World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:2, No:6, 2008

# A Complexity Measure for JavaBean based Software Components

Sandeep Khimta, Parvinder S. Sandhu, and Amanpreet Singh Brar

***Abstract*—**The traditional software product and process metrics are neither suitable nor sufficient in measuring the complexity of software components, which ultimately is necessary for quality and productivity improvement within organizations adopting CBSE. 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. In this proposed study it is proposed to find the complexity of the JavaBean Software Components as a reflection of its quality and the component can be adopted accordingly to make it more reusable. The proposed metric involves only the design issues of the component and does not consider the packaging and the deployment complexity. In this way, the software components could be kept in certain limit which in turn help in enhancing the quality and productivity.

***Keywords*—**JavaBean Components, Complexity, Metrics, Validation.

## I. Introduction

A software component is a system element offering a predefined service or event, and able to communicate with other components. Clemens Szyperski [1] and David Messerschmitt give the following five criteria for what a software component shall be to fulfil the definition:

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated i.e., non-investigable through its interfaces
- A unit of independent deployment and versioning.

A simpler definition can be: A component is an object written to a specification. It does not matter what the specification is: COM, Enterprise JavaBeans, etc., as long as the object adheres to the specification. It is only by adhering to the specification that the object becomes a component and gains features such as reusability.

When a component is to be accessed or shared across execution contexts or network links, techniques such as serialization or marshalling are often employed to deliver the component to its destination. Reusability is an important characteristic of a high quality software component. A software component should be designed and implemented so that it can be reused in many different programs.

Benefits of Component-based development include:

- Lower cost of development and shorter delivery schedules
- Better reliability and reduced maintenance costs
- Lets developers focus on their business requirements and core competencies, rather than re-solving the same technical problems over and over
- Provides extensibility because components can be assembled into many different configurations to provide unique variants of a system as needed. (This is especially common today for industries such as cellular technology, consumer electronics, and automotive systems)
- Components that use different languages and technologies can be mixed and matched
- Higher level models make complex systems easier to understand: component based development is the best technique for managing complexity of systems as they increase in size and scope.

Measuring the complexity of software is helpful during analyzing, testing, and maintaining the system. This measurement could direct the process of improvement and reengineering work. A complexity measure could also be used as a predictor of the effort that is needed to maintain the system. In component-based systems, functionalities are not performed within one component. Components communicate and share information in order to provide system functionalities. So to measure the complexity of component-based systems we require metrics that consider component's interfaces and component's relations apart from its internal codes. It is clear from the study of the existing literature that researchers have proposed a wide range of complexity metrics for software systems.

The traditional software product and process metrics are neither suitable nor sufficient in measuring the complexity of software components, which ultimately is necessary for quality and productivity improvement within organisations adopting CBSE. Researchers have proposed a wide range of complexity metrics for software systems [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. In this proposed study it is proposed to find the complexity of the JavaBean Software Components as

Parvinder S. Sandhu is Professor with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali, Punjab-140104, India (phone: +91-98555-32004; e-mail: parvinder.sandhu@gmail.com).

Sandeep Khimta is Lecturer with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali , Punjab, India

Amanpreet Singh Brar is Asstt. Professor & Head (Computer Science & Engineering Department), Guru Nanak Dev Engg. College, Ludhiana, Punjab, India.

a reflection of its quality and the component can be adopted accordingly to make it more reusable. In this way, the software components could be kept simple which in turn help in enhancing the quality and productivity.

## II. Proposed Complexity Metric

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, four elements are taken into consideration to propose the new metric.

The first element, *V(Component$_x$)* is the *Variable Factor* that tells complexity of the variables defined in the component. Variables may consist of member variables of a class having scope for the entire class and the parameters, which are local to a particular method. This may be defined as shown in the following equation:

$$V(Component_x) = \sum_{i=1}^{n1} w_{simple_i} + \sum_{j=1}^{n2} w_{medium_j} + \sum_{k=1}^{n3} w_{complex_k}$$

Where *n1, n2* and *n3* are the total number of simple, medium and complex variables in the *Component$_X$*. Here, $w_{simple_i}$ , $w_{medium_j}$ and $w_{complex_k}$ are the corresponding weight value of the simple, medium and complex variables variable respectively. Variables are categorized into three categories; *primitive, user defined and structured.*

*Primitive variables* are the variables, which are of primitive data type such as int, float, char, double, long etc.

*User defined variables* having derived data types such as string, date etc.

*Structured variables* having complex nature like link list, stack, queue etc.

These variables are put into three categories called simple, medium and complex which may have different weight values as a contribution towards the overall complexity of the class.

