Empirical Evaluation and Critical Review of Complexity Metrics for Software Components

Arun Sharma
Amity Institute of Information Technology
Amity University, Uttar Pradesh

Rajesh Kumar
School of Mathematics and Computer Applications
Thapar Institute of Engineering. & Technology, Patiala.

P. S. Grover
Dept. of Computer Science
Delhi University, Delhi

Abstract *- Component-based development has become a highly widespread approach for application development. Various metrics have been developed by researchers for improving the quality of software components as traditional software products and process metrics are neither suitable nor sufficient in measuring the complexity of these components. The paper proposes a complexity metric for components based on the different constituents of the components, like inheritance of classes, methods and attributes. This metric is applied to various JavaBean components for empirical evaluation. Further, a correlation study has been conducted for this metric with another metric called Rate of Component Customizability (RCC), available in the literature. The study conducted shows the negative correlation between the two which confirms the assumption that high complexity of the components leads to the high cost of maintainability.*

Keywords: *Component, Component-based systems, Complexity*

1. INTRODUCTION

There is a trend of using components in software application development, due to its obvious advantages of low cost, decreased development time and increased usability. The kind of flexibility required by open systems are presently best supported by component oriented Software technology better known as component based software engineering (CBSE). Component Based Development (CBD) is expected a big future and thus a tremendous scope in research. There are many software component models available in the industry, some of these are *Microsoft's* COM (Component Object Model), DCOM, .NET Framework, *Sun's* Java Beans, EJB (Enterprise Java Beans), J2EE specification and *OMG's* (Object Management Group) CORBA (Common Object Request Broker Architecture) specification.

As the technology in software and hardware is changing very fast, it is important to manage the complexity and rapidly adapt to change [1]. Researchers have proposed a wide range of complexity metrics for software systems. However, these metrics are not sufficient for components and component-based system and are restricted to the module-oriented systems and object-oriented systems. We propose here a metric called Component Complexity metric, which may be used to limit the complexity of the component.

This paper is divided into eight sections. Section 2 describes some of the most widely used complexity metrics for procedure-oriented and object-oriented systems. Section 3 describes the proposed complexity metric for components. Section 4 evaluates the proposed metric against Weyuker's properties. Section 5 carries an empirical evaluation of the proposed metric on several JavaBean components along with a correlation study with a metric, called Rate of Component Customizability (RCC) in Section 6. Conclusions and future directions are given in the Section 7 and Section 8 of the paper.

2. RESEARCH PROBLEM

Complexity is a major driver of the cost, reliability, and functionality of software systems. To control complexity, one must be able to measure it. Several metrics have been created for measuring various aspects of complexity such as size, control flow, data structures, and intermodule structure. The most widely used complexity metric is Cyclomatic Complexity proposed by McCabe [2] in 1976. This metrics is based on the program graph and is defined as

$$V(G) = e - n + 2p$$

Where e is the number of edges, n is the number of nodes in the graph and p is the no. of connected components. McCabe proposed that V(G) can be used as a measure of procedure complexity of the program. Kafura and Henry [3] also proposed the complexity metrics based on the number of local information flows entering (fan-in) and exiting (fan-out) in each procedure. This metrics is given as

$$Complexity=(Proc.\ Length) * (fan\text{-}in * fan\text{-}out)^2$$

But in component context, both the metrics defined above may not be used to measure the complexity of entire component as these metrics measure the complexity for procedures only and does not consider the other aspects of the component like classes and attributes, which may contribute a good amount of complexity to the component [4].

Li et. al. [5] proposed other metrics for complexity, based on size measures such as number of methods and attributes. In an object-oriented environment, Chidamber and Kemerer [6] proposed a set of quality measures for class complexity. They proposed Depth of Inheritance Tree (DIT), Number of Children (NOC), and Weighted Methods Complexity (WMC) Coupling between Objects (CBO), Response for a Class (RFC) and Lack of Cohesion in Methods (LCOM). Out of these metrics DIT, NOC, CBO and RFC evaluate the external complexity of the relation between classes and do not depend upon the code complexity of the methods. WMC measures the complexity of the methods and is defined as

$$WMC(C) = \sum_{i=1}^{N} C_i$$

Where C_i is the static complexity of the corresponding method M_i. This metric evaluates the complexity of methods for a class.

