Journal of Theoretical and Applied Information Technology
20th March 2015. Vol.73 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

EMPIRICAL EVALUATION OF COMPLEXITY METRICS FOR COMPONENT BASED SYSTEMS

[1]MWALILI TOBIAS, [2]WAWERU MWANGI, [3]KIMWELE MICHAEL

[1]School of Computing and Information Technology
Jomo Kenyatta University of Agriculture and Technology
P. O. Box 62000- 00200 Nairobi, Kenya

[2]School of Computing and Information Technology
Jomo Kenyatta University of Agriculture and Technology
P. O. Box 62000- 00200 Nairobi, Kenya

[3]School of Computing and Information Technology
Jomo Kenyatta University of Agriculture and Technology
P. O. Box 62000- 00200 Nairobi, Kenya

E-mail: [1]tmwalili@jkuat.ac.ke, [2]kimwele@icsit.jkuat.ac.ke, [3]waweru_mwangi@icsit.jkuat.ac.ke,

ABSTRACT

Reuse-based software engineering is gaining currency as an approach for constructing software applications that are based on existing software components. Factors that have contributed to increased reliance on software components include increased dependability, reduced process risk, standards compliance and reduced time to market. Software components are usually delivered and handled as "black boxes," which tremendously increases risks associated component integration, system testing and deployment. Due to these risks, metrics for evaluating the quality of component-based systems must be developed and validated. In this work, we analyze the Interface Complexity Metric for JavaBeans components and propose an enhanced metric. We also perform validation of the proposed metric and make recommendations for future research work.

Keywords: *CBSD, Component complexity, Complexity metrics, Software complexity, Quality metrics*

1. INTRODUCTION

The earliest approach to accelerating software delivery relied on function reuse. With the paradigm shift towards object-oriented development, object-based reuse became the preferred way of achieving the objective. Over time, object reuse has failed to provide the required level of abstraction to model and construct complex systems, within budget and time constraints. Due to these limitations, Component-Based Software Engineering (CBSE) or Component-Based Software Development (CBSD) has emerged [1]. According to Sommerville [2], the CBSE is a process that defines implements and integrates components into a system. It involves the use of already existing software components to assemble a system, without building from scratch [3].

A software component is a unit of composition [4] with a clearly defined interface. It can be deployed and composed independently by third party developers. A software component can also be described as an independent service provider which has two interfaces, a "provides-interface" that specifies the services provided by the component and a "requires-interfaces" that specifies what services must be provided by other components in the system [2].

The CBSD approach has potential advantages over object-base reuse, namely; reduced development time, increased flexibility, reduced process risks, and enhanced quality, low maintenance costs and standardization. Despite these promises, the CDSD approach is faced by numerous challenges, which include user requirements satisfaction, components interoperability, component trustworthiness and

Journal of Theoretical and Applied Information Technology
20th March 2015. Vol.73 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

inability to predict the quality of the constructed system.

The above challenges underline the ever growing need for techniques that could improve the process of component selection and evaluation, by introducing efficient tools for estimating and predicting the quality of target components.

The objectives of this work include:-

I. Review on metrics for component-based systems (CBS), and identify existing gaps or limitations.

II. Propose new or enhanced metrics for CBS, based on identified gaps or limitations.

III. Perform an empirical evaluation of the proposed metrics.

IV. Make recommendations for further research work.

2. LITERATURE REVIEW

A component model defines a standard for implementing, documenting and deploying components based on a particular technology. Over time different technologies for component models have emerged. They include, Sun's JavaBeans, OMG's CORBA Component Model (CCM), Microsoft's .NET and OSGI Open Service Gateway Initiative (OSGI). Since the JavaBeans and .NET models are the most widely used, we will give a brief description of their architecture.

2.1 JavaBeans Component Model

The JavaBeans component model is a Sun technology, for integrating components developed using the Java language. According to Ivica [5] JavaBeans Application Programming Interface defines a software component model for Java, this allows developers to create and deploy components that can be assembled into applications by users. The interface for this model is defined by methods, properties, event sources, and event listeners as depicted in Figure 1.

