Establishing Relationship between Complexity and Faults for Object-Oriented Software Systems

Usha Chhillar[1], Sucheta Bhasin[2]

[1] Department of Computer Science, Kurukshetra University, Kurukshetra, Haryana, India

[2] Department of Computer Science, Kurukshetra University, Kurukshetra, Haryana, India

Abstract
Controlling and minimizing software complexity is the most important objective of each software development paradigm because it affects all other software quality attributes like reusability, reliability, testability, maintainability etc. For this purpose, a number of software complexity measures have been reported to quantify different aspects of complexity. Complexity and fault proneness are two prominent parameters for improving quality of the software . The software industry is continuously facing the challenges of growing complexity of software and increased level of data on defects. To control the complexity and faults is one of the major challenges for researchers to predict different parameters which are responsible for increasing complexity and fault proneness. In this paper, faults prediction through bebugging technique has been tried through an experiment applied to C++ programs and compared the results with various object-oriented complexity measures. The results have been found encouraging. Relationship between faults and complexity has also been established.

Keywords: *Reusability, Reliability, Testability, Maintainability, Fault Proneness, Faults Prediction, Bebugging.*

1. Introduction

From time to time, various complexity metrics have been designed in an attempt to measure the complexity of software systems. Software complexity directly affects maintenance activities like software reusability, understandability, modifiability and testability. Estimates suggest that about 50 to 70 % of annual software expenditure involve maintenance of existing systems. Predicting software complexity and faults can save millions in maintenance [1,7,9,10,18]. Clearly, if complexities could somehow be identified and measured, then software developers could adjust development, testing and maintenance procedures and effort accordingly. This concern has motivated several researchers to define and validate software complexity measures and establish relationship between software complexity and faults [1, 2, 3, 5, 7, 16, 19, 20, 21]. It is accepted by both software developers and researchers that complexity of software can be controlled more effectively through object-oriented approach than traditional function-oriented approach. It is because that objected-oriented paradigm controls complexity of a software system by supporting hierarchical decomposition through both data and procedural abstraction [9]. But, the complexity of software is an essential attribute, not an accidental one [6]. Traditional software complexity metrics are not appropriate for object-oriented software systems due to their distinguish features like class, inheritance, polymorphism, coupling, and cohesion.

In this paper, faults prediction through bebugging technique has been tried through an experiment applied to C++ programs and compared the results with various object-oriented complexity measures. The results have been found encouraging. Relationship between faults and complexity has also been established.

Rest of this paper is organized as follows: Section 2 presents overview of software complexity and existing complexity measures. Faults prediction through bebugging is explained in section 3. Section 4 describes experiment design for faults prediction. Section 5 discusses the experimental results. Finally, section 6 concludes the paper with directions for future work.

2. Overview Of Software Complexity And Existing Complexity Measures

2.1 Software Complexity

In literature, software complexity has been defined differently by many researchers. Zuse [11] defines software complexity as the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs. According to Henderson-Sellers [12] the cognitive complexity of software refers to those characteristics of software that affect the level of resources used by a person performing a given task on it. Basili [4] defines software complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. Here, interacting system may be a machine or human being. Complexity is defined in terms of execution time and storage required to perform the computation when computer acts as an interacting system. In case of human being (programmer) as an interacting system, complexity is defined by the difficulty of performing tasks such as coding, testing, debugging or modifying the software. Bill Curtis [13] has reported two types of software complexity – Psychological and Algorithmic. Psychological complexity affects the performance of programmers trying to comprehend or modify a class/module whereas algorithmic or computational complexity characterizes the run-time performance of an algorithm. Brooks [6] states that the complexity of software is an essential attribute, not an accidental one. Essential complexity arises from the nature of the problem and how deep a skill set is needed to understand a problem. Accidental complexity is the result of poor attempts to solve the problem and may be equivalent to what some are calling complication. Implementing wrong design or selecting an inappropriate data structure adds accidental complexity to a problem.

