# An Interface Complexity Measure for Component-based Software Systems

Usha Kumari
Research Scholar
Deptt. of Computer Science,
Kurukshetra University
Kurukshetra, Haryana, India

Shuchita Upadhyaya
Associate Professor
Deptt. of Computer Science,
Kurukshetra University
Kurukshetra, Haryana, India

## ABSTRACT
Controlling and minimizing software complexity is one of 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 component-based software is rising, more and more complexity metrics are being developed for the same. In this paper, we have attempted to design an interface complexity metric for black-box components to quantify an important aspect of complexity of a component-based system. The proposed measure takes into account one major type of complexity of a component. It is due to its interactions (interfaces) with other components. Graph theoretic notions have been used to illustrate interaction among software components and to compute complexity. The proposed measure has been applied to five cases chosen for this study and yields quiet encouraging results which may further help in controlling the complexity of component-based systems so as to minimize both integration and maintenance efforts. As a thumb rule, we propose that average number of interactions (interfaces) per component in a component based system (CBS) should not be greater than five, otherwise that CBS would be highly complex and will be more prone to errors and hence unreliable. However, this rule requires further empirical support.

## Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *complexity metrics*

## General Terms
Measurements

## Keywords
Software measurement, component-based software development (CBSD), component-based system (CBS), component-based software, reusability, reliability, testability, maintainability, black-box components.

## 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 [1,7,9,10,18]. 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 [1, 2, 3, 5, 7, 16, 19]. It is accepted by both software developers and researchers that complexity of software can be controlled more effectively through component-based and object-oriented approach than traditional function-oriented approach. It is because that objected-oriented and component-based paradigms control 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 and component-based software systems due to their distinguish features like class, inheritance, polymorphism, coupling, interface, links and cohesion.

Component-based software development (CBSD) is one of the most important, modern paradigm and is expected to be at the forefront of new approaches to the construction of large and complex software systems [21,24,25]. The main objective of this approach is to minimize the development effort, time and cost by means of reuse. CBSD improves quality, productivity and maintainability of the software [26,27,29]. Due to this, CBSD is likely to become the main and preferred stream for software development. This paradigm focuses on developing large software systems by integrating prefabricated software components. It facilitates the process of software development and solves many adaptation and maintenance problems [28]. It is very much clear and visible that in the last few years research has focused on methods and approaches that work towards developing software systems by integrating already developed components. As the development of component-based software is rising, more and more complexity metrics are being developed for the same. Most of the metrics proposed so far are based on the source code of the component and therefore cannot be used by the application developers, who do not have the source code of these components.

In this paper, we have attempted to design an interface complexity metric for black-box components to quantify an important aspect of complexity of a component-based system. The proposed measure takes into account one major type of complexity of a component. It is due to its interactions (interfaces) with other components. Graph theoretic notions have been used to illustrate interaction among software components and to compute complexity. The proposed measure has been applied to five cases chosen for this study and yields quiet encouraging results which may further help in controlling the complexity of component-based systems so as

to minimize both integration and maintenance efforts. The same has also been theoretically evaluated against Weyuker's properties. As a thumb rule, we propose that average number of interactions (interfaces) per component in a component based system (CBS) should not be greater than five, otherwise that CBS would be highly complex and will be more prone to errors and hence unreliable. However, this rule requires further empirical support.

Rest of this paper is organized as follows: Section 2 presents overview of software complexity and existing complexity measures. Software component definition, its dependency and interaction issues are discussed in section 3. Section 4 describes proposed composite complexity measure. Section 5 reports theoretical evaluation of proposed metric using Weyuker's properties. Section 6 discusses the case study and experimental results. Finally, section 7 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. IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [32]. 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. Traditional software metrics have been designed and applied to the measurement of software complexity of structured systems since 1976 [5, 17, 18, 34]. Halstead's software science metrics, Source line of code, McCabe's cyclomatic number and Kafura's & Henry's fan-in, fan-out, function point analysis, bugs or faults per line of code, code coverage are the best known early reported complexity metrics for traditional function-oriented approach. Traditional software metrics are usually applicable to small programs, whereas the metrics for object-oriented and component-based software systems should depend mainly on the granularity and interoperability aspects of the classes and components. So traditional software complexity metrics are not suitable for measuring complexity of object oriented and component-based software systems.