For Component- based systems, Gill and Grover [7] proposed a metric, called Component Interface Complexity Metric (CICM). The paper discusses the interface characterization of software components and assumes that the complexity of a component is mainly due to interface signature, interface constraints and interface packaging and configurations. Interface signature characterizes the functionality of the component and consists of properties, operations and events. Interface constraints involve the individual elements and the relationships among these elements. Last is the Interface packaging and configurations, which deals that how a component will be used in an application or in another component. First two parts of this metric deal with the internal functioning of the component and depend on the coding involved during the development while the third one deals with the use of the component after the development.

We extend the approaches adopted in WMC (C) and CICM (C) metrics to propose a new metric for measuring the design complexity of the component. During the designing of the component, the designer usually has one or more use scenarios in mind which may or may not fit at the time of implementation so it is very difficult to decide at the time of designing that in which environment this component will be used. Therefore, we are taking only the interface signature and interface constraints into consideration for our work and not considering the packaging complexity.

3. PROPOSED METRICS

We assume that the complexity of a component depends closely on what contributes to develop components. Strictly, in an object-oriented context, component may consist of classes (base class and derived classes), which in turn may involve various methods, attributes and interfaces. So, we take these aspects into consideration to propose the new metric. The metric is defined as

$$Component\ Complexity\ (CC) = \alpha\, C_V(C) + \beta\, C_M(C) + \gamma\, C_I(C)$$

where α, β and γ are the coefficients for C_V, C_M and $C_I(C)$ and are dependant on the nature of software component.

$C_V(C)$ is the complexity of the variables defined in the component. Variables may consist of member variables having scope for the entire class and the parameters, which are local to a particular method. This may be defined as

$$C_V(C) = \sum_{i=1}^{N} w_i V_i$$

where N is the total number of variables in the component and

w_i is the corresponding weight value of the variable V_i.

Similarly, C_M (C) is the rate of complexity of the methods given in the component and is given as

$$C_M(C) = \sum_{j=1}^{M} w_j M_j$$

where M is the total number of methods in the component and w_j is the corresponding weight value of the method M_j.

Lastly, C_I(C) is the rate of complexity due to the interface methods used in the components. Interfaces are the access points of component, through which a component can request a service declared in an interface of the service providing component. C_I(C) is defined as

$$C_I(C) = \sum_{k=1}^{L} w_k C_k$$

where L is the total number of interface methods in the component and w_k is the corresponding weight value of the method C_k. Therefore the complexity of the component will be

$$CC = \alpha \sum_{i=1}^{N} w_i V_i + \beta \sum_{j=1}^{M} w_j M_j + \gamma \sum_{k=1}^{L} w_k C_k$$

The weight values of all the parts (variables, methods and interfaces) can be assigned on the basis of complexity involved and their nature. We have categorized variables into three categories; primitive, structured and enumerated. Primitive variables are the variables, which are of primitive data type such as *int*. User defined/derived variables having derived data types such as string and date. Last are enumerated variables having complex nature like link list, stack and queue etc. These variables are put into three categories called simple, medium and complex which may have different weight values.

Methods are categorized on the basis of their arguments and return types. Arguments and return types can have any of the three data types discussed earlier (primitive, user defined and structured/class). The following table categorizes methods into four categories, called simple, medium, complex and highly complex. The weight values can be assigned to these methods by considering the total number of methods in each category.

Arguments → / Return Type ↓	No	Primitive	Structured	Enumerated
No	S	S	M	C
Primitive	S	S	M	M
Structured	S	M	M	C
Enumerated	M	M	C	HC

Classification of Methods

Classes contained in a component are derived into base class and derived classes. Base classes are imported classes from other reused library or packages. Derived classes are identified classes during component design in a domain. For the experimentation, we have restricted this inheritance only upto one level. Classes can be categorized on the basis of methods and attributes used in the class. The weight values to these classes are assigned on the basis of total number of methods and variables used in that class.

4. THEORETICAL EVALUATION OF PROPOSED METRIC USING WEYUKER'S PROPERTIES

Weyuker has proposed an axiomatic framework for evaluating complexity measures [8]. The properties are not without critique and these have been discussed in various literatures. The properties, however, have been used to validate the C-K metrics by Chidamber & Kemerer [6] and, as a consequence, we will employ the same framework for compatibility's sake. We show the modified properties below; the original definitions are available at [8]. The properties are:

Property 1: There are programs P and Q for which $M(P) \neq M(Q)$.

