Journal of Computer Science 2 (12): 870-874, 2006
ISSN 1549-3636

Evaluating the Effect of Inheritance on the Characteristics of Object Oriented Programs

[1]Thabit Sultan Mohammed and [2]Hayam K. Mustafa
[1]Software Engineering Department, Faculty of Science and IT, Al-Zaytoonah University, Amman-Jordan
[2]Computer Information Systems Department, Faculty of Science and IT
Al-Zaytoonah University, Amman-Jordan

Abstract: This paper considers a fact that software measures, which many of them were defined many years ago, are still not widely used in software industry, and therefore some additional insights will be gained by investigating Halstead's metrics and use them to propose more software metrics. Since the object oriented approach was considered an active technology for achieving high quality software, three metrics for evaluating the extent to which the inheritance property was invested in the object oriented programs are proposed in this paper. The first proposed metric was "the inheritance ratio" which studies the reduction in the program volume as a result of using the inheritance property with respect to the volume of the same program when it was written as functional oriented. The second metric "the inheritance level" points at the reduction achieved in program volume when the inheritance property was implemented in different levels. The third metric "effort ratio" relates to the reduction in developer's effort during the process of program development.

Keywords: inheritance, object oriented metrics, software science, software measurement.

INTRODUCTION

Producing low-cost, high quality software is highly desirable in major software development projects. One of the most important activities of process improvement is the ability to measure the process. DeMarco in [1] has said "you cannot control what you can not measure". Software metrics are therefore important and can be used as quality indicators to help in risk management by providing means to identify risky parts at early stages of the software design. They can also help managers to prioritize their decisions, quantify improvements in the process, and assess failure and success.

Halstead's metrics, or what are commonly referred to as 'software science' [2], are among the most widely quoted software measures. These metrics were proposed by Maurice Halstead as a means of determining quantitative measures directly from the operands and operators in the program. Although Halstead metrics are most often used as maintenance metrics, they are also useful during software development to asses code quality. Researchers have used Halstead's metrics for evaluation in many examples. These metrics are used to evaluate student programs[3] and query language[4], to measure software written for real time switching system[5], to measure functional programs, to incorporate software measurements into a compiler[6] and to measure open sources software[7].

The objective of Halstead's metrics is to measure the basic program characteristics such as; length, vocabulary, volume, level, difficulty, effort and time. Some researchers have extends the work on more characteristics relating to the object oriented techniques [8] such as; average class size, average method size, and polymorphism.

The metrics presented in this paper tends to be compact by concentrating specifically on the effect of implementing the inheritance property in object oriented programs, while covering the most important of basic program characteristics without excluding what is referred to as developer attributes "the programming effort". The effort according to Halstead is based on program difficulty and reflects the time required for developing a program. In fact, not many studies have considered the impact of this metric on software quality [8].

The model presented in our paper is directed towards analyzing open source software programs written in C++ language. In[9], Halstead metrics are calculated for Java language programs not as open source but at the level of Java byte code, where it was assumed that some flexibility in analysis will be granted since much commercial software is distributed as byte code only.

It is important to distinguish between the design principles of object oriented approach and the

Corresponding Author: Dr. Thabit Sultan Mohammed, Al-Zaytoonah University, Faculty of Science and IT, Software Engineering Department, P.O. Box 130 Amman- (11733) Jordan

design principles of functional oriented approach, in order to clarify many aspects of the object orientation and allow better quality and administration management.

Pressman[10] points at five situations, where the object oriented metrics can be configured.

- **Localization**: It relates to the tendency of information in being centralized.
- **Encapsulation:** Encapsulation means that objects include their data and attributes.
- **Information Hiding:** Information hiding means to hide object characteristics (data and attributes).
- **Inheritance**: This property allows the possibility of deriving a new class and giving it the attributes of a class or more (partially or as a whole).
- **Object Abstraction Technique**: This technique allows the designer to concentrate only on the basic and necessary details of certain parts of programs.