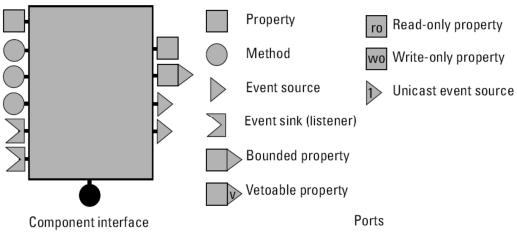

Figure 1: Interface of a JavaBean component and its ports. (Source: Ivica, [5])

The Java Bean is designed to run inside a builder tool (Composition time) and also at run-time (execution time) within the generated application. A simple Java object can be used to implement a component with the object being encapsulated in the component, where the mapping between object methods and component is done in an implicit version as long as the object and the component adhere to the standard java naming convention. In other cases, a component could be implemented by wrapping a legacy object that does not follow the standard naming convention.

The Java Bean component model is designed to support different ways of assembling components, such that builder tools can allow visual direct plugging together of Java Bean while users write Java classes or simple scripting language that interact with and control a set of beans. The model also provides a set of methods for packaging components as archives for deployment [5].

2.2 The .NET Component Model

The .NET is a Microsoft technology, first released in July of 2000 and billed as a whole new development framework for windows. The .NET technology serves as a foundation for all Microsoft technologies. The .NET is basically a class library with tools needed to write applications based on various programming languages which include C#, VB, C++, Jscript, etc. Fig. 2 shows the architecture of the .Net framework.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

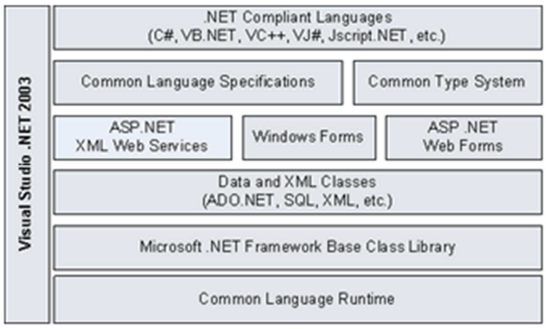

Figure 3: The .NET Framework Architecture (Source: Microsoft TechNet [6])

2.3 Software Metrics: An Overview

According to IEEE [7], a metric is a measurable quantity, the degree to which a system or component possesses a given property. When discussing metrics for software components, we will confine ourselves to attributes that can be measured and evaluated.

Perhaps the earliest known software metric is size oriented Kilo Lines of Code (KLOC), which has been used as an input to derive measurements such as effort, error rate and documentation. Application of KLOC is straight forward where LOC are an available and can be easily counted. Metrics derived from KLOC are biased in some aspects since LOC measures are programming language dependent. Also, for reuse-based approach source codes are precompiled and may be completely unavailable.

Albrecht[8] proposed Function-oriented metrics, based on a measure called the function point (FP). Function points are calculated using countable aspects of the software as assessments for software complexity. The function points so derived then can be used to compute metrics for software, for example, productivity, quality, documentation, etc.

Widely referenced software metric is the cyclomatic complexity proposed by McCabe [9]. It uses graph theory to measure software complexity. It looks at the program's control flow graph and determines the minimum number of paths in that graph. McCabe argued that this number determines the complexity (cyclomatic complexity) of the program.

Halstead [10] devised a metric, based on two quantities: the number of distinct operators in the program and the number of distinct operands in the program. From these numbers, one can construct the "Halted Length" which is the measure of the complexity of the program. Usually the "Halted length" is calculated after the code is written but is also used for the measurement of programming effort.

Chidamber and Kemerer [11] proposed a suite of six object-oriented metrics. These metrics provided a paradigm shift towards object orientation in the development of software metrics and have had a major influence in the construction of metrics for CBSD.

Sedigh [12] proposed three categories for CBS metrics. They include management, requirements and quality-based metrics. These metrics are broad recommendations and suffer from lack of formalism and therefore not easy to validate. To provide a firm ground for formalization Washizaki's [13] proposed the several metrics for measuring reusability of software components, which include:-

(a) Rate of Component Observability (RCO) given by

$$RCO(c) = \frac{No\ of\ readable\ propoerties\ in\ class\ C}{No\ of\ fields\ in\ C's\ facade\ class} \quad \ldots. (1)$$

A very low RCO value indicates a component that is difficult to understand while a very high RCO value means users will have difficulties in finding specific properties among the available ones

(b) Rate of component customizability (RCC)

$$RCC(c) = \frac{No.of\ writable\ propoerties\ in\ class\ C}{No.of\ fields\ in\ C's\ facade\ class} \quad \ldots. (2)$$

A low RCC value implies poor adaptability of the component, while a very high one indicates a break in the encapsulation of the component.