Property 2: If c is non-negative number, then there are only finitely many programs P for which M(P)=c

Property 3: There are distinct programs P and Q for which M(P)=M(Q)

Property 4: There are functionally equivalent programs P and Q for which $M(P) \neq M(Q)$

Property 5: For any program bodies P and Q, we have $M(P) \leq M(P;Q)$ and $M(Q) \leq M(P;Q)$.

Property 6: There exist program bodies P, Q and R such that $M(P)=M(Q)$ and $M(P;R) \neq M(Q;R)$.

Property 7: There are program bodies P and Q such that Q is formed by permuting the order of statements of P and $M(P) \neq M(Q)$.

Property 8: If P is a renaming of Q, then $M(P) = M(Q)$.

Property 9: There exist program bodies P and Q such that $M(P)+M(Q) < M(P;Q)$.

We evaluate these properties for our proposed metric.

1. As per the assumptions made above, a component comprises of various design constituents like interfaces, methods and variables, which may always have different complexities, thus satisfying first property.

2. As a component will have only the finite number of methods and variables, which always will have a finite value of the complexities, thus resulting a finite complexity for the entire component.

3. There may always be two distinct components having the same complexities thus satisfying the third property. We can have the same assumptions for the constituents of the components also.

4. There may be two methods, which have the same functionality but with different logic and algorithm thus will have the different complexities. The same thing may also exist for two different components with the same functionality but having different complexities, as these components may be designed by using different technologies and programming concepts.

5. If we increase the functionality of a method by adding some logic to it, it may increase the complexity as compared to the original method, thus satisfying property 5 for the proposed metric.

6. For two methods with different functionality but having same complexities, it may always be possible to extend for some common functionality. But in result the newly developed constituents now may have the different complexities due to for example different type of interactions, values returned etc. This ensures for 6th property.

7. Permutation on component's constituents does not affect on the metric value. Therefore it satisfies 7th property.

8. It is obvious that renaming a method or variable will not affect the complexity of that method or interface, thus satisfying this property.

9. In an object-oriented perspective, modularity reduces the complexity. Thus the total complexities of the two modules will be lesser than the complexity of the combined module, which satisfies the last property.

5. EMPIRICAL EVALUATION

To get the values of the above metrics, an experiment is conducted on various JavaBean components (from www.componentsource.com and www.acme.com). These JavaBean components have different LOC, number of methods, attributes etc. The weight values for variables and methods are assigned on the basis of total number of variables/methods in that class and is given in the following table. 20 is suggested as an upper limit for methods in a nominal class [9] so we are assuming that more than 20 methods or variables in a component will have the highest weight value in each of the category discussed.

Category ▶ / Number ▼	Simple	Medium	Complex	Highly Complex
1-5	0.05	0.30	0.55	0.80
5-10	0.10	0.35	0.60	0.50
10-15	0.15	0.40	0.65	0.90
15-20	0.20	0.45	0.70	0.95
>20	0.25	0.50	0.75	1.0

Weight values for variables and methods

Component	Class	Methods	Attributes	$C_V(C)$	$C_M(C)$	$C_I(C)$	CC
ACME01	2	5	1	0.30	0.5	0.6	1.4
SimpleBean	2	5	2	1.1	0.25	0.6	1.95
WordBean	2	11	2	0.35	1.55	0.35	2.25
ACME02	2	7	3	0.9	1.1	0.6	2.6
ACME03	2	7	3	0.9	1.1	0.6	2.6
DocBean	2	8	5	2.5	1.4	0.35	4.45
ACME04	2	16	9	2.4	3.35	0.85	6.7
ACME05	2	26	11	2.50	5.6	0.85	8.95
GameBean	2	36	10	0.75	0.3	0.32	11.65
ACME06	2	28	12	3.05	7.5	1.1	11.65
ACME07	2	43	14	4.0	14.75	1.35	20.1
ACME08	2	43	15	4.35	14.75	1.35	20.45

Results of Complexity Metric

These weight values are used to compute the complexity metric defined above. The above table gives the value of the complexity metrics on these components.