The next section of this paper presents a table containing the equations governing the basic Halstead model as well as the adopted counting method of program tokens. Section 3, however, presents the proposed model and its three metrics. The results obtained in applying the model on a set of programs are presented and analyzed in section 4. In section 5, some concluding remarks are presented.

Basic metrics: According to Halstead the program source code is interpreted as a sequence of tokens and classifying each token to be an operator or an operand. The following are therefore calculated:

- the total number of unique (distinct) operators (n_1),
- the total number of unique (distinct) operands (n_2),
- total number of operators (N_1),
- total number of operands (N_2).

The number of unique operators and operands (n_1 and n_2) as well as the total number of operators and operands (N_1 and N_2) are calculated by collecting the frequencies of each operator and operand token in the source program.

Other Halstead measures are derived from these four quantities with certain fixed formulas as shown in Table I:

Table 1:

Measure	Formula
Program Length	$N = N_1 + N_2$
Program Vocabulary	$n = n_1 + n_2$
Volume	$V = N (\log_2 n)$
Difficulty	$D = (n_1/2) (N_2/n_2)$
Effort	$E = DV$

It is important that the counting strategy be clearly defined and consistent, since all Halstead's software science depends on counts of operators and operands and there is no general agreement among researches on the most meaningful way to classify and count these tokens. We have used a counting strategy on which there exist a consensus in [11] and [12]. In [13], some rules are proposed for identifying operators and operands in the object oriented programming language. The entities that can be used to apply Halstead metrics are the source code itself or the algorithms of that source code. When Halstead metrics are applied to these two entities, different values for the same base measures are obtained. In both C++ and Java languages, each statement in the source code must be ended with a semicolon (;), which is an operator. This requirement, however, does not exist in the equivalent algorithm for that source code. In our work, we have excluded this operator (i.e. the semicolon at the end of each statement), while counting the operators. This representation condition effects directly the program length $(L=N_1 + N_2)$, whose equation is shown in Table (I). This effect on program length was studied by Kiricenko and Oramanjienva in [14].

The proposed model: The proposed model is composed of three metrics concentrating on the investment of the inheritance property in program design. These metrics are derived by establishing relations between program volume before and after the use of inheritance and hence measuring the achieved reduction in program volume.

a. The inheritance ratio (h_r): This metric is calculated according to the following formula:

$$h_r = V_r / V_{nh} \quad (1)$$

where

V_r represents the volume of the program when using inheritance.

V_{nh} represents the volume of the same program when no inheritance is used.

The ratio (h_r) represents the saving achieved in program volume, when the program is designed with inheritance to its volume designed according to the functional oriented approach. This metric will be a tool for estimating and evaluating the costs of program design and program test as well as program complexity.

b. The inheritance level (h_l): This metric refers to the reduction achieved in program volume when different levels of inheritance are used in designing the same program, compared with the program volume when it is designed without implementing the inheritance property.

This metric is given by the following formula:

$$h_l = V_{hi} / V_{nh} \quad (2)$$

where:

V_{hi} represents the volume of the program when the i^{th} level of inheritance is implemented.

V_{nh} represents the volume of the same program when no inheritance is used.

This metric is an extension to the inheritance ratio (h_r) metric., where for a certain program, a design alternative being assessed for the that metric may be among the design alternatives considered for assessment for this metric.

The lower the value of (h_l) the better the design alternative and of course the lowest achieved value of (h_l) gives an indication to the best design alternative.

c. Effort ratio (Er): This metric reflects the save in the programmers effort for writing a program. The implementation effort according to Halstead is proportional to both the volume (V) and the difficulty (D) of the program, as shown in Table I.

The effort ratio (*Er*) metric is obtained by applying the following formula:

$$E_r = E_h / E_{nh} \quad (3)$$

where

E_h represents the effort to write a program when inheritance is implemented.

E_{nh} represents the volume of the same program when no inheritance is used.

To give a better indication, the value of this metric need to be less than one. The lower its value, the better the indication, means that less effort is required in writing a program with implementing the inheritance property.