(c) Self-Completeness of Component's Return Value (SCCr)

$$SCCr(c) = \frac{No.of\ void\ methods\ in\ class\ C}{No.of\ \ Methods\ in\ class\ C} \quad \ldots.. (3)$$

It is a degree of the component's self-completeness and independence. The higher the value is, the higher the component portability.

Self-Completeness of Component's Parameter (SCCp)

$$SCCp(c) = \frac{No\ of\ methods\ with\ parameters\ \ in\ class\ C}{No.of\ Methods\ in\ class\ C}$$

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

.....(4)

This metric measures the self-completeness of the information dealt by the component. A low value indicates a low dependency of the component on the exterior.

Miguel [14] implemented formal specifications for the Washizaki's metrics. Working within the framework of UML 2.0 they applied Object Constrained Language (OCL) to automatically compute metrics from fine-grained Java Beans components. Sharma[15] proposed the Interface Complexity Metric (ICM) that based on complexity factors derived from components interface methods and properties; our work study focuses on the ICM whose details are discussed in the next section. Other recent research initiatives on metrics for CBSD could be attributed to Navneet [17], this research work performed a survey of existing metrics for CBSD.

The outcome of this survey indicated the need for development of complexity metric that can measure the component complexity without going into internal details of components. As a continuation of the previously mentioned survey, Navneet [18] highlighted the shortcomings of component existing metrics especially the fact that most of the existing metrics can only be used to asses small programs or components, while others rely on parameters that are difficult to measure in practice. They proposed a new metric called, The Components Complexity Metric for Black Box Components CCM (BB), based on interface methods complexity and coupling complexity between the components. However, they did not perform empirical or theoretical validation for the proposed metric.

2.4 The Interface Complexity Metric

Sharma [17] proposed the Interface Complexity Metric (ICM). This section gives a brief description the ICM and points some limitations against which we make a proposal for an improved ICM metric.

The ICM models the external behavior of the component as aggregation components methods and properties complexity factors given by Equation (5)

$$ICM(C) = A\sum_{i=1}^{m} CIM_i + B\sum_{j=1}^{n} CP_j$$

….(5)

Where, CIM_i is the complexity of the i^{th} interface method and CP_j is the complexity of the j^{th} property. A and B are the weight values for methods and properties respectively. For their study the fixed A=b=1 and as such, the complexity metric reduced to Equation (6)

$$ICM(C) = \sum_{i=1}^{m} CIM_i + \sum_{j=1}^{n} CIP_j$$

…..(6)

The complexity of an interface method is computed based weighed values that are assigned to each return values or argument according to its data type as summarized in Table 1 below.

Table 1: Assigned complexity weights (Source: Sharma, [17] pp 28)

	Assigned weight for each type of argument/ Return value			
No of Args	Simple (Int, double)	Medium (Date, String)	Complex (Vector, Array)	Highly Complex (Objects references, User defined)
1-3	0.10	0.15	0.20	0.25
4-6	0.20	0.30	0.40	0.50
7-9	0.30	0.45	0.60	0.75
>=10	0.40	0.60	0.80	0.10

To validate the proposed metric they performed an empirical analysis based for JavaBeans components collected from websites (JarsD.com, ElegantJBeans.com, and Oreilly.com). For each of the JavaBeans component they computed; Component execution time (default values of parameters) and Components Interface Complexity. They also performed a correlation analysis Washizaki's metrics for customizability and readability of a component. The results indicate a strong correlation between complexity and execution time, negative correlation between complexity and customizability and negative correlation between complexity and readability.

2.5 Limitations of the ICM

A Scatter plot analysis of the data set provided by Sharma [17] shows that there is a positive positive linear relationship between the ICM the size

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

(Number of methods + Properties) of its interface class (Figure 3). Further, correlation analysis results provided in Table 3 shows a very strong positive correlation between the complexity and number of methods and properties.

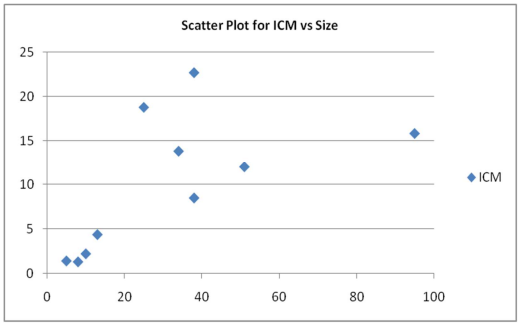

Figure 3: Scatter Plot For ICM Against Size

Table 3: Correlation Analysis For ICM Against Component Size

Characteristic	Correlation
ICM VS No of Methods	0.8398
ICM VS No of Properties	0.4925
ICM VS Size	0.6051

An interpretation of these results indicates that complexity of a component will increase with its size. Based on this, we could argue that the ICM and size are equivalent since they provide the same information. This fact is very significant, given that the functionality of a particular component is accessed via the interface. The ICM will, therefore, punish (give a low rating) to an elaborate component that provides broad spectrum of functionalities to the user and give credit to a component that has limited functionalities. We also note that the data set used by Sharma [17] to validate ICM is limited in size. For these reasons, we are proposing an enhancement to the ICM as well as an in-depth empirical analysis for the proposed metric.