Software complexity can not be defined by a single definition because it is multidimensional attribute of software. So, different researchers/users have different view on software complexity. Therefore, no standard definition exits for the same in literature. However, knowledge about software complexity is useful in many ways. It is indicator of development, testing, and maintenance efforts, defect rate, fault prone modules and reliability. Complex software/module is difficult to develop, test, debug, maintain and has higher fault rate.

2.2 Software Complexity Measures

Software complexity can not be removed completely but can be controlled only. But, for effective controlling of complexity, we need software complexity metrics to measure it. From time to time, many researchers have proposed various metrics for evaluating, predicting and controlling software complexity. Halstead's software science metrics, McCabe's cyclomatic number and Kafura's & Henry's fan-in, fan-out are the best known early reported complexity metrics for traditional function-oriented approach [16,17,18]. But these metrics do not consider object oriented features of software for measuring the complexity of software. So traditional software complexity metrics are not suitable for measuring complexity of object oriented software.

Various researchers have proposed many object oriented metrics to compute complexity of object oriented software. Chidamber and Kemerer [1] proposed a suite of six metrics : Number Of Children (NOC) - number of immediate derived classes, Depth Of Inheritance Tree (DIT) - maximum path length from root to node in inheritance tree, Weighted Methods per Class (WMC) - sum of all methods of a class, Coupling Between Objects (CBO) - number of classes to which a class is coupled, Lack Of Cohesion in Methods (LCOM) - measures the dissimilarity of methods in a class and Response For a Class (RFC) - number of methods of a class to be executed in response to a message received by an object of that class. These metrics measure complexity of object-oriented software by using design of classes. WMC measures the complexity of a class as a sum of complexity of individual methods. Higher values of NOC and DIT are indicator of higher complexity due to involvement of many methods. CBO value for a class is the indicator of total number of other classes to which it is coupled. Mishra [14] proposed a metric for computing the complexity of a class at method level by considering internal structure of method. Fothi et al [8] designed a metric which computes complexity of a class on the basis of complexity of control structures, data and relationship between data and control structures. A metric which calculates overall complexity of design hierarchy was proposed by Mishra [14]. It computes complexity by considering inherited methods only and does not take into account internal characteristics of methods.

3. Faults Prediction Through Bebugging

The process of finding and rectifying faults in a program is called debugging. Bebugging is the reverse of debugging. In bebugging , a fixed number of artificial bugs are introduced in a source program. The complete detail of these artificial bugs is kept for identifying and removing the same from the source program after the experiment. By applying bebugging method, we may predict how many faults are still present in the source code and thus in a software system. Suppose P is the source program in which we want to predict the number of faults present through bebugging method.

Let
I = Number of artificial faults introduced in program P.

T = Total number of faults find out by the reviewer or a tester in program P.
R = Number of faults find out from I.
(T-R) = New additional faults find out
Total number of predicted faults (PF) in the program P :

$$PF = ((T - R)/R) \times I$$

The bebugging method is generally used by Zoologist for estimating the number of fish in a tank.

For example, take a sample of 100 fish from a pond. Mark them and put them back into the pond for mixing them with the total population of the tank. Again take the sample of 100 fish and find how many marked fish are in this sample. Let marked fish are 10. According to the bebugging method, there are 900 fish in the pond. In this method , we assume that the original sample was random and remixing of fish was homogenous.

Similarly, if we insert 5 bugs in a program and reviewer reports total 9 faults through bebugging process. Let out of these 9 bugs, 3 bugs are out of 5 bugs inserted by us.
Then
I = 5, T = 9, R = 3, T-R = 6.
PF = ((T-R) /R)*I = 10
It means predicted number of faults present in the program are 10.

4. Experiment Design

In this experiment, the main objective is to predict number of faults in a program by using the bebugging method described in the previous section and also to analyze the effect of faults on the complexity. It is intuitive that a programmer finds lesser number of faults in a complex program than a simple program in a given time period.

