Volume 1 No. 3, July 2011 ISSN-2223-4985
International Journal of Information and Communication Technology Research

http://www.esjournals.org

A New Weighted Composite Complexity Measure for Object-Oriented Systems

Usha Chhillar, Shuchita Bhasin
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. As the development of object-oriented software is rising, more and more complexity metrics are being developed for the same. In this paper, we have attempted to design a weighted composite complexity measure by employing the concept of weights to quantify various aspects of complexity which may arise due to inheritance level, type and nesting level of control structures, and size of the class/program. The proposed weighted measure yields quiet interesting results and insight into various complexity aspects of software. Estimation of the new measure is also compared with four Chidamber and Kemerer's (CK) metrics – weighted methods per class (WMC), number of children (NOC), depth of inheritance tree (DIT), coupling between objects (CBO) and McCabe's complexity measure v(G) and is illustrated by practical programs.

Keywords: *Weighted composite complexity measure, testability, maintainability, reusability, object-oriented software, WMC, NOC, DIT, CBO, v(G).*

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 can save millions in maintenance [1,7,21]. 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 [1,2,3,5,15,18,19]. As the development of object-oriented software is rising, more and more complexity metrics are being developed for the same. 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 [10]. Traditional software complexity metrics are not appropriate for object-oriented software systems due to its distinguish features like class, inheritance, polymorphism, coupling and cohesion.

Different complexity measures take into account different aspects of complexity. For function-oriented approach, Halstead's software science metrics [16] and McCabe's cyclomatic number [17] are the two prominent contributions but account for different characteristics of a computer program. For Halstead's metric, the basis of detecting and measuring the complexity is the size of software module only and it does not take into account other characteristics of software systems such as program flow control, nesting and so on [16,17]. Similarly, McCabe's cyclomatic number measures the complexity due to program flow control only, but does not keep track of other features of software systems such as size in terms of operators, operands and methods etc. Similarly, for object-oriented approach, Chidamber and Kemerer's suite of six metrics are prominent complexity measures. But, some of these differ in their basis for complexity computation. For example, WMC is based on number of methods in a class while NOC and DIT take into consideration the concept of inheritance for complexity calculation.

In this paper, we have attempted to design a weighted composite complexity measure which take into account the complexities due to inheritance level of statements in the classes, size of class/program in terms of token counts – operators, operands, methods, type of control structures present in the class/program and their nesting. The concept of weight is used to quantify these various aspects of complexity of a class/program.

Rest of this paper is organized as follows : Section 2 presents overview of software complexity and existing complexity measures. Section 3 describes proposed complexity measure. Section 4 discusses the experimental results and the comparison of results with four Chidamber and Kemerer's (CK) metrics [1]] – WMC, NOC, DIT, CBO and McCabe's complexity measure V(G) [17]. Finally, section 5 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 [10] 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, maintenance etc. 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. 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 [Fothi,03] 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 Yadav et al [20]. It computes complexity by considering inherited methods only and does not take into account internal characteristics of methods.

3. PROPOSED WEIGHTED COMPLEXITY MEASURE

Software complexity can not be computed by a single parameter of program/software because it is multidimensional attribute of software. The prominent factors which contribute to complexity of a program/software are:

Inheritance Level of Statements in Classes: A statement which is at deeper level of inheritance of classes is harder to understand and thus contributes more complexity than otherwise. We take effect of inheritance level of classes by assigning weight 0 to statements at level one (the outermost level/class) i.e in the base class, weight 1 for those statements which are at level 2 i.e first derived class, weight 2 to those statements which are in next derived class (level 3) and so on.

Types of Control Structures: A program/class with more control structures is considered to be more complex and vice-versa But, we assume that different control structures contribute to the complexity of a program/class differently. For example, iterative control structures like for loop, while .. do, do .. while contribute more

complexity than decision making control structures like if .. then .. else. Therefore, we assign different weights to different control structures.

Nesting of Control Structures: A statement which is at the deeper most level of nesting (the inner most level) of control structures is harder to understand and thus contributes more complexity than otherwise. We also take effect of nesting of control structures by assigning weight 1 to statements at level one i.e the outer most level, weight 2 for those statements which are at level 2 i.e the next inner level of nesting and so on. The weight for sequential statements is taken as zero.

