A Metric for Accessing Black Box Component Reusability

Navneet Kaur , Ashima Singh

Abstract— The Component Based Software Development (CBSD) approach is becoming the trend for software development. This approach is based on developing the software from existing components instead of developing software from scratch everytime. The quality of resulting system depends upon the complexity of the composed components. Because the component complexity is an important factor affecting the understandability, testability, maintainability of resulting system. So it is necessary to select the less complex components which are more reusable, for Component Based Software system. Thus evaluation of component complexity is a critical activity in the component selection process for CBSD. Although the researchers have proposed a wide range of metrics for evaluating component complexity but many of the existing metrics are not appropriate for measuring component complexity due to component's black box nature. Thus in this paper an Interface Complexity metric for Black Box components, IC(BB), has been proposed which is based on component interface specifications.

Index Terms— Black box component, CBSD, Component Complexity, complexity metrics, IC(BB), software complexity.

1. INTRODUCTION

THE Component Based Software Development (CBSD) approach is increasingly being adopted for software development. CBSD approach is based on using the existing components as building blocks for constructing software systems. CBSD provides many advantages like reduced development time and effort, increased quality along with many others. These advantages are mainly provided by the reuse of already built-in software components. The following Fig.1 shows the technique for developing software from existing components.

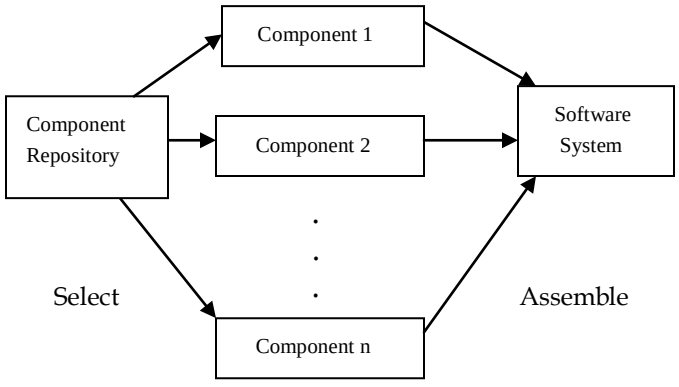

Fig. 1 Component based software development

But it is necessary to measure the software complexity in each software development approach because software complexity affects many other aspects of software like development effort,cost, testability,maintainability etc. So many metrics have been proposed for measuring software complexity. But traditional software product and process metrics are not sufficient for measuring the component and Component Based Software (CBS) complexity and most of the existing metrics are based on source code. Thus CBSD provides one of the central problems in measuring component and CBS complexity. Measuring component complexity plays an important role in determining CBS system complexity. Because complexity of CBS system depends upon the complexity of its components . The component complexity is an important factor affecting the understandability, testability, maintainability etc of CBS system . So it is necessary to select the less complex components which are more reusable for CBS system. But now a days black box components are being provided by component vendors for reuse and most of the times source code is not provided with components which creates difficulty in measuring component complexity. In this paper a complexity metric named, Interface Complexity metric for Black Box components, IC (BB) has been proposed. The proposed metric is based on the component interface specification and
concept of assigned weights.

2. BRIEF STUDY OF EXISTING METRICS

In this section some existing complexity metrics have been discussed that are relevant for measuring component complexity.

- *Navneet Kaur is currently pursuing masters degree in M.E.(CSE) in Thapar University, India, E-mail: navirathour27@gmail.com*
- *Ashima Singh is an Assistant Professor in Computer Science and Engineering Department , Thapar University, India, E-mail: ashima@thapar.edu*

2.1 Object Oriented Metrics

There are many object oriented metrics that can be used to measure the component complexity . Some of the existing metrics[6,16], have been discussed below:

Metric 1: Weighted Methods Per Class (WMC)
WMC gives the combined complexity of local methods in a given class. The greater value of this metric shows more complexity, increase in testing effort and decrease in understandability.

Metric 2: Depth of Inheritance (DIT)
DIT metric is for class . It gives maximum length from the class node to root. More length means more complexity.

Metric 3: Response For Class (RFC)
The RFC metric gives the number of methods that can be invoked in response to a message sent to an object within this class ,using to one level of nesting.

Metric 4: Coupling Between Objects (CBO)
For a given class, this metric measures the number of other classes to which the class is coupled. High value of this metric shows the poor design, difficulty in understanding, decrease in reuse and increase in maintenance effort.