For this purpose, an experiment was conducted at the end of the academic session by involving 15 MCA fourth semester students on scheduled date and time. For this purpose, 10 programs written in C++ language were used. In each of these programs, five logical and syntax errors are inserted knowingly called artificial bugs. The purpose of the experiment was explained well to the students before conducting the experiment and they were asked to find out logical and syntax errors as many as they can. The experiment was conducted in 10 continuous sessions of 10 minutes duration each. After each session, the sheets of the program specified for the session were collected from the students . In this way 10 different programs were given to the students in 10 different sessions. One sample program has been given in table 2.

5. Experimental Results

For all the 10 programs used in the experiment, the number of predicted faults (PF) were calculated by using above mentioned bebugging technique for each of the 15 students (S1-S15) and results are tabulated in table 1. One sample program has been given in table 2. Type and description of errors inserted in sample program are described in table 3. We have also calculated three Chidamber and Kemerer’s (CK) metrics – WMC, NOC, DIT, McCabe’s complexity measure V(G), lines of code (LOC) metric and composite weighted complexity metric (CWT) for the programs studied here and results are shown in table 4 [1,5,15,17,18].

We also analyzed the relationship between complexity and predicted number of faults through bebugging method. For this purpose, we have drawn bar graphs among complexity metrics and faults described in table 4. These bar graphs have been given in figures 1-6. The bar graphs clearly show that fault rate is directly proportional to complexity i.e. more complexity implies more possibility of faults and hence less quality . However, the results of program no 9 and 10 vary because in these two programs coupling is the dominating factor . Due to more coupling, number of faults are more where as the value of complexity measures WMC, NOC, DIT,V(G) and LOC are less for these two programs. From this it is clear that coupling plays major role for increasing the complexity and reducing the quality of programs/software. So, it should be controlled to minimum to develop a good quality software.

Table 1: Students wise Experimental Predicted Faults (PF)

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
S1	0	0	3.3	0	3.3	0	10	3.3	25	0
S2	1.5	3.3	0	1.5	0	7.5	5	0	10	1.25
S3	1.7	0	2.5	1.7	1.7	0	0	0	5	0
S4	0.5	2.5	0	0.5	0	2.5	0	0	1.5	2.5
S5	2.5	2.5	0.5	0	0.5	2.5	0	5	5	20
S6	0	0	5	0	0	2.5	2.5	0	5	7.5
S7	0.5	0	2.5	0	0	0	0	0	2.5	1.25
S8	1.7	1.0	0.1	1.7	0	1.25	0	0	1.25	1.7
S9	20	1.5	1.5	0	0.5	2.5	20	5	5	6.5
S10	0	0.5	0.5	0	0.5	10	0	5	5	3.3
S11	5	0	1.5	0	6.5	0	10	6.5	10	1.7
S12	0	1.7	3.3	0	0	15	2.5	0	0	2.5
S13	0	2.5	1.7	5	0	1.7	3.3	0	7.5	3.3
S14	0	2.5	2.5	0	0	0	0	0	1.5	5
S15	0	0	2.5	0	1.7	2.5	0	1.7	5	0
Total	33.4	18.0	27.4	10.4	14.7	47.95	53.3	26.5	89.25	56.50

Table 2: Sample program

Program : To find out the greatest number out of three number using single inheritance

```
   #include<iostream.h>
   #include<conio.h>
01      Class Abc
02      {
03       protected:
04         int a ,b ,c;
05        public:
06        void input();
07        void output();
08       };
09      void Abc:: input()
10      {
11        cout<< enter the value  ;
12        cin >>a>>b
13        }
14    void Abc:: output()
15      {
16        cout<<" a ="<<a;
17        cout<< "b="<<b;
18       cout<<"c="<<c;
19      }
20  class xyz : private Abc
21      {
22        public:
23       void  greatest();
24      };
25     void greatest :: greatest()
26      {
27        input();
28       if (a>b)
29        if(a>c)
30      {
31        cout<<"a is the greatest number" ;
32       }
33        else{
34              cout<<" c is the greatest number" ;
35             }
36     }
37      else
38        if(b>c)
39         {
40          cout <<" b is the greatest number" ;
41          }
42       else;
43         {
44          cout <<" c is the  greatest   number";
45          }
46  void  main()
47       {
48          xyz a;
49         clrscr();
50         a.output();
51         a.greatest();
52          getch();
53         }
```