RESULTS AND THEIR ANALYSIS

i. The inheritance ratio (h_r): We have experimented our model by applying it to a sample composed of five programs.

Figure (1) illustrates the results for the first metric (i.e. the inheritance ratio). In figure (1.a) the volumes, when the inheritance property is implemented and when no inheritance is used for the five different programs are shown. The drops in volumes are shown in figure (1.b), where inheritance ratios are illustrated.

All of the five programs used have a relatively long source code. For short programs, it will be difficult to make a comparison between volumes, and the inheritance ratio may not necessarily be more than 1.

The general indication obtained from this metric is that the implementation of the inheritance property leads to an expected reduction in the costs of both software design and test as program volumes have decreased.

ii. The inheritance level (h_l): To investigate how the second metric (the inheritance level) behaves, two other programs (referred to as PROG1 and PROG2) were designed, and four different design alternatives were implemented for each one of them. The results of these design alternatives are shown in figures (2.a) and (2.b).

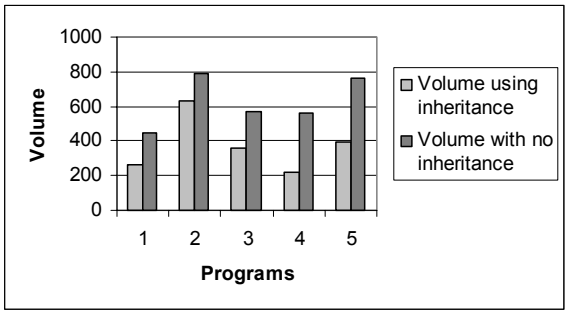

Fig. 1a: Volumes for five sample programs with and with no inheritance

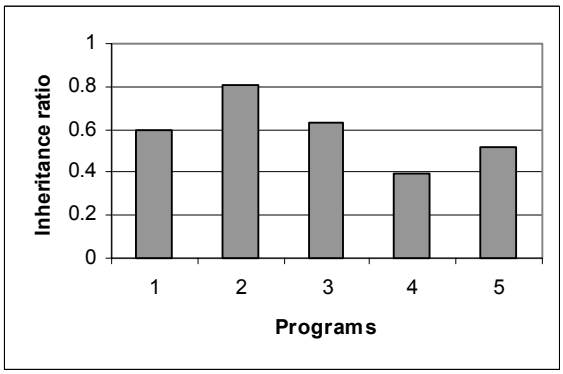

Fig. 1b: Inheritance ratios of the 5 sample programs

Both figures behave in agreement with the results obtained in section (i) above, where volumes have dropped compared with the cases of point 0 at the inheritance level axis. The curves in these figures are shaped as part of a parabolic curve with their minima are at the points of inheritance level =2, showing the minimum volume. This behavior gives an indication that going deep in inheritance levels is not necessarily always in favor of program volume reduction.

With the increase of inheritance levels, the number of methods coupled between different classes increases, thereby increasing the difficulty of the software and the estimated costs of test.