The second element, *I(Component$_x$)* is the *Interface Factor* that tells complexity of 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. Mathematically, *I(Component$_x$)* is defined as sum of complexity of the interface methods of the class. The complexity of interface methods depends on its nature. The nature of the interface methods are determined 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). The weight values can be assigned to these methods by considering the total number of methods in each category. The different category methods have different value of weight. Weight of the method also depends on the number of methods in that category. If the number of methods are more then the weight value assigned will also increase. Now, Mathematically the Interface Factor, *I(Component$_x$),* can be written as:

$$I(Component_x) = \sum_{i=1}^{m1} w_{simple_i} + \sum_{j=1}^{m1} w_{medium_j} + \sum_{k=1}^{m3} w_{complex_k}$$

Where *m1, m2* and *m3* are the total number of interface methods of simple, medium and complex nature respectively. Here, $w_{simple_i}$ , $w_{medium_j}$ and $w_{complex_k}$ are the corresponding weight value of the simple, medium and complex nature of interface methods respectively.

Third element, *C(Component$_x$)* is the *Coupling Factor* that tells rate of coupling of the methods in the component and defined in terms of ratio of number of other methods called in the methods of the component and total number of declared methods in the component.

$$C(component_x) = \frac{O}{D}$$

Where *O* is total number of other methods being called in the methods of the *Component$_x$* and *D* is the total number of declared methods in the *Component$_x$*.

Fourth element, *CC(Component$_x$)* is cyclometric complexity of the methods of the *Component$_x$*.

The complexity measure approach taken is to measure and control the number of paths through a program. This approach, however, immediately raises the following nasty problem: "Any program with a backward branch potentially has an infinite number of paths." It is possible to define a set of algebraic expressions that give the total number of possible paths through a (structured) program, using the total number of paths has been found to be impractical. Because of this the complexity measure developed here is defied in terms of basic paths-that when taken in combination will generate every possible path [3]. The conditional constructs are calculated to express the cyclometric complexity of a Method of a class.

Therefore the complexity of the component is sum of all the four elements defined above:

$$Complexity(C_x) = V(C_x) + I(C_x) + C(C_x) + CC(C_x)$$

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, this inheritance is restricted 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.

## III. Results & Discussion

To get the values of the above metrics, an experiment is conducted on twenty JavaBean components collected from the open source repositories and results of the calculated complexity is given in Table I where TV stands for Total Variables, TDM stands for Total Declared Methods, TIM stands for the total Interface Methods in the component.

TABLE I
COMPLEXITY RESULTS OF THE EXAMPLE COMPONENTS

| *TV* | *TDM* | *TIM* | *V ($C_x$)* | *I ($C_x$)* | *C($C_x$)* | *CC ($C_x$)* | *Complexity ($C_x$)* |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 0 | 0.5 | 1.5 | 0 | 2 |
| 3 | 4 | 4 | 0 | 1 | 1 | 0 | 2 |
| 3 | 4 | 4 | 0 | 1 | 1 | 0 | 2 |
| 8 | 6 | 6 | 2.9 | 1.5 | 2.16 | 0 | 6.5667 |
| 12 | 10 | 9 | 4.98 | 2.1 | 2.3 | 1 | 10.38 |
| 12 | 11 | 10 | 5.34 | 2.5 | 2.18 | 3 | 13.022 |
| 22 | 16 | 15 | 9.58 | 4.2 | 2.18 | 7 | 22.967 |
| 21 | 16 | 15 | 9.48 | 4.44 | 2.12 | 8 | 24.045 |
| 25 | 23 | 22 | 10.7 | 6.34 | 2.08 | 8 | 27.207 |
| 1 | 2 | 2 | 0 | 0.5 | 1 | 0 | 1.5 |
| 3 | 3 | 3 | 0 | 0.9 | 1 | 0 | 1.9 |
| 3 | 3 | 3 | 0 | 0.9 | 1 | 0 | 1.9 |
| 8 | 5 | 5 | 2.9 | 1.4 | 2 | 0 | 6.3 |
| 12 | 9 | 8 | 4.98 | 2 | 2.44 | 1 | 10.424 |
| 12 | 10 | 9 | 5.34 | 2.4 | 2.3 | 3 | 13.04 |
| 21 | 14 | 13 | 9.06 | 3.58 | 2.28 | 7 | 21.926 |
| 20 | 14 | 13 | 8.96 | 3.82 | 2.21 | 8 | 22.994 |
| 24 | 21 | 20 | 10.2 | 5.6 | 2.09 | 8 | 25.895 |
| 2 | 4 | 4 | 0 | 1 | 1.25 | 0 | 2.25 |
| 3 | 7 | 7 | 0 | 1.6 | 1.42 | 1 | 4.0286 |