Metric 5: Lack of Cohesion Method (LCOM)
The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. LCOM is defined as the number of disjoint sets of local methods. High value of this metric shows good class subdivision.

Metric 6: Number of Children (NOC)
NOC is based on a node (class) of inheritance tree. This metric gives the number of immediate successors of the considered class. High value of this metric shows more reuse, poor design and increase in testing effort.

Metric 7: Lines of Code (LOC)
LOC is based on the size of methods. It gives measure of physical lines , statements , and/or comments. High value of this metric shows more complexity .

Metric 8: Cyclomatic Complexity (CC)
Cyclomatic Complexity measures the complexity of methods. It gives the measure of independent algorithmic test paths. More independent paths means more testing effrot.

2.2 Metrics for the Integration of Software Components

Narasimhan and Hendradjaya proposed the following complexity metrics[8] that have been widely accepted .

a) Metric 1: Component Packing Density (CPD)
The CPD metric measures the component constituents to the number of integrated components. This metric is used to identify the density of integrated components. Thus, a higher density represents a higher complexity.

$$CPD<\text{constituent\_type}> = \frac{\#<\text{Constituent}>}{\#\ \text{Components}}$$

Where #<Constituent> is the number of lines of code, operations, classes, and/or modules in the related components.

b) Metric 2: Component Interaction Density (CID)
The CID metric measures the ratio of actual number of interactions to the available number of interactions in a component.

$$CID = \frac{\#I}{\#\ Imax}$$

Where #I and #Imax represents the number of actual interactions and maximum available interactions respectively. When the density of interaction increases, complexity increases.

Metric 3: Component Incoming Interaction Density (CIID)
The CIID metric measures the ratio of actual number of incoming interactions to the maximum available incoming interactions in a component.

$$CIID = \frac{\#\ Iin}{\#\ Imax\_in}$$

Where # Iin and # Imax_in represents the actual number of incoming interactions and maximum number of incoming interactions available in a component respectively . High density shows that a particular component requires so many interfaces.

Metric 4: Component Outgoing Interaction Density (COID)
The COID metric measures the ratio of actual number of outgoing interactions to the maximum number of outgoing interactions available in a component.

$$COID = \frac{\#\ Iout}{\#\ Imax\_out}$$

Where # Iout and # Imax_out represents the actual number of outgoing interactions used and maximum number of outgoing interactions available in a component respectively.

Metric 5: Component Average Interaction Density (CAID)
The CAID metric is a sum of interaction densities for each component divided by the number of components in software system .

$$CAID = \sum_{i=1}^{n} \frac{CIDn}{\#\ Components}$$

Where, ∑ CIDn represents the sum of interaction densities for components 1...n and # components represents the number of existing components in the software system.

c) Criticality Metrics

Metric 6: Link Criticality Metric (CRITlink)
Link Criticality metric is defined as the number of components which have links more than a threshold value.

CRITlink = # linkcomponents

Where # linkcomponents represents the number of components, with their links more than a critical value. The threshold is considered as 8 links.

Metric 7: Bridge Criticality Metric (CRITbridge)
Bridge Criticality metric is defined as the number of bridge components in a component assembly.

CRITbridge = # bridge_component

Where # bridge_component represents the number of bridge components . A bridge component may be defined as a component which links two or more components/ application. If there is a defect in bridge, the whole application might malfunction. More number of bridge components means more chances of failure.

Metric 8: Inheritance Criticality Metric (CRITinheritance)
Inheritance Criticality metric is defined as the number of components, which become root or base for other inherited components.

CRITinheritance = # root _ component

Where # root_component represents the number of root components which has inheritance. It is the number of components which act as a parent/root/base for other components .

Metric 9: Size Criticality Metric (CRITsize)
Size Criticality metric is defined as below :

CRITsize = # size_component

Where # size_component represents the number of components which exceed a given critical size value. The size is defined in terms of LOC, number of classes, operations and modules in the application.

Metric 10: # Criticality Metric
The #Criticality Metric (CRITall) is defined as the sum of all critical metrics.

CRITall = CRITlink + CRITbridge + CRITinheritance + CRIT-size

d) Triangular Metrics
Component Packing Density (CPD) , Component Average Interaction Density (CAID), Component Criticality (CRITall) metrics are considered as 3 axes which can be further modified as 2 axes diagrams with CPD and CAID. For different values varying as high and low for the 2 axes, different cases are considered as the behaviors vary for different systems based on real time, business type etc.

e) Dynamic Metrics
These metrics are collected during the execution time. These are not available during the design phase as they are collected dynamically. These metrics are used for maintenance purposes.

