, N.J., 1993.
3. Booch, G. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Redwood City, C.A., 1994.
4. Samadzadeh and Nandakumar, K. *A Study of Software Metrics.* The Journal of Systems and Software 16, 3 (November 1991), 229-234.
5. Vessey, I. and Weber R. *Research on Structured Programming: An Empiricist's Evaluation.* IEEE Transactions on Software Engineering 10, 4 (), 394-407.