Table 3: Description of errors in sample program

LINE NO	ERROR TYPE	ERROR DESCRIPTION
11	Syntax	Undefined symbol 'enter' due to" " is missing in cout statement
12	Logical	Always c is the greatest number because the value of c is not read.
28	Syntax	Declaration terminated incorrectly due to '{' brace missing.
42	Syntax	else is terminated with ';'.
51	Syntax	abc::out not accessible due to abc class is inherited in private mode

Table 4: Values of complexity measures and errors

P.No	LOC	VG	WMC	NOC	DIT	CWT	Errors
P1	20	4	1.5	0.5	0.5	153	33.4
P2	16	2	1	0.67	0.33	112	18.0
P3	24	4	1.6	0.67	1	285	27.4
P4	11	2	1	0.5	0.5	69	10.4
P5	16	2	1.5	0.5	0.5	70	14.9
P6	35	6	1	0.67	0.33	303	47.95
P7	39	10	1.3	1	1	645	53.3
P8	24	4	1.6	0.67	1	285	26.5
P9	12	1	1	0.67	0.67	59	84.75
P10	17	1	2	0	0	71	56.50

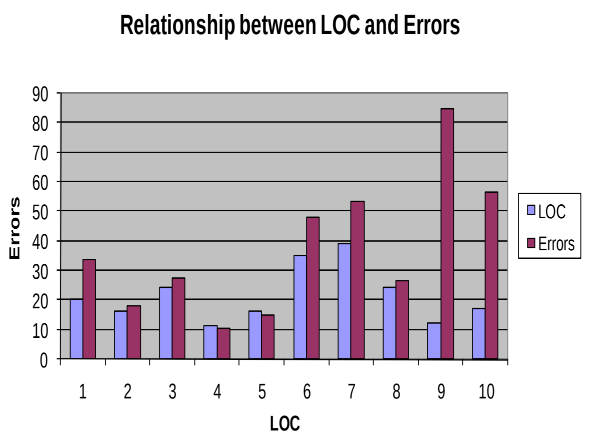

Fig. 1: Relationship between LOC and Errors

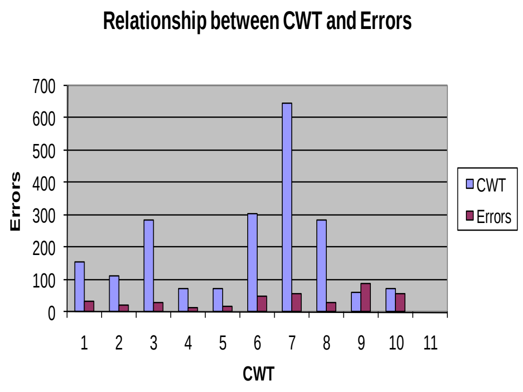

Fig. 2: Relationship between CWT and Errors

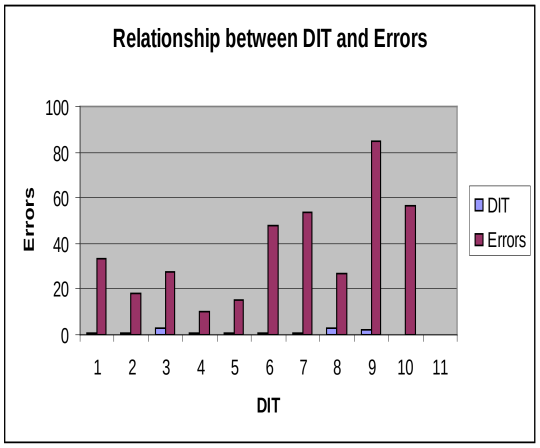

Fig. 5: Relationship between DIT and Errors

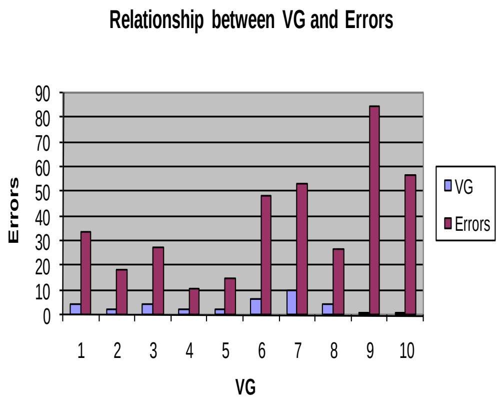

Fig. 3: Relationship between VG and Errors

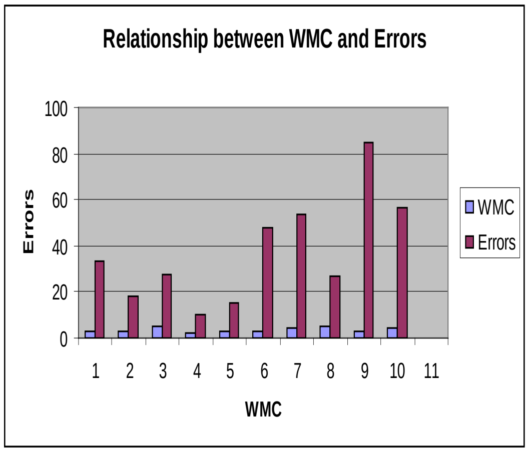

Fig. 6: Relationship between WMC and Errors