2.3 Limitations of Existing Metrics

- Most of the existing metrics are applicable to small programs or components, The objective of having metrics is to test the behavior, reusability, and reliability of the components when placed in a large system.
- Some metrics like WMC, CC, LOC, CRITsize etc depend upon the availability of source code or internal details of component , these kind of metrics can not be applied for determining black box component complexity because of unavailability of source code. So there is a need of complexity metric for black box component because a number of existing metrics can not be applied directly.

In this paper a metric has been proposed which measures the complexity of a black box component on the basis of component interface specification.

3. PROPOSED WORK

Interfaces are the access points of a component, through which a component can request a service declared in an interface of the service providing component. Mathematically, interface complexity is defined as sum of complexity of the interface methods. The complexity of interface method depends on its nature. The nature of the interface method can be determined on the basis of number and type of arguments and return type. Rotaru et al.(2005) considered the interface methods complexity to determine the composability of the component. The components interfaced by methods having no return value and no parameter will have biggest composability degree because it does not have any external dependency. The interface methods having no parameter value but having return value will have less composability degree and the interface methods having the parameters as well as return value will

cause in lowest composability degree.

We extended the approaches described in(Rotaru et al., 2005; Gill and Grover, 2004; Boxall and Araban,2004;) while proposing a new interface method complexity metric for components. We propose that the interface method complexity depends upon the return type, number and type of parameters and the number of parameter incompatibilities which arise when the parameters are passed between the components. Thus an Interface Method Complexity Metric (IMCM) has been proposed as below :

IMCM = Wr + PCM(M) + Number of parameter incompatibilities

Where Wr is the weight assigned to return type , PCM(M) is the Parameter Complexity Metric for method , which determine the complexity caused by parameters of method.

$$PCM(M) = \sum_{i=1}^{n} Wp(Pi)$$

Where Wp(Pi) is the weight assigned to the ith parameter of the method on the basis of its data type , n represents the number of parameters in a method.

Thus by using the mathematical definition of Interface Complexity, a metric named Interface Complexity metric for Black Box components ,IC(BB) , has been proposed for determining the interface complexity.

$$IC(BB) = \sum_{i=1}^{m} IMCMi$$

Where IMCMi is the interface method complexity of ith method in interface and m represents the number of methods in component interface.

The interface methods can be divided in the following categories:

- Interface methods having no return value and no parameters.
- Interface methods having return value but no parameters .
- Interface methods having no return value but having parameters.
- Interface methods having return value as well as parameters

The complexity of the interface methods can be measured on the basis of data types of return value and parameters, and on the basis of number of parameter incompatibilties . On the basis of data types of return value and parameters , and by considering the number of parameter incompatibilities for a method , some weight values will be assigned to the interface method which will show its complexity.

The data types can be divided in the following categories:

- Very simple includes integer,float,double,boolean etc.
- Simple includes structure data types.
- Medium includes class type and object type.
- Complex includes pointer and built in data types.
- Very complex includes user defined data types.

The methods having no return value and no parameters have been considered as simple methods and their weight value has been assumed .025 . All other interface methods are assigned weight values depending on the data types of parameters and return value . The Table I represents the weight values assigned to different categories of data types for parameters and return values.

We have included a factor in the existing approaches that affects the complexity of interface methods and it will decrease the composability. This factor is parameter incompatibility. Because when the components are integrated with each other then one component may pass the parameter to the another component's function but some times the data type of the passed parameter may be different from the data type of parameter declared in the function to which the parameter is passed. Then there will be parameter incompatibility problem

Table I. Represents the assigned weight values to the different categories of data types

Parameter Type ,Return Value Type →	Very Simple	Simple	Medium	Complex	Very Complex
Assigned Weight ↓	.10	.20	.30	.40	.50

.

For example, suppose return value of one component's meth-

od is passed to the another component's method as a parameter to perform its task, but if their data types are different then there will be parameter incompatibility problem. So the return value must be converted in the required form before passing as a parameter to second component's method(i.e it needs adaption.). More number of incompatibilities result in more difficulty for using component interface method. It will reduce the understandability of method's behavior but it will increase integration complexity to connect the component with other components to provide accurate functionality. Thus it will be more difficult to use the component.

4. CASE STUDY

In order to validate the proposed metric, we have considered a case study of Student Information (SI) system from which the students of different departments can receive information about their marks details, fee details and course details . This system has been developed by integrating the components. This system has been represented in the form of class diagram as shown below in Fig.2. The class name shows the component name and we have considered only the business methods in our case study. For simplicity we have considered only one parameter in the business methods, in order to validate the metric.