3. PROPOSED INTERFACE COMPLEXITY METRIC

The previous section highlighted some limitations associated with the ICM. Our concern here is the fact that the ICM grows with the size of the component interface. Based on the ICM, suppose a given component of size M is determined to have complexity factor C, later suppose the developer provides new functionalities by adding new methods and properties, thereby increasing components self-completeness. If D is the resultant new complexity factor, it follows that the relation D>C will always be true. This means that, due to "increased" complexity the new improved component will be rated low; while true sense it is now much more self-contained than the previous one.

We, therefore, propose a Bounded ICM (BICM), it's bounded in such a way that it may not necessarily grow with the size, as shown in Equation (7)

$$BICM(C) = A\frac{\sum_{i=1}^{m} CIM_i}{M} + B\frac{\sum_{j=1}^{n} CIP_j}{N} \quad \ldots\ldots\ldots (7)$$

Where, CIM_i is the complexity of the i^{th} interface method and CP_j is the complexity of the j^{th} property. M and N represents the count of component methods and properties respectively while A and B are the weight values. The proposed metric may guarantee that the complexity does not grow with size, and can ne bounded to a definite interval [a,b], for example [0,1].

4. EMPIRICAL ANALYSIS OF THE BICM METRIC

To perform the analysis, we downloaded 36 sample JavaBeans components from the components super store, ComponentSource.com. The analysis was carried out in a series of steps as discussed below.

4.1 Extraction of façade class interface information

Class reflection technology was used generate the components façade class methods interface, and properties. For each method in the façade class we extracted the methods return type, and a list of method argument types. We also captured the property data-type for all properties in the class. Tables 4 and 5 shows sample data summarized from a components façade class.

Table 4: Sample methods data derived from a components façade class

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

#	Method Name	Return Type	Arguments inMethod						#Args
1	fireSSLServerAuthentication	void	byte[]	String	String	String	boolean[]		5
2	fireSetCookie	void	String	String	String	String	String	boolean	6
3	fireStatus	void	String	int	String		.	.	3
4	fireTransfer	void	int	long	int	byte[]	.	.	4
5	class$	Class	String	.	.	.	.	.	1
6	getAbout	String	.	.	.	.	.	.	0
7	getCookies	HTTPCookieList	.	.	.	.	.	.	0
8	getFirewall	Firewall	.	.	.	.	.	.	0
.	....	....	....	....	....	....	....	....	.....
.	....	....	....	....	....	....	....	....	....
.	getProxy	Proxy	....	....	....	....	....	....	0
54	getSSLCert	Certificate	....	....	....	....	....	....	0
55	isIdle	boolean	....	....	....	....	....	....	0
	Total Number Of Methods	**55**							

Table 5: Sample properties data derived from a components façade class

Properties Number	Property Type
1	String
2	bn
3	GmailEventListener
4	String
5	boolean
6	Class
Total Number Of Properties	**6**

4.2 Computation of components metrics

Using the data summarized in the previous step, we computed various parameters and metrics for all the sampled components. The ICM and BICM metrics were computed using Equations 6 and 7 respectively. The weights in Table 1 were used for assigning complexity factors for methods arguments and class properties. The Washizaki's metrics, SCCR and SCCP were computed using Equations 3 and 4 in that order. The results of these computations are presented in Table 6 below, where table headings **M** and **P** represents components façade class methods and properties count respectively and **Size** the sum of methods and properties. The rest of the headings are as discussed previously.