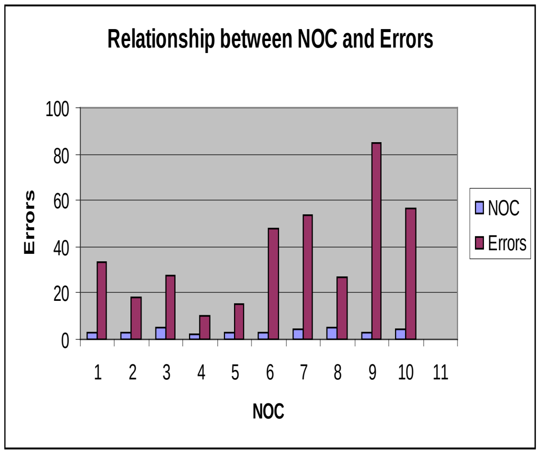

Fig. 4: Relationship between NOC and Errors

6. Conclusions And Directions For Future Work

In this paper, faults prediction through bebugging technique has been implemented through an experiment applied to C++ programs and compared the results with various object-oriented complexity measures. The results have been found encouraging. Relationship between faults and complexity has also been established. Making early decisions about complexity of a object-oriented system may help a lot to software developers in reducing design, testing and maintenance efforts and can improve its quality and reliability as well. The results appear to be logical and fit the intuitive understanding – if more complexity, then more possibility of faults. However, application of conclusions to real life situations needs further study.

So, further empirical research is required using data from industrial projects to validate these findings and to derive more useful and generalized results. Using data from industry implemented projects will provide a basis to examine the relationship between complexity and faults and we can comment on quality of software in a better way.