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.

For component-based systems, complexity metrics reported in [23, 28, 29, 30, 31, 35] are based on complexity attributes like interaction, coupling, cohesion, interface etc. Most of the metrics proposed so far are based on the source code of the

component and therefore cannot be used by the application developers, who do not have the source code of these components. So, there is strong demand and need for designing of complexity metrics for black-box components, which may be used by the application developers to choose the best components and then finally produce better quality CBS. For black-box components, major complexity parameters are interface, integration and semantics. Interface complexity measures are the estimates of the complexity of interfaces. Interface defines provided services of a component and acts as a basis for its use and implementation. It acts as one of the major definitive source for component understanding and may be the only available source. An interface consists of a set of operations, which act as access points for interaction with the outside computing environment. Integration metrics are the measures of efforts required in the integration process of components and semantic measures estimate the complexity of relationship of components to application.

## 3. SOFTWARE COMPONENT DEFINITION, ITS DEPENDENCY AND INTERACTION ISSUES

A software component is a self-contained piece of software that provides clear functionality, has open interfaces and offers plug-and-play services. It can be regarded as a reusable software element such as a function, file, module, class or subsystem. A component-based software system can be obtained as a result of the composition of some components with defined interfaces [21,22]. A component's functionality is implemented in its methods and is provided for other components through its well-defined interfaces/interactions. The dependency among components can be described as the reliance of a component on other component(s) to support a specific functionality or configuration. In highly structured environments, like component-oriented systems, unit of computation (e.g. component) communicate and share information in order to provide system functionalities. Components are composed on regular basis for the purpose of offering more abstract services in a system. This composition creates interaction that promotes dependencies among components. System functionalities are not dependent solely on one component. Therefore, modifying a component may affect that composite functionality, which is reflected in different components. Similarly, replacing a new version of a specific component might involve replacing the component(s) on which it depends, in order to preserve a specific system's functionality. The key point to analyze such aspects is the knowledge about possible relations, interfaces and dependencies among them.

## 4. PROPOSED COMPLEXITY MEASURE

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

Size of each component: Size is also considered one of the parameter of program/class/component complexity. A class with more methods is harder to understand than a class with less number of methods and hence contributes more complexity [1,17]. Large programs/components 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 [1,17,20]. So, size is a factor which adds complexity to a component.

Interfaces of each component: In CBSD, a component is linked with other components and hence has interfaces with them. Two or more components are said to be interfaced if there is a link between them, where a link means that a component submits an event and other components receive it. The direction of the link indicates that which component requests the services or dependent on the other. Interface between two components can be through incoming and outgoing interactions. These both types of interactions add complexity to a component-based software system.

By taking only interface complexity into account, an interface complexity measure for a component-based system is suggested as :

Average Incoming Interactions Complexity (AIIC) =

$$\frac{\sum_{i=1}^{m} II_i}{m}$$

Average Outgoing Interactions Complexity (AOIC)

$$= \frac{\sum_{i=1}^{m} OI_i}{m}$$

Average Interface Complexity of a Component Based System

$$(\text{AIC (CBS)}) = \frac{\sum_{i=1}^{m} II_i}{m} + \frac{\sum_{i=1}^{m} OI_i}{m}$$ where

$m =$ Number of components in the Component Based System (CBS)

$II =$ Incoming Interactions

$OI =$ Outing Interactions

$\sum =$ Summation symbol,

$i =$ Index variable

## 5. THEORETICAL EVALUATION OF PROPOSED METRIC USING WEYUKER'S PROPERTIES

Weyuker [33] proposed an axiomatic framework in the form of several properties for evaluating complexity aspects of software systems. The proposed interface complexity metric reported here is evaluated against these properties for compatibility. The properties are:

Property 1: There are programs/components P and Q for which M(P) ≠ M(Q).
Property 2: If C is non-negative number, then there are finitely many programs/components P for which M(P)= C.
Property 3; There are distinct components/programs P and Q for which M(P)= M(Q).
Property 4: There are functionally equivalent components/programs P and Q for which M(P) ≠ M(Q).
Property 5: For any program/component bodies P and Q, we have M(P)<=M(P;Q) and M(Q)<=M(P;Q).
Property6: There exist program/component bodies P, Q and R such that M(P)=M(Q) and M(P;R) ≠ M(Q;R).
Property 7: There are program/component bodies P and Q such Q is formed by permuting the order of statements of P and M(P) ≠ M(Q).
Property 8: If p is renaming of Q, then M(P)=M(Q).
Property 9: There exist program/component bodies P and Q such that M(P)+M(Q)< M(P;Q).

These properties are evaluated for the proposed interface metric as described below:

i. There may be two different components with different complexities, thus satisfying the property 1.
ii. As each component will have at least one method with some functionality, therefore its complexity will always have some positive value. It validates the second property.
iii. For two different components with different functionality, the proposed complexity metric value may be same, as these methods may have same interface structure but with different functionality. It satisfies Property 3.
iv. Even if the functionality of the two components is same, both may have different complexities as these components may be designed by using different concepts of programming and technologies. It validates property number 4.
v. If a component is assembled with other component to get an integrated component for enhanced functionality, the complexity of these two individual components will be lesser than the complexity of the integrated component, which satisfies the 5th property.
vi. Two components with the same complexity means both will have same no. of interfaces. However, they may be developed by using different programming methodologies and therefore when integrating in the system, both may have different integration code and implementation thus resulting in different complexities of the system in both the cases. It confirms property number 6.
vii. If the ordering of interfaces in a component is changed, then it will not change the complexity of the new modified component. So, this property is not satisfied by our proposed interface metric.
viii. It is obvious that renaming an interface/method or a component will not affect the complexity of that interface or the component, thus satisfying the property no 8.
ix. When any two components are assembled, then we may have to write some more methods related with the integration in addition to the existing methods. This will increase the complexity of the assembled component. It validates the last property.

In this way, eight Weyuker properties are satisfied by the proposed interface metric.

# 6. CASE STUDY AND EXPERIMENTAL RESULTS

In order to compute the complexity of component-based systems through proposed measure, firstly directed graphs of design of components are developed and then proposed metric is applied. Five cases have been selected for the experimental study. In case 1, we have a component-based software system (CBSS) having four components as Figure 1. Here some components (A,B,D) having both directions interactions and component C has single direction interaction. In case 2, the directed graph consists of four components and four interactions, case 3 involves four components and six interactions, case 4 having six components and nine interactions and case 5 consists of six components and five interactions. It is clear that in case 2 and case 3, number of components are same ( four components ) but case 3 having more interactions than case 2. Similarly, case 4 and case 5 having same number of components ( six components ) but there are lesser number of interactions in case 5 than case 4.

**Case 1 : A CBSS having four components with both directions and single direction interactions**

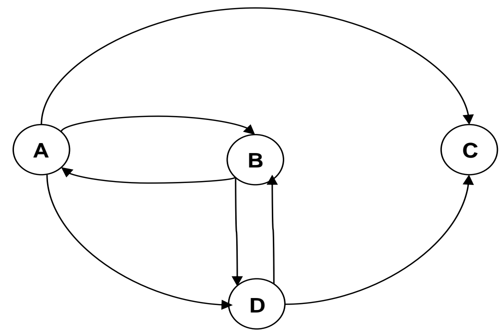


**Figure 1**

| | A | B | C | D | Total |
|---|---|---|---|---|---|
| II | 1 | 2 | 2 | 2 | 7 |
| OI | 3 | 2 | 0 | 1 | 6 |