Size: Size is also considered one of the parameter of program/class complexity. A class with more methods is harder to understand than a class with less number of methods and hence contributes more complexity [1,16]. Large programs incur problem just by virtue of volume of information that must be absorbed to understand the program and more resources have to be used in their maintenance [1,7]. So, size is a factor which adds complexity to a program/class.

By taking these factors into account, a weighted complexity measure for an object-oriented program P is suggested as:

$$C_w(P) = \sum_{j=1}^{n} (S_j) * (W_t)_j$$

Where
$C_w(P)$: Proposed weighted complexity measure of program P,
S_j = size of j th executable statement in terms of tokens count (operators + operands + methods/functions + strings),
n = Total number of executable statements in program P,
Σ = Summation symbol,
j = Index variable
W_t = Total weight of j th executable statement in program P,
Where $W_t = W_n + W_i + W_c$
Here
W_n = weight due to nesting level of control structures and it is
= 0 for sequential statements,
= 1 for statements inside the outer most level of control structures,
= 2 for statements inside the next inner level of control structures and so on.

W_i = weight due to inheritance level of statements in classes and it is
= 0 for statements inside the outer most level of inheritance i.e inside base class,
= 1 for statements inside the next level of inheritance i.e first derived class,
= 2 for statements inside the next deeper level of inheritance i.e next derived class and so on.

W_c = weight due to types of control structures and it is
= 0 for sequential statements,
= 1 for decision making control statements like if .. then .. else,
= 2 for decision making control statements like while .. do, for loop, do .. while,
= n for switch statement with n cases.

4. EXPERIMENTAL RESULTS

Consider the two test programs P1 and P2 as listed below:

Test Program P1

```
Class Abc
{
 int a,b,c;
public :
void input();
void output();
};
void Abc:: input()
{
 cout<<"enter the value";
 cin >>a>>b>>c;
}
void Abc:: output()
{
 cout<<" a ="<<a;
cout<< "b="<<b;
cout<<"c="<<c;
}
class greatest : public Abc
{
 void check();
};
void greatest :: check()
{
 if (a>b)
 {
 if(a>c)
 {
 cout<<"a is the greatest number" ;
 else
 {
 cout<<" c is the greatest number" ;
 }
}
 else
 if(b>c)
 cout <<" b is the greatest number" ;
 else
 cout <<" c is the greatest number";
}
void main()
```

```
{
 greatest g;
 g.getdata();
 g.putdata();
 g.greatest();
 getch();
}
```

Test Program P2

```
Class input
{
 protected :
 int a[10];
 int n,I;
 public:
 void getdata();
 void putdata();
}
void input:: getdata()
{
 cout<<"enter the value of n"
 cin>>n;
 cout<<"enter the array element";
 for(i=0;i<n;i++)
 cin>>a[i];
};
void input::putdata()
{
 cout<<"The output is \n ";
 for(i=0;i<n;i++)
 cout<< a[i]<<"\t";
}
class sort::public input()
{
 int j;
 public:
 void sorting();
};
 void sort::sorting()
 {
 int temp;
 for(i=0;i<n;i++)
 {
 for(j=i+1;j<=n;j++)
 {
 if (a[i]>a[j])
 {
 temp=a[i];
 a[i]=a[j];
 a[j]=temp;
 }
 }
 }
 }
class sum:public sort
{
 int s;
 public:
 void summing(int s);
};
void sum:: summing(int s)
{
 for(i=0;i<n;i++)
 s+=a[i];
 cout<<"sum="<<s;
}
void main()
{
 sum st;
 clrscr();
 st.getdata();
 st.putdata();
 st.sorting();
 st.putdata();
 st.summing();
 getch();
}
```

It is clear from the codes of these programs and values of Lines of Code (LOC) metric (LOC=20 for P1 and LOC=29 for P2) that P2 appears more complex than P1. Applying four CK [Chidamber, 94] metrics – WMC, NOC, DIT, and CBO, we get the following results for programs P1 and P2:

	P1	P2
WMC	1.5	1.3
NOC	0.5	1.0
DIT	0.5	1.0
CBO	0.0	0.0

As per the values of CK metrics NOC and DIT, it is clear that program P2 is more complex than P1, but these metrics take into account only one aspect of complexity i.e inheritance and does not reflect the effect of size, type and nesting level of control structures. From the value of WMC metric, it appears that program P1 is more complex than P2. But, it is due the fact that we have calculated average value of WMC for programs P1 and P2. It is surprising that values of CBO metric comes to be equal for both the programs P1 and P2.