References

[1] Chidamber, S. R., Kemerer, C.F. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, 1994, pp. 476-492.
[2] Mark, L, Jeff, K. Object Oriented Software Metrics, Prentice Hall Publishing, .1994
[3] Basili, V.R., Biand, L., Melo, W.L. A validation of object-oriented design metrics as quality indicators, Technical report, Uni. of Maryland, Deptt. of computer science, MD, USA., 1995
[4] Basili, V. Qualitative Software Complexity Models: A Summary, In Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1980.
[5] Singh, R., Grover, P.S. A New Program Weighted Complexity Metric, Proc. International conference on Software Engg. (CONSEG'97), Chennai, India, 1997, pp. 33-39.
[6] Brooks, I. Object Oriented Metrics Collection and Evaluation with Software Process, presented at OOPSLA'93 Workshop on Processes and Metrics for Object Oriented software development, Washington, DC, 1993.
[7] Harrison, W. Magel, K, Kluezny, R., dekock, A.: Applying Software Complexity Metrics to Program Maintenance, IEEE Computer, 15,1982, pp. 65-79.
[8] Fothi, A. Gaizler, J., Porkol, Z. The Structured Complexity of Object-Oriented Programs, Mathematical and Computer Modeling, 38, 2003, pp. 815-827.
[9] Da-wei, E. The Software complexity model and metrics for object-oriented, IEEE International Workshop on Anti-counterfeiting, Security, Identification, 2007, pp. 464-469.
[10] Brooks, F.P. The Mythical Man Month: Essays on Sofware Engineering, Addison-Wesley, 1995.
[11] Zuse, H. Software Complexity Measures and Methods, W.de Gruyter, New York, 1991.
[12] Sellers, B. H. Object-Oriented Metrics : Measures of Complexity, Prentice Hall, New Jersey, 1996.
[13] Curtis, B. Measurement and Experimentation in Software Engineering, Proc. IEEE conference, 68,9, 1980, pp. 1144-1157.
[14] Mishra, S. An Object Oriented Complexity Metric Based on Cognitive Weights, Proc. 6th IEEE International Conference on Cognitive Informatics (ICCI'07), 2007.
[15] Usha Chhillar, Sucheta Bhasin. A New Weighted Composite Complexity Measure for Object-Oriented Systems, International Journal of Information and Communication Technology Research, 1 (3), 2011.
[16] Elish, M.O., Rine, D. Indicators of Structural Stability of Object-Oriented Designs: A Case Study, Proc. 29th Annual IEEE/NASA Software Engineering Workshop(SEW'05), 2005.
[17] Halstead, M.H. Elements of Software Science, New York: Elsevier North Holland, 1977.
[18] McCabe, T.J. A Complexity Measure, IEEE Trans. On Software Engg., SE-2, 4, 1976, pp. 308-320.
[19] Aggarwal, K.K. Empirical Study of Object-Oriented Metrics, Journal of Object Technology, 5, 2006, pp. 149-173.
[20] Usha Kumari, Sucheta Bhasin. Application of Object-Oriented Metrics To C++ and Java : A Comparative Study, ACM SIGSOFT Software Engineering Notes, 36 (2), 2011, pp. 1-6.
[21] Singh, R. Improving Quality through Faults Prediction, International conference on Quality, Reliability and Information Technology at the Turn of the Millennium, December 21-23, 2000, New Delhi.

Usha Chhillar is working as Head, Department of Computer Science, A.I.J.H.M. PG college, Rohtak, Haryana, India. Currently, she is persuing her Ph.D Degree from Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana, India. She obtained her Master Degree in Computer Science from Maharshi Dayanand University (MDU), Rohtak and M.Phil (Computer Science) from Ch. Devi Lal University (CDLU), Sirsa. She has total more than twelve years teaching experience. Her research interests include Software Engineering, Object-Oriented and Component-based Software Metrics.

Dr. Sucheta Bhasin is working as Associate Professor, Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana, India. She has total more than 23 years teaching and research experience in the University. She has published more than 60 research papers in International/National journals and conferences. She is life member of International Forum for Interdisciplinary Mathematics and Indian Society of Information Theory & Applications. Her research areas include Networking, Software Metrics, Object-Oriented and Component-based Software Metrics.