TABLE II
COMPARATIVE RESULTS OF RCC & PROPOSED METRIC

| *Component No.* | *Complexity($C_i$)* | *RCC ($C_i$)* |
|---|---|---|
| 1 | 2 | 0 |
| 2 | 2 | 2 |
| 3 | 2 | 2 |
| 4 | 6.5667 | 2.5 |
| 5 | 10.38 | 1.5 |
| 6 | 13.022 | 1.5 |
| 7 | 22.967 | 1.1667 |
| 8 | 24.045 | 1 |
| 9 | 27.207 | 0.875 |
| 10 | 1.5 | 0 |
| 11 | 1.9 | 2 |
| 12 | 1.9 | 2 |
| 13 | 6.3 | 2 |
| 14 | 10.424 | 1.5 |
| 15 | 13.04 | 1.5 |
| 16 | 21.926 | 1 |
| 17 | 22.994 | 0.85714 |
| 18 | 25.895 | 0.85714 |
| 19 | 2.25 | 4 |
| 20 | 4.0286 | 0 |

These weight values are used to compute the proposed complexity metric defined in the last section. The implementation of the complexity metric calculation is performed in the MATLAB 7.4 that make use of the regular expression to parse the code to generate the essential information needed for the mathematical formulae of the proposed complexity metric. Table I gives the value of the complexity metrics on these components along with the other information such as: *Total Variables*, *Total Interface Methods*, *Total Declared methods*, *Variable Factor, Interface Factor, Coupling Factor* and *Cyclometric Complexity Factor* of the proposed metric.

To validate the proposed metric, a metric called Rate of Component Customizability (*RCC*) defined by Washizaki et. al. [10] is used. Metric *RCC(C)* is the percentage of writable properties in all attributes in a class of a component. The same JavaBean components are used to get the value of this metric and the result obtained is given in Table II.

A correlation analysis was carried out for complexity metric *Complexity($C_i$)* and Rate of Component Customizability *RCC($C_i$)*by using the Karl Pearson Coefficient of Correlation. The correlation coefficient between *Complexity($C_i$)* and *RCC($C_i$)* is -0.301, which shows a negative correlation between these two metrics.

## IV. CONCLUSION

The result shows that there exists inversely proportional relation between the Rate of Component Customizability and newly proposed complexity metric. When we interpret it means 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. But, the above empirical evaluation is restricted to only one level of inheritance; it ignores the complexity involved due to the multi-level inheritance; it involves only the design issues of the component and does not consider the packaging and the deployment complexity. So, in future the metric can be extended and more dimensions can be added in more comprehensive complexity measure of JavaBean components.

## REFERENCES

[1] Clemens Szyperski, Component Software: Beyond Object-Oriented Programming. 2nd ed. Addison-Wesley Professional, Boston 2002.
[2] Sedigh Ali, S Gafoor, A. Paul, Raymond A., "Software Engineering Metrics for COTS-based Systems", IEEE Computer, May 2001. pp 44-50
[3] T. McCabe, "A Software Complexity Measure", IEEE Trans. Software Engineering SE-2 (4), 1976, 308-320.
[4] D. Kafura, S. Henry, "Software Quality Metrics Based on Interconnectivity", Journal of Systems and Software, June 1981, pp 121-131
[5] H. Li, "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] M. Bertoa, A. Vallecillo, “Quality Attributes for COTS Components”, 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002), Málaga, Spain. (2002)

[9] M. Bertoa, A. Vallecillo, “Usability metrics for software components”, QAOOSE 2004, Oslo. (2004)

[10] N. S. Gill, P. S. Grover, "Component-Based Measurement: Few Useful Guidelines." ACM SIGSOFT Software Engineering Notes 28(6) (2003).

[11] R. Dumke, A. Schmietendorf, "Possibilities of the Description and Evaluation of Software Components." Metrics News 5(1) (2000).

[12] M. A. Boxall, S. Araban, “Interface Metrics for Reusability Analysis of Components”, Australian Software Engineering Conference (ASWEC'2004), Melbourne, Australia. (2004)

[13] H. Washizaki, H. Yamamoto, Y. A. Fukazawa, “Metrics Suite for Measuring Reusability of Software Components”, Metrics'2003. (2003)

[14] M. Goulão, F. B. Abreu, “Independent Validation of a Component Metrics Suite”, IX Jornadas de Ingeniería del Software y Bases de Datos, Málaga, Spain. (2004)

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