Applying McCabe's complexity metric to the same programs P1 and P2, we get the following values:

v(G) = Number of control structures +1= 3+1=4 for program P1.
v(G) = Number of control structures +1= 6+1=7 for program P2.

By looking at these values of McCabe's complexity measure v(G), we infer that program P1 is less complex than P2. But, this metric take into account only one parameter of complexity i.e flow of control. It does not consider object-oriented aspects of complexity like inheritance, level of inheritance, class etc. and other

parameters of complexity – size, type of control structures, and nesting.

Now, by applying the weighted composite complexity measure C_w to the same programs P1 [Table 1] and program P2 [Table 2], we get the following results :

$C_w(P1) = 153$ for program P1.
$C_w(P2) = 460$ for program P2.

From these results, we infer that program P2 is more complex than program P1. But, here, differences are due to multiple attributes of complexity – size, inheritance level of statements in classes, types of control structures and nesting of control structures.

We have applied the above mentioned complexity measures to 5 pairs of programs collected from open literature (details of programs with authors). In each test pair, the second program is supposed to be more complex than the first. Therefore, the values of the complexity metrics applied to these pairs should be more for the second program than the first. Results of our calculations are presented in Table 3. From this table, it is clear that this trend holds. Thus, we infer that, the proposed weighted complexity measure detects complexity and gives realistic estimates. This is because the weighted complexity measure also takes into account the complexity due to factors not considered earlier in the CK metrics suite, McCabe's measure. These factors are size, types of control structures, and their nesting level. As a software system is developed in terms of classes/programs, so the proposed measure can be used to calculate and compare the complexities of classes/programs and hence of object-oriented software systems.

Table 1: Calculation of metric C_w for program P1

Statement no.	Executable statement	$(S)_j$	W_n	W_i	W_c	$(W_t)_j$ $W_t = W_n+W_i+W_c$	$(S)_j*(W_t)_j$
(s1)	void Abc::input()	4	0	1	0	1	4*1=4
(s2)	cout<<"enter value"	3	0	1	0	1	3*1=3
(s3)	cin>>a>>b>>c	7	0	1	0	1	7*1=7
(s4)	void Abc::output()	4	0	1	0	1	4*1=4
(s5)	cout<<"A="<<a	5	0	1	0	1	5*1=5
(s6)	cout<<"B="<<b	5	0	1	0	1	5*1=5
(s7)	cout<<"C="<<c	5	0	1	0	1	5*1=5
(s8)	void greatest::check()	4	0	2	0	2	4*2=8
(s9)	cout<<"The greatest is"	3	0	2	0	2	3*2=6
(s10)	if(a>b)	4	1	2	1	4	4*4=16
(s11)	if(a<c)	4	2	2	1	5	4*5=20
(s12)	cout<<"a"	3	2	2	0	4	3*4=12
(s13)	cout<<"c"	3	2	2	0	4	3*4=12
(s14)	if(b>c)	4	1	2	1	4	4*4=16
(s15)	cout<<"b"	3	1	2	0	3	3*3=9
(s16)	cout<<"c"	3	1	2	0	3	3*3=9
(s17)	void main()	2	0	0	0	0	2*0=0
(s18)	g.input()	3	0	1	0	1	3*1=3
(s19)	g.output()	3	0	1	0	1	3*1=3
(s20)	g.greatest()	3	0	2	0	2	3*2=6