Table 6: Tabulation of components Size against ICM, BICM, Washizaki's SCCR and SCCP

S#	Component	M	P	Size	ICM	BICM	SCCR	SCCP
1	Gmail	55	6	61	22.25	0.58	0.5091	0.5273
2	Gcalender	120	9	129	38.45	0.47	0.5167	0.4500
3	Gdata	87	6	93	34.35	0.57	0.5287	0.4828
4	Gdocuments	102	6	108	36.35	0.54	0.5490	0.5098
5	Gcontacts	111	6	117	31.95	0.47	0.0631	0.4775
6	Gspreadsheets	115	6	121	46.75	0.59	0.5217	0.4435
7	Gstorage	102	16	118	36.15	0.46	0.6176	0.5686
8	Paapi	72	30	102	23.45	0.39	0.4722	0.5000
9	Amazonrequest	100	9	109	34.10	0.49	0.5600	0.5500
10	S3	104	13	117	36.60	0.48	0.6154	0.5769
11	Sqs	68	6	74	26.50	0.56	0.5522	0.6269
12	Ec2	63	6	69	37.00	0.76	0.6508	0.6349
13	Simpledb	57	9	66	17.85	0.45	0.5965	0.5439
14	Blob	110	12	122	41.35	0.51	0.6636	0.5727
15	Table	82	9	91	25.15	0.45	0.5854	0.4878
16	Queue	76	9	85	25.60	0.48	0.6316	0.5000
17	Azurerequest	97	11	108	30.30	0.47	0.5567	0.5567
18	Sapclient	53	8	61	12.95	0.39	0.5849	0.4528
19	Ftp	70	9	79	17.95	0.40	0.6429	0.5143
20	Rss	80	20	100	34.10	0.52	0.6125	0.6125
21	Smtp	90	22	112	22.70	0.35	0.6000	0.5333
22	Soap	104	19	123	29.90	0.39	0.4904	0.4712
23	Mime	31	6	37	10.15	0.53	0.6452	0.5484
24	Pop	71	13	84	19.05	0.47	0.5211	0.4225
25	Syslog	21	6	27	12.40	0.73	0.6190	0.6667
26	Telnet	60	6	66	17.00	0.46	0.7500	0.7333
27	Whois	28	6	34	8.10	0.44	0.6071	0.6071
28	Webupload	71	16	87	24.00	0.44	0.5915	0.5634
29	Webdav	93	20	113	34.35	0.47	0.6344	0.6237
30	Xmpp	92	16	108	45.60	0.61	0.6739	0.6304
31	atePicker	155	57	212	54.95	0.49	0.5419	0.5484
32	PVTree	192	54	246	72.85	0.51	0.5313	0.6354
33	PVTable	247	102	349	97.45	0.48	0.5182	0.5992
34	PVCalculator	77	29	106	22.30	0.38	0.6364	0.5844
35	PVCalendar	209	87	296	110.35	0.62	0.5598	0.5837
36	PVChoice	188	57	245	54.10	0.40	0.5484	0.5806

4.3 Correlation Analysis for ICM and BICM

The scatter plot in Figure 4, which is generated using data in table 6, indicates that there exists a linear relationship between size and ICM. Correlation coefficient for ICM against Size is 0.9346 (Table 7). We note that this factor is positive and close to 1.0, this outcome, confirms that indeed ICM increases with the size of component. As argued in the previous section, the ICM will, therefore, punish (give a low rating) to an elaborate component that provides expanded functionalities to the user and give credit to a component that lacks essential functionalities.

Journal of Theoretical and Applied Information Technology
20th March 2015. Vol.73 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

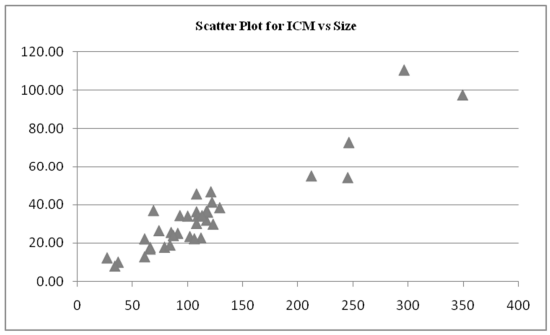

Figure 4: Scatter plot for ICM against size

To see how BICM behaves, we generated the scatter plot for BICM against size (Figure 5). The scatter plot, analyzed together with the weak negative correlation of -0.0575 (Table 7) indicates that BICM does not grow with the size of a component. This is a desirable outcome for this experiment since the BICM may be deemed to have eliminated the limitations of ICM.

.