Description of working of Student Information System

The system is composed of eight components and their working has been described as below:

Login Component : This component checks the password entered by the user in order to authenticate the user. If the password is correct then it will return value 1 otherwise it will return value 0. The return value will be passed to the second and third method of component named Student_Info_System.

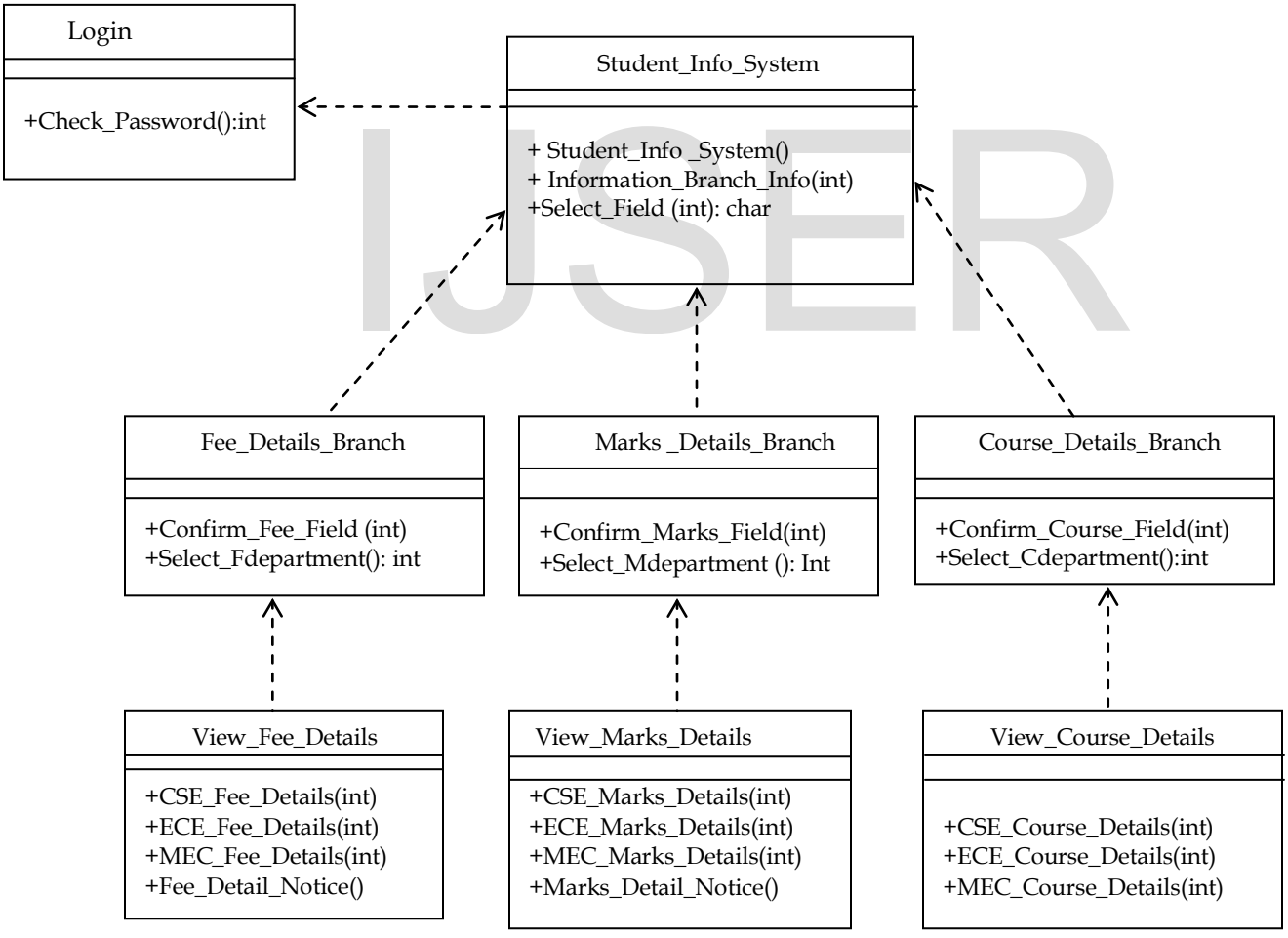

Fig 2. Class diagram of Student Information System

Student_Info_System Component : This component provides the information about the system's working, information about the various information branches .The third method of this component provides the options to the user to select the branch from which the user want to get information. The second and third method will perform their actions only if the password is correct. This component passes F,M,C if user want to get information about fee details, marks details and course details respectively, for the confirmation of the selected branch . But the components Fee_Details_Branch, Marks_Details_Branch and Course_Details_Branch accept the integer value for the confirmation of the selected branch .Thus when this component is used three parameter incompatibilities are caused which will make it difficult to use the method. It will also reduce the composability.