So, $C_w(P1) = \sum_{j=1}^{20} (S_j)*(W_t)_j = 153$

Table 2: Calculation of metric C_w for program P2

Statement no.	Executable statement	$(S)_j$	W_n	W_i	W_c	$(W_t)_j$ $W_t = W_n+Wi+W_c$	$(S)_j*(W_t)j$
(s1)	void input::getdata()	4	0	1	0	1	4*1=4
(s2)	cout<<"Enter value of n"	3	0	1	0	1	3*1=3
(s3)	cin>>n	3	0	1	0	1	3*1=3
(s4)=	cout<<"enterelements\n"	3	0	1	0	1	3*1=3
(s5)	for(i=0;i<n;i++)	9	1	1	2	4	9*4=36
(s6)	cin>>a[i]	3	1	1	0	2	3*2=6
(s7)	void input::putdata()	4	0	1	0	1	4*1=4
(s8)	cout<<"output is"	3	0	1	0	1	3*1=3
(s9)	for(i=0;i<n;i++)	9	1	1	2	4	9*4=36
(s10)	cout<<a[i]<<"\n"	6	1	1	0	2	6*2=12
(s11)	void sort::sorting()	4	0	2	0	2	4*2=8
(s12)	for(i=0;i<n;i++)	9	1	2	2	5	9*5=45
(s13)	for(j=i+1;j<n;j++)	11	2	2	2	6	11*6=66
(s14)	if(a[i]>a[j])	6	3	2	1	6	6*6=36
(s15)	temp=a[i]	4	3	2	0	5	4*5=20
(s16)	a[i]=a[j]	5	3	2	0	5	5*5=25
(s17)	a[j]=temp	4	3	2	0	5	4*5=20
(s18)	void sum::summing(ints)	4	0	3	0	3	4*3=12
(s19)	for(i=0;i<n;i++)	9	1	3	2	6	9*6=54
(s20)	s+=a[i]	5	1	3	0	4	5*4=20
(s21)	cout<<"sum="<<s	5	1	3	0	4	5*4=20
(s22)	void main()	2	0	0	0	0	2*0=0
(s23)	st.getdata()	3	0	1	0	1	3*1=3
(s24)	st.putdata()	3	0	1	0	1	3*1=3
(s25)	st.sorting()	3	0	2	0	2	3*2=6
(s26)	cout<<"sorted array\n"	3	0	0	0	0	3*0=0
(s27)	st.putdata()	3	0	1	0	1	3*1=3
(s28)	st.summing()	3	0	3	0	3	3*3=9

$$\text{So, } C_w(P2) = \sum_{j=1}^{28} (S_j)*(W_t)_j = 460$$

Table 3: Values of complexity metrics for pairs of programs

P.No.	WMC	NOC	DIT	CBO	v(G)	LOC	Cw
1a	1.5	0.5	0.5	0	4	20	153
1b	1.3	1	1	0	7	29	460
2a	1	0.67	0.33	0	2	16	112
2b	1	0.67	0.33	0	6	35	303
3a	1.6	0.67	1	1	4	24	285
3b	1.6	0.67	1	0	5	37	453
4a	1.5	0.5	0.5	1	7	26	407
4b	1.3	1	1	0	10	39	645
5a	1	0.5	0.5	0	2	11	69
5b	1.5	0.5	0.5	1	2	16	70

5. CONCLUSIONS AND FUTURE WORK

A new weighted composite complexity measure has been proposed which takes into account different aspects of complexity: size, control structures, their nesting and inheritance level of statements in classes. Our study shows that the effect of these structures on complexity is quite significant. We have considered only one object-oriented feature of complexity, that is, inheritance level. Other object-oriented features like polymorphism, information hiding, and encapsulation require further study. We have taken small object-oriented programs in our present study, but we expect that these results will serve as guide lines to explore more complex object-oriented systems. Application of conclusions to real life situations needs further study.

REFERENCES

[1] Chidamber, S. R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, 20, 476-492, June 1994.

[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. April 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, 33-39, January 1997.

[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, 65-79, Sept. 1982.

[8] Fothi, A. Gaizler, J., Porkol, Z.: The Structured Complexity of Object-Oriented Programs, Mathematical and Computer Modeling, 38, 815-827, 2003.

[9] Da-wei, E.: The Software complexity model and metrics for object-oriented, IEEE International Workshop on Anti-counterfeiting, Security, Identification, 464-469, 2007.

[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, 1144-1157, September 1980.

[14] Mishra, S.: An Object Oriented Complexity Metric Based on Cognitive Weights, Proc. 6th IEEE International Conference on Cognitive Informatics (ICCI'07), 2007.

[15] 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.

[16] Halstead, M.H.: Elements of Software Science, New York: Elsevier North Holland, 1977.

[17] McCabe, T.J.: A Complexity Measure, IEEE Trans. On Software Engg., SE-2, 4, 308-320, Dec. 1976.

[18] Aggarwal, K.K.: Empirical Study of Object-Oriented Metrics, Journal of Object Technology, 5, 149-173, 2006.

[19] Harrison, R.: An Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Trans. On Software Engg., 24, 1998.

[20] Yadav, A., Khan, R.A.: Measuring Design Complexity – An Inherited Method Perspective, SIGSOFT Software Engineering Notes, 34, 4, July 2009.

[21] Stark, G. Durst, R.C., Vowell, C.W.: Using Metrics in Management Decision Making, IEEE Computer, 42-48, Sept. 1994.