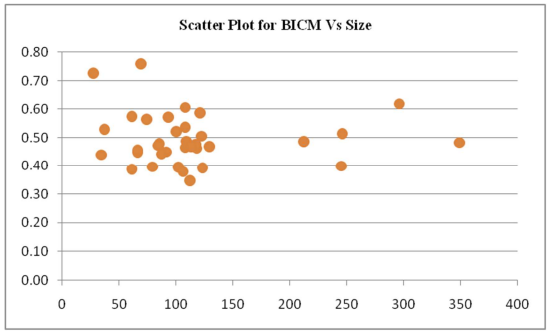

Figure 5: Scatter plot for BICM against size

In order to validate the BICM against other metrics in literature, we computed the correlation between BICM and Washizaki's metrics, Self-Completeness of Component's Return Value (SCCr) and Self-Completeness of Component's Parameter (SCCp) The results, (Table 7) show a positive correlation between the BICM and SCCP. This implies that BICM may be used to evaluate component quality characteristics such self completeness, independence and portability

Table 7: Correlation analysis for components Size against ICM, BICM and Washizaki's SCCR and SCCP

Characteristic	Coefficient
ICM VS Size	0.9346
BICM VS Size	-0.0575
BICM VS SCCR	0.0940
BICM VS SCCP	0.3275

5. CONCLUSION AND RECOMMENDATION FOR FURTHER WORK

Component-based software engineering approach promises delivery software products within constrained time and budget. Metrics for CBS have been introduced and developed. We have presented an empirical analysis of existing and proposed metrics for JavaBeans components. In particular we have carried out in-depth analysis for the Interface complexity metric (ICM) and showed that it fully depends on the number of interface method and properties (size). A component quality evaluation based on the ICM would, therefore, be biased against components that provide increased services by via added methods.

We also suggested an improvement to the ICM and proposed a new metric BICM. The analysis of the BICM reveals that it is independent of interface size. We also validated the BICM against existing metrics and demonstrated that the BICM can be applicable in evaluating components self-completeness, independence and portability.

We however note that there some aspects of complexity this study did not address, for example, the BICM defined in Equation (1), has the customization constants A and B. in our study these constants were assumed to equal to 1(one), further work is, therefore, needed to study how the BICM when the component customization constants are loaded. We performed an empirical analysis for each component as a stand-alone. There is therefore need to investigate how the BICM will behave at system level, that is when components are composed into a system and its overall system-BICM computed.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REFERENCES:

[1] Mahmood, S. Lai R. Kim Y.S. (2007), Survey of component based software development, IET Software, 1 (2), pp 57-66

[2] Sommerville, I. (2007). Software Engineering, 8th Edition. Pearson Education Limited, pp 440-450

[3] Kaur, K. and Singh H. (2010). Candidate Process Models for Component Based Software Development. Journal Of Software Engineering 4(1): 16-29

[4] Szyperski, C., Gruntz, D, and Murer, S.(2002). Component software –beyond object –oriented programming 2nd Edition. Addison- Wesley pp 27-38

[5] Ivica C and Magnus L,(2002) "Building Reliable Component-Based Software Systems", Archtech House Inc, pp 57-70

[6] Microsoft TechNet: Introduction to NET, http://technet.microsoft.com/en-us/library/bb496996.aspx,accesed 19th July 2014

[7] IEEE, (1993), IEEE Software Engineering Standards, Standard 610.12-1990, pp. 47–48

[8] Albrecht, A.J.(1979), "Measuring Application Development Productivity," Proc. IBM Application Development Symposium, Monterey, CA, October 1979, pp. 83–92.

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

[10] Halstead, M., (1977) Elements of Software Science, North-Holland

[11]] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. IEEE Transaction on Software Engineering, 20(6), 476-493.

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

[13] Hironori Washizaki, Hirokazu Yamamoto & Yoshiaki, Fukazawa.(2003) A Metrics Suite for Measuring Reusability of Software Components. In 9th IEEE International Software Metrics Symposium (METRICS 2003), Sydney, Australia. IEEE Computer Society

[14] Miguel G, Fernando B(2004), Formalizing metrics for COTS ("Proceedings of the International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC'04)" , EEI

[15]] Sharma Arun (2009), Design and Analysis of Metrics for Component Based Software Systems, Phd Thesis, Thapar University (Punjab), India

[16] Weyuker, E.J., 1988. Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering, Vol. 14, Issue 9, pp: 1357-1365

[17] Navneet Kaur and Ashima Singh (2013a), A Complexity Metric for Black Box ComponentsInternational Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-3, Issue-2, May 2013

[18] Navneet Kaur and Ashima Singh (2013b) Component Complexity Metrics : A Survey, International Journal of Advanced Research inComputer Science and Software Engineering, Volume 3, Issue 6, June 2013