$$AIIC = \frac{\sum_{i=1}^{m} II_i}{m} = 7/4 = 1.75$$

$$AOIC = \frac{\sum_{i=1}^{m} OI_i}{m} = 6/4 = 1.50$$

AIC(CBS) = 1.75 + 1.50 = 3.25

We suggest that AIC of a CBS should be less than equal to 5. It means that average number of interactions (interfaces) per component in a component based system (CBS) should not be greater than five, otherwise design complexity of that CBS would be highly complex and will be very complex to maintain, debug that system. As a result of this, that CBS system will be more prone to errors and hence unreliable. In figure 1, component B is more complex having total four interactions and so it is difficult to manage it than other components. However, this rule requires further empirical support.

**Case 2 : Four components with four interactions ( Case having less interactions between components )**

The directed graph of this case example is depicted in Figure 2.

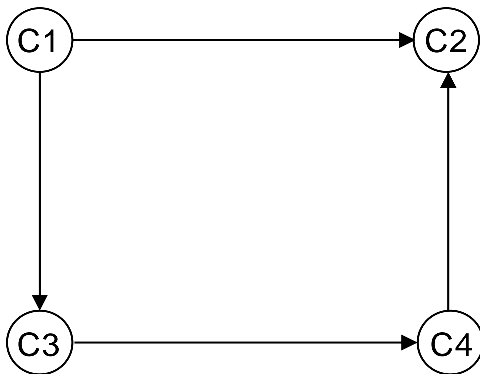


**Figure 2**

II(C1)=0, OI(C1)=2,
II(C2)=2, OI(C2)=0,
II(C3)=1, OI(C3)=1,
II(C4)=1, OI(C4)=1,
AIIC = 4/4=1, AOIC = 4/4=1
AIC = AIIC + AOIC = 1+1=2

**Case 3 : Four components with six interactions (Case having same number of components and more interactions between components than case 1)**

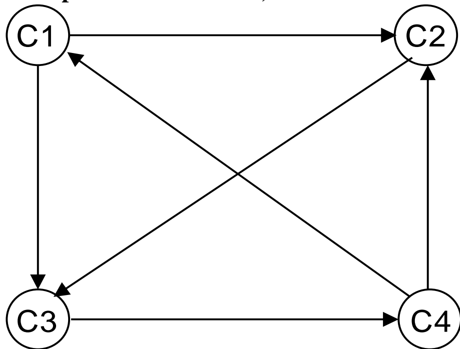


**Figure 3**

The directed graph for this case is depicted in Figure 3.

II(C1)=1, OI(C1)=2
II(C2)=2, OI(C2)=1
II(C3)=2, OI(C3)=1
II(C4)=1, OI(C4)=2
AIIC = 6/4=1.5, AOIC = 6/4=1.5
AIC = AIIC + AOIC = 1.5+1.5=3

| **For Case 2** | **For Case 3** |
|---|---|
| AIC(CBS)=2 | AIC(CBS) = 3 |

From these results, we infer that case 3 is more complex than case 2, though both cases having the same number of components. Results agree with reality as depicted by flow graphs of case 2 and case 3 and has brought quantitative affirmative of the same.

**Case 4 : Six components with nine interactions ( Case having more components and more interactions among components than case 1 and case 2 )**

The directed graph for this case is depicted in Figure 4.

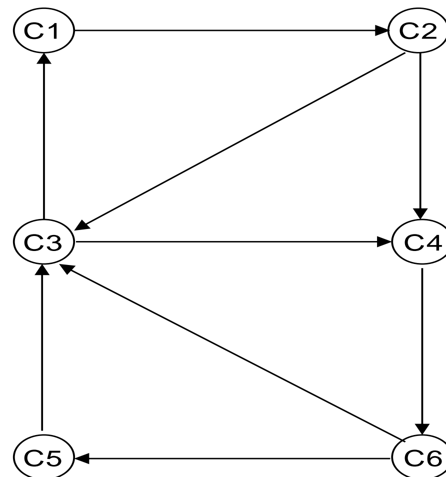


**Figure 4**

II(C1)=1, OI(C1)=1
II(C2)=1, OI(C2)=2
II(C3)=3, OI(C3)=2
II(C4)=2, OI(C4)=1
II(C5)=1, OI(C5)=1
II(C6)=1, OI(C6)=2
AIIC = 9/6=1.5, AOIC = 9/6=1.5
AIC = AIIC + AOIC = 1.5+1.5=3

**Case 5 : Six components with five interactions ( Case having same components and less interactions among components than case 4 )**

The directed graph for this case is depicted in Figure 5.