Fee_Details_Branch Component : The first method of this component confirms the selected branch from which the user want to get information. If user has selected fee details branch to get information then it will return 1 other wise 0. Second method of this component displays list of departments from which the user can select any department for which the user want to check fee_details. This component will pass 1,2,3 ,for CSE, ECE, and MEC departments respectively, to the View_Fee_Details Component. When this component is used it will create one parameter incompatibility because component Student_Info_ System passes the char parameter but the Fee_Details_Branch component takes the integer parameter to confirm the selcted branch .

View_Fee_Details Component : 1,2,3 values are passed to this component when the user want to view fee details of CSE, ECE and MEC departments respectively . The Fee_Details_Branch component passes the integer value for the selected department to the View_Fee_Details component and this component also accept the integer value to confirm the selected department. Because the passed parameter and received parameter data types are same so this component does not create any incompatibility problem.

Marks_Details_Branch and Course_Details_Branch Components act same like Fee_Details_Branch component . View_Marks_Details and View_Course_Details components act same like View_Fee_Details component.

Thus from the above information the interface complexity can be calculated for each component in the Student Information System by using the IC(BB) Metric .

Interface Complexity of Student_Info_System Component

IMCM value for first method = 0.025 , because this method is a simple method which does not have any parameter or return value .

IMCM value for second method = 0.10 = 0.10

IMCM value for third method = 0.10 + 0 .10 + 3 = 3. 20

Thus IC(BB) = .025 + 0.10 +3.20 = 3.325

Similarly the interface complexity of other components can be calculated. The following Table II shows the value of IC(BB) for each component in SI system.

Table II. Representing Value of IC(BB) for each component in SI system

Component Name	IC(BB)
Login	0.10
Student _Info_System	3.325
Fee_Details_Branch	1.20
Marks_Details_Branch	1.20
Course_Details_Branch	1.20
View_Fee_Details	0.325
View_Marks_Details	0.325
View_Course_Details	0.30

In order to validate the proposed metric , an another metric named Self-Completeness of Component's Parameter (SCCp) defined by Washizaki et al. [13] has been used . SCCp metric is used in determining the external dependency of Java Beans components which are black box in nature. Because we are considering the black box components in our case study, so this metric is also applicable in our case.

Definition of SCCp : Self-Completeness of Component's Parameter

SCCp(c) is the percentage of business methods without any parameters in all business methods implemented within a component c :

$$SCCp(c) = \begin{cases} \frac{B_p(c)}{B(c)} & (B(c) > 0) \\ 1 & (\text{otherwise}) \end{cases}$$

Where Bp(c) : number of business methods without parameters in c.

SCCp indicates the component's degree of self-completeness, and the low degree of external dependency for users of the

component. Simply, the smaller the number of business methods without parameters, the more the possibility of having dependency outside the component which shows the more

complexity for component usage. The following Table III shows the value of SCCp metric for each component in SI System .

Table III. Representing the value of SCCp metric for each component in SI System

Component Name	SCCp
Login	1
Student_Info_System	0.333
Fee_Details_Branch	0.5
Marks_Details_Branch	0.5
Course_Details_Branch	0.5
	0.25
ails	0.25
ails	0

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

A correlation analysis has been carried out for Interface complexity metric IC(BB) and Self-Completeness of Component's Parameter (SCCp) by using the Karl Pearson Coefficient of Correlation. The formula for calculating the Karl Pearson Correlation Coffecient is as below:

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

This vlaue of Coefficient shows the relation between two variables. The negative value of this Coefficient shows the negative relation between two variables, means increase in one variable decreases the value of another variable and vice versa. The following Table IV shows the calculations for Karl Pearson Coefficient. In our case IC(BB) and SCCp represents X and Y respectively.