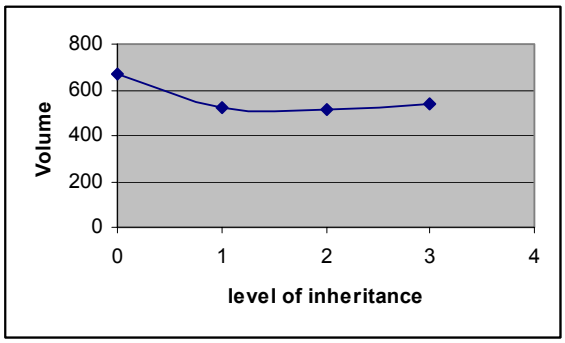

Fig. 2a: Volumes for program (PROG1) with different design alternatives

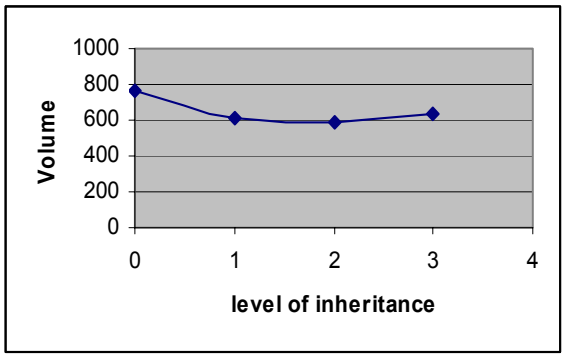

Fig. 2b: Volumes for program (PROG2) with different design alternatives

iii. The effort ratio (E_r): Figure (3) illustrates the developer's effort required for writing programs. The same five different programs of (i) above are used for experimentation. In figure (3.a) a comparison between the efforts required with and without implementation of inheritance is presented. Figure (3.b) shows the behavior of the effort ratio for the sample programs.

The curves shown in figures (3.c) and (3.d) below illustrate the calculated effort for the two previously mentioned programs (PROG1 and PROG2) implemented with different designs. Each design alternative is based on different level of inheritance.

Figure (3.c) behaves in a similar manner as figure (2.a) behaves, showing that the lowest effort is required, when the inheritance level=2. Figure (3.d), however, shows a slightly different behavior, where the effort has increased when the inheritance level =1. Such increase can be justified by the increase in the difficulty ($D=n_1/2$) (N_2/n_2), where more operators and operands are used but not invested for inheritance yet. When the inheritance levels are increased and more classes are derived with inherited properties, a noticeable decrease in effort is obtained.

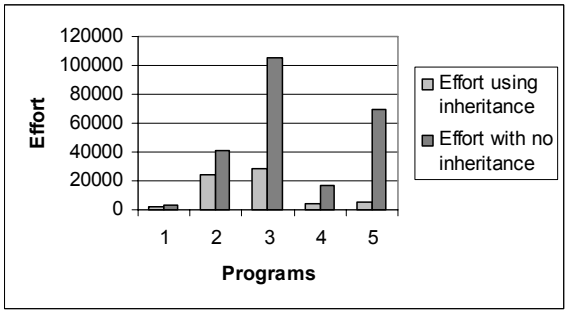

Fig. 3a: Efforts for five sample programs with and with no inheritance

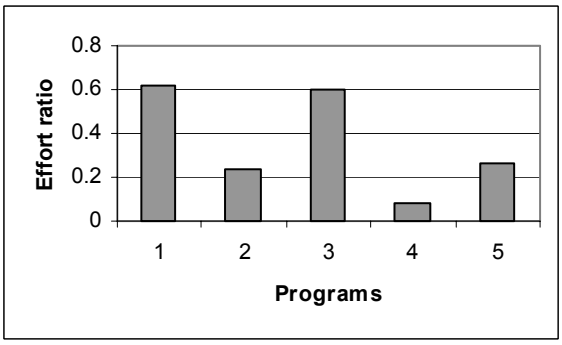

Fig. 3b: Effort ratios of five sample programs

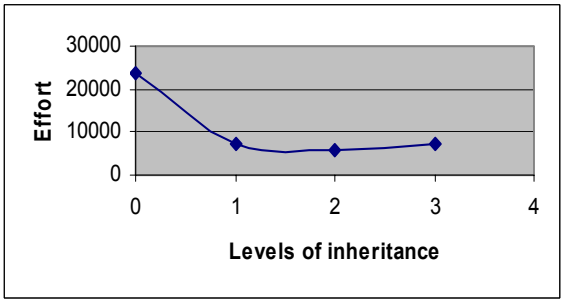

Fig. 3c: Efforts for program (PROG1) with different design alternatives

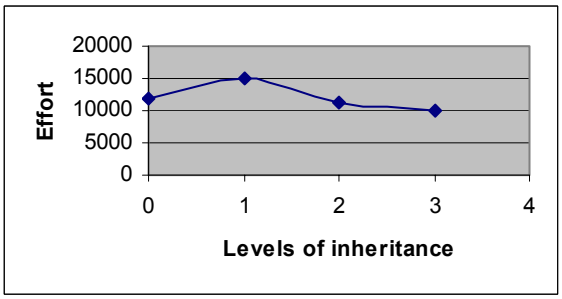

Fig. 3d: Efforts for program (PROG2) with different design alternatives

CONCLUSION

Metrics are units of measurement that are used to characterize products, processors and people and hence allow a definition for their success or failure. Metrics can also help in identifying and quantifying improvement or degradation in our products, processes and people.