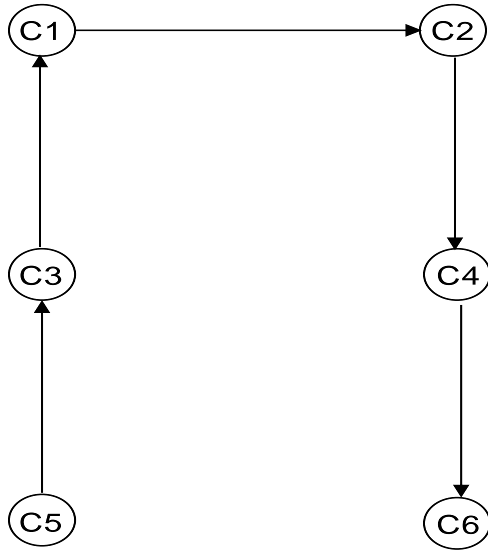


**Figure 5**

II(C1)=1, OI(C1)=1
II(C2)=1, OI(C2)=1
II(C3)=1, OI(C3)=1
II(C4)=1, OI(C4)=1
II(C5)=0, OI(C5)=1
II(C6)=1, OI(C6)=0
AIIC = 5/6=0.84, AOIC = 5/6=0.84
AIC = AIIC + AOIC = 0.84+0.84=1.68

On comparison of results of case 4 and case 5, we find that :

| **For Case 4** | **For Case 5** |
|---|---|
| AIC(CBS)=3.0 | AIC(CBS)=1.68 |

It is clear from the above results of case 4 and case 5 that case 4 is more complex than case 5 though both cases having the same number of components. The results agree with the complexities of flow-graphs of these cases 4 and 5 and has brought quantitative affirmative of the same.

# 7. CONCLUSIONS AND FUTURE WORK DIRECTIONS

In this paper, an interface complexity measure has been proposed which takes into account – interaction complexity, an important aspect of complexity of a component-based system. The results show that the effect of this parameter on complexity of a component-based system is quite significant. The results agree with the intuition that higher interaction between components increases the complexity because of more coupling among components. The same has also been theoretically evaluated against Weyuker's properties. Making early decisions about complexity of a component-based system may help a lot to software developers in reducing design, testing and maintenance efforts. The proposed measure appears to be logical and fits the intuitive understanding but is not the only criteria for deciding the overall complexity of a component-based system. As a thumb rule, we propose that average number of interactions (interfaces) per component in a component based system (CBS) should not be greater than five, otherwise that CBS would be highly complex and will be more prone to errors and hence unreliable. However, application of conclusions to real life situations needs further study and empirical support 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 proposed metric values and several quality attributes of component-based systems.

# 8. REFERENCES

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

[2] Mark, L, Jeff, K.(1994): Object Oriented Software Metrics, Prentice Hall Publishing.

[3] Basili, V.R., Biand, L., Melo, W.L. (1995): A validation of object-oriented design metrics as quality indicators, Technical report, Uni. of Maryland, Deptt. of computer science, MD, USA.

[4] Basili, V. (1980): Qualitative Software Complexity Models: A Summary, In Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos, CA.

[5] Singh, R., Grover, P.S. (1997): A New Program Weighted Complexity Metric, Proc. International conference on Software Engg. (CONSEG'97), Chennai, India, pp. 33-39.

[6] Brooks, I. (1993): 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.