Table IV. Representing the calculations for Karl Pearson Coefficient

Component Name	X= IC(BB)	Y= SCCp	X^2	Y^2	XY
Login Component	0.10	1	0.01	1	0.10
Student_I nfo_System	3.325	0.333	11.056	0.111	0.107
Fee_Details_Branch	1.20	0.5	1.44	0.25	0.6
Marks_Details_Branch	1.20	0.5	1.44	0.25	0.6
Course_Details_Branch	1.20	0.5	1.44	0.25	0.6
View_Fee_Details	0.325	0.25	0.106	0.0625	0.08125
View_Marks_Details	0.325	0.25	0.106	0.0625	0.08125
View_Course_Details	0.30	0	0.09	0	0
	$\sum X = 7.98$	$\sum Y = 3.33$	$\sum X^2 = 15.69$	$\sum Y^2 = 1.986$	$\sum XY = 2.1695$

By putting all the values in the formula for Karl Pearson Correlation Coefficient , we get the value of Karl Pearson Correla-

tion Coefficient. In our case the value of Karl Pearson Correlation Coefficient is (- 0.432) which shows the negative relationship between IC(BB) and SCCp. It means if the value of IC(BB) increases then value of SCCp decreases which shows the more external dependencies, which cause more difficulty in component use and decrease its reusability. Thus we can say that the high value of IC(BB) for any component shows more complexity , which causes decrease in reusability, understandability and increase in integration and testing effort.

5. CONCLUSION

Although the Component Based Software Development is increasingly being adopted for software development. But measuring the black box component complexity during component selection , for selecting a less complex and more reusable component , is still a difficult task. Because most of the existing component complexity metrics, as discussed in section II, can not be applied directly for determining the component complexity. So in this paper a metric named Interface Complexity Metric for Black Box components, IC(BB), has been proposed which is based on component interface specification. This metric will help an application developer in selecting a less complex and more reusable component during the selection of components for CBSD. This will help in reducing the integration and testing effort

REFERENCES

[1] Navneet Kaur, Ashima Singh, "Generating More Reusable Components while Development: A Technique ," International Journal of Innovative Technology and Exploring Engineering , Volume-2, Issue-3, February 2013.

[2] Sandeep Khimta, Parvinder S. Sandhu and Amanpreet Singh Brar, "A Complexity Measure for JavaBean based Software Components," World Academy of Science, Engineering and Technology,2008.

[3] Ben Whittle and Mark Ratcliffe, "Software Component Interface Description for Reuse," Software Engineering Journal, November 1993.

[4] Ben Whittle and Mark Ratcliffe, "Software Component Interface Description for Reuse," Software Engineering Journal, November 1993.

[5] Luiz Fernando Capretz and Miriam A. M. Capretz, "Component-Based Software Development," The 27th Annual Conference of the IEEE Industrial Electronics Society,2001.

[6] Chidamber, S. R., Kemerer and C.F, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, pp. 476-49,1994.

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

[8] V. L. Narasimhan and B. Hendradjaya, "A New Suite of Metrics for the Integration of Software Components," University of Newcastle , Australia.

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

[10] Seyyed Mohsen Jamali, "Object Oriented Metrics," Department of Computer Engineering ,Sharif University of Technology, January 2006.

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

[12] Nael Salman, "Complexity Metrics As Predictors of Maintainability and Integrability of Software Components," Journal of Arts and Sciences, 2006.

[13] Hironori Washizaki, Hirokazu Yamamoto and Yoshiaki Fukazawa, "A Metrics Suite for Measuring Reusability of Software Components," Department of Computer Science, Waseda University,Japan.

[14] Parvinder Singh Sandhu and Dr. Hardeep Singh, "A Critical Suggestive Evaluation of CK Metric," Guru Nanak Dev Engineering College, Ludhiana,Punjab.

[15] Rajender Singh Chillar, Priyanka Ahlawat and Usha Kumari, "Measuring Complexity of Component Based System Using Weighted Assignment Technique," 2nd International Conference on Information Communication and Management, Singapore,2012.

[16] P. K. Suri and Neeraj Garg, " Software Reuse Metrics: Measuring Component Independence and its applicability in Software Reuse," IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.5, May 2009.

[17] Octavian Paul Rotaru and Marian Dobre, " Reusability Metrics for Software Components," The Computer Science and Engineering Department , University "Politehnica" of Bucharest, Romania.

[18] Marcus A. S. Boxall and Saeed Araban, "Interface Metrics for Reusability Analysis of Components," Department of Computer Science & Software Engineering, The University of Melbourne,2004.

[19] N. S. Gill and P.S.Grover, " Component-Based Measurement: Few Useful Guidelines," ACM SIGSOFT SEN Volume 28 No. 6, pp. 30,2003.