6. VALIDATION

To validate the proposed metric, we considered a metric called Rate of Component Customizability (RCC) defined by Washizaki et. al. [10]. RCC(C) is the percentage of writable properties in all attributes in a class of a component. It is given by

$$RCC(C) = \begin{cases} \dfrac{P_w(C)}{A(C)} & A(C) > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $P_w(C)$ is the number of writable properties in C and can be measured by counting the setter methods used in the JavaBean component.

A(C) is the number of attributes in C.

The same JavaBean components are used to get the value of this metric and the result obtained is given in table:

Component	RCC(C)
ACME01	1
SimpleBean	0.5
WordBean	0.5
ACME02	0.5
ACME03	0.5
DocBean	0.375
ACME04	0.375
ACME05	0.375
GameBean	0.33
ACME06	0.33
ACME07	0.5
ACME08	0.44

Results of RCC Metric

A correlation analysis was carried out for complexity metric (CC) and Rate of Component Customizability (RCC) by using the Karl Pearson Coefficient of Correlation. The correlation coefficient between CC and RCC is -0.31, which shows a negative correlation between these two metrics.

The result justifies that high complexity leads to the low customizability thus results in high maintainability. The proposed metric seems to be logical and fits into the empirical evaluation also but may not be the sole criteria for deciding the complexity of the software component on the basis of the computed values of this metric. The empirical evaluation is restricted to only one level of inheritance and ignores the complexity involved due to the multi-level inheritance. Moreover this metric involves only the design issues of the component and does not consider the packaging and the deployment complexity.

7. FUTURE WORK

The proposed metric seems to be logical and fits into the empirical evaluation also but may not be the sole criteria for deciding the complexity of the software component on the basis of the computed values of this metric. Moreover this metric involves only the design issues of the component and does not consider the packaging and the deployment complexity. The proposed

work is preliminary and more work is required towards the empirical validation of this metric for some of the other existing component models like Enterprise Java Beans, .NET etc, so that a confidence can be established on this metric to be used for quality development.

8. CONCLUSION

Several previous papers [2,3,5,7,9] have discussed the maintainability and complexity for procedure- oriented systems and object-oriented systems. Not much work has been done to evaluate quality metrics for components and component-based systems. The present work assumes that the complexity of the whole system can be considerably reduced if the component(s) used in that system is/are not so complex. Paper proposes a metric to measure the complexity of software components, which is evaluated theoretically by standard Weyuker's properties. Higher complexity leads to the high cost of maintainability. It is very difficult to customize an application, which is highly complex. The work conducts an empirical study on various JavaBeans components and ensures the same assumption for components also.

8. REFERENCES

[1] Sedigh Ali, S Gafoor, A. Paul, Raymond A., "Software Engineering Metrics for COTS-based Systems", IEEE Computer, May 2001. pp 44-50

[2] McCabe T, "A Software Complexity Measure", IEEE Trans. Software Engineering SE-2 (4), 1976, 308-320.

[3] Kafura, D. and S. Henry, "Software Quality Metrics Based on Interconnectivity", Journal of Systems and Software, June 1981, pp 121-131

[4] Arun Sharma, P S Grover, Rajesh Kumar, "Classification of component metrics", International Conference on Software Engineering Research and Practices (SERP) June 2005.

[5] Li, Henry, "Object-oriented metrics that predict maintainability", Journal of Systems and Software 1993, Volume 23 Issue 2, pg: 111-122

[6] Chidamber, Shyam and Kemerer, Chris, "A metrics Suite for Object-oriented Design", IEEE Transactions on Software Engineering, June 1994, pp. 476-492

[7] Nasib S. Gill, P. S. Grover: "Few important considerations for deriving interface complexity metric for component-based systems", ACM SIGSOFT Software Engineering Notes**,** March 2004 Volume 29 Issue 2

[8] E.J. Weyuker: "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, September 1988 (Vol. 14, No. 9) pp. 1357-1365

[9] Robert V. Binder, “Design for Testability in Object-oriented Systems”, Communications of the ACM, September 1994, Vol. 37, No. 9, pp 87-100

[10] Hironori Washizaki, Hirokazu Yamamoto and Yoshiaki Fukazawa: "A Metrics Suite for Measuring Reusability of Software Components", Proceedings of the 9th International Symposium on Software Metrics September 2003