[7] Harrison, W. (1982). Magel, K, Kluezny, R., dekock, A.: Applying Software Complexity Metrics to Program Maintenance, IEEE Computer, 15, pp. 65-79.

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

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

[10] Brooks, F.P. (1995): The Mythical Man Month: Essays on Software Engineering, Addison-Wesley.

[11] Zuse, H. (1991): Software Complexity Measures and Methods, W.de Gruyter, New York.

[12] Sellers, B. H. (1996): Object-Oriented Metrics : Measures of Complexity, Prentice Hall, New Jersey.

[13] Curtis, B. (1980): Measurement and Experimentation in Software Engineering, Proc. IEEE conference, 68,9, pp. 1144-1157.

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

[15] Usha Chhillar, Sucheta Bhasin (2011): A New Weighted Composite Complexity Measure for Object-Oriented Systems, International Journal of Information and Communication Technology Research, 1 (3).

[16] Elish, M.O., Rine, D. (2005): 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. (1977): Elements of Software Science, New York: Elsevier North Holland.

[18] McCabe, T.J. (1976): A Complexity Measure, IEEE Trans. On Software Engg., SE-2, 4, pp. 308-320.

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

[20] Usha Kumari, Sucheta Bhasin (2011): Application of Object-Oriented Metrics To C++ and Java : A Comparative Study, ACM SIGSOFT Software Engineering Notes, 36 (2), pp. 1-6.

[21] Capretz, L.F (2005): Y : A New Component-Based Software Life Cycle Model, Journal of Computer Science, 1 (1), pp. 76-82.

[22] Brown, Wallnau (1998): The Current State of CBSE, IEEE Software, 15 (5).

[23] Gill, N.S, Balkishan (2008): Dependency and Interaction Oriented Complexity Metrics of Component-Based Systems, ACM SIGSOFT Software Engineering Notes, 33 (2), pp. 1-5.

[24] Ravichandran, T., Rothenberger, M. (2003): Software Reuse Strategies and Component Markets, Communications of the ACM, 18 (5), pp. 410-422.

[25] Dogru, A.H., Tanik, M. (2003): A process Model for Component Oriented Software Engineering, IEEE Software, pp. 34-41.

[26] Vitharana, P., Zahedi, F.M., Jain, H. (2003): Design Retrieval and Assembly in Component-Based Software Development, Communications of the ACM, 46 (11), pp. 97-102.

[27] Basili, V.R., Boehm, B. (2001): COTS-Based Systems Top 10 List, IEEE Computer, 34 (5), pp.91-93.

[28] Noel SALMAN (2006): Complexity Metrics As Predictors of Maintainability and Integrability of Software Components, Journal of Arts and Sciences, 5, pp. 39-50.

[29] Jianguo Chen et.al (2011): Complexity Metrics for Component-based Software Systems, International Journal of Digital Content Technology and its Applications, 5 (3), pp. 235-244.

[30] Sengupta, S., Kanjilal, A. (2011): Measuring Complexity of Component Based Architecture : A Graph Based Approach, ACM SIGSOFT Software Engineering Notes, 36 (1), pp. 1-10.

[31] Sharma, A., Grover, P.S., Kumar, R. (2009): Dependency Analysis for Component-Based Software Systems, ACM SIGSOFT Software Engineering Notes, 34 (4), pp. 1-6.

[32] IEEE Standard Glossary of Software Engineering Technology, 1990, IEEE Std. 610.12-1990, The Institute of Electrical ad Electronics Engineers, Inc.

[33] Weyuker, E.J. (1988): Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering, 14 (9), pp.1357-1365.

[34] Usha Chhillar, Sucheta Bhasin (2011): A Journey of Software Metrics : Traditional to Aspect-Oriented Paradigm, 5th National Conference on Computing For Nation Development, 10th -11th March, 2011, New Delhi, pp. 289-293.

[35] Usha Chhillar, Sucheta Bhasin (2011): A Composite Complexity Measure For Component-Based Systems, ACM SIGSOFT Software Engineering Notes, 36 (6), pp. 1-5.