Metrics for object-oriented software engineering is affected by the features of the object oriented approach of software development such as: localization, encapsulation, information hiding inheritance and object abstraction technique.

The three proposed metrics in this paper depend on implementing the inheritance property when designing software programs. These three metrics are; the

inheritance ratio, the inheritance level and the effort ratio.

From the application of the model on a number of sample programs we can conclude that the investment of the inheritance property leads to a decrease in the volume of programs. It also leads to a decrease in efforts required for implementation. The depth of inheritance affects the volume and effort in program development. Generally speaking, having more levels of inheritance leads to reducing volume and effort. Practically, however, there exists a level which can be considered better than others.

Through the application of the sample programs on our model, the second level of inheritance gave optimum volumes and efforts. And in general we can claim that our results give indications to the level of inheritance that is relatively better than others. Further application of sample programs on our model will improve the results and may lead to a rule that can quickly point out the most suitable inheritance level for a given program.

REFERENCES

1. DeMarco, T., 1998. Controlling Software projects: Management, measurement, and estimation. Yourdon Press, New York.
2. Halstead, M.H., 1977. Elements of Software Science. New York: Elsvier North-Holland.
3. Leach, R.J., 1995. Using metrics to evaluate student programs. ACM SIGCSE Bulletin, 27: 41-48.
4. Chuan, C.H., L. Lin, L.L. Ping and L.V. Lain, 1994. Evaluation of query languages with software science metrics. Proc. IEEE Eegion 10's Ninth Annual Intl. Conf. Frontiers of Computer Technology, Singapore, pp: 516-520.
5. Baily, C.T. and W.L. Dingee, 1981. A Software Study Using Halstead Metrics. Proc. 1981 ACM Workshop / Symposium on Measurements and Evaluation of Software Quality, Maryland, USA, pp: 189-197.
6. Al Qutaish, R.E., 1987. Incorporating Software Measurements into a Compiler. MSC Thesis. Department of Computer Science, Serdang: Putra University of Malaysia.
7. Samoladas, L., I. Stamelos, L. Angelis and A. Oikonomou, 2004. Open source software development should strive for even greater code maintainability. Communication of ACM, 47: 83-88.
8. Subhas, C. M. and Virendrakumar C. B., 2003. Measures of Software Systems Difficulty. www. Asq.org, SQP Vol. 5, No. 4.
9. Daniel, K. 2003. Halstead Metrics with Java Bytecode. ACM SIGPLAN Notices, Vol. 17, No. 11, 2.
10. Pressman, R.S., 2005. Software Engineering: A Practitioner's Approach. 6th Edn. Mc-Graw Hill Co.
11. Abd Ghani, A. A. and Hunter, R. 1996. An Attribute Grammar Approach to specifying Halstead's Metrics. Malaysian Journal of Computer Science, Vol. 9, No.1, 1996, pp.56-67.
12. Conte, S. D., Dunsmore, H. E., and Shen, V., Y. 1986. Software Engineering Metrics and Models. Menlo Park, California : Benjamin Cummings.
13. Li, D. Y., Kiricenko, V., and Ormandjieva, O. 2004. Halstead's software Science in Today's Object Oriented World. Metrics News, Vol. 9, No. 2. pp. 33-40.
14. Kiricenko, V., and Ormandjieva, O. 2005. Measurement of OOP size based on Halstead's Software Science. In proceedings of the 2nd Software European Forum. Rome, Italy.