*Sādhanā* Vol. 36, Part 3, June 2011, pp. 317–337. © Indian Academy of Sciences

# An inheritance complexity metric for object-oriented code: A cognitive approach

SANJAY MISRA, IBRAHIM AKMAN and MURAT KOYUNCU*

Department of Computer Engineering, Atilim University, 06836, Ankara, Turkey
e-mail: smisra@atilim.edu.tr; akman@atilim.edu.tr; mkoyuncu@atilim.edu.tr

MS received 25 April 2010; revised 6 October 2010; accepted 5 December 2010

**Abstract.** Software metrics should be used in order to improve the productivity and quality of software, because they provide critical information about reliability and maintainability of the system. In this paper, we propose a cognitive complexity metric for evaluating design of object-oriented (OO) code. The proposed metric is based on an important feature of the OO systems: Inheritance. It calculates the complexity at method level considering internal structure of methods, and also considers inheritance to calculate the complexity of class hierarchies. The proposed metric is validated both theoretically and empirically. For theoretical validation, principles of measurement theory are applied since the measurement theory has been proposed and extensively used in the literature as a means to evaluate the software engineering metrics. We applied our metric on a real project for empirical validation and compared it with Chidamber and Kemerer (CK) metrics suite. The theoretical, practical and empirical validations and the comparative study prove the robustness of the measure.

**Keywords.** Software metrics; object-oriented programming; software complexity; cognitive weights; measurement theory; empirical validation.

## 1. Introduction

Object-Oriented (OO) techniques have started to dominate software engineering over the last two decades. One of the reasons for this situation is the maintainability of the OO software. In order to evaluate the maintainability of OO software, the quality of their design must also be evaluated using adequate quantification means (Marinescu 2005). Because once the design has been implemented it is difficult and expensive to change. This means that the design should be good from starting of the software development (Reißing 2001) and software metrics are the tools to evaluate the quality of the design. Today, the literature provides a variety of metrics to compute the complexity of software from various perspectives. Amongst them, we can mention the Chidamber & Kemerer (CK) metrics suite (1994), MOOD metrics for OO Design (Harrison *et al* 1998), design metrics for testing (Binder 1994), product metrics for object-oriented design (Purao & Vaishnavi 2003; Vaishnavi *et al* 2007), Lorenz and Kidd metrics (Lorenz & Kidd 1994),

*For correspondence

Henderson–Sellers metrics (Henderson–Sellers 1996), (slightly) modified CK metrics (Basily *et al* 1996), size estimation of OO systems (Costagliola *et al* 2005), weighted class complexity metric (Misra & Akman 2008a), metrics for XML documents (Basci & Misra 2009a), metrics for web services (Basci & Misra 2009b) and package coupling measurement (Gupta & Chhabra 2009). A summary of various metrics and closely related literature for OO code can also be found in (Babsiya & Davis 2002; Briand & Wust 2001; Kim *et al* 1995; Kim & Lerch 1991; Olague *et al* 2007; Stephen 2003). All these complexity measures are the indication of some quality attributes and they have their own advantages and disadvantages. Furthermore, introducing new complexity measures or improving existing ones is always there to achieve higher quality software evaluated by more effective measures.

The popularity of the OO software development is due to its powerful features like encapsulation, objects composition, inheritance, interaction, polymorphism, dynamic binding and reusability. Further, OO approach is characterized by its classes and objects, which are defined in terms of attributes (data) and operations (methods). These elements are defined in class declarations. Among these, the method plays an important role since it operates on data in response to a message. Although complexity of methods directly affects understandability of the software, complexity metrics based on the method have not yet been studied carefully. There are very few metrics in the literature for measuring the complexity of operations in the method. Most of these metrics do not consider the internal architecture of the method as well as special features of OO design.

One way to calculate the complexity of the method is through traditional metrics. However, the applicability of these metrics is under several criticisms in OO code (Chidamber & Kemerer 1994; Wand & Weber 1990; Weyuker 1988; Wilde & Huitt 1992). These criticisms are mainly based on lack of theoretical basis, lack of desirable measurement and mathematical properties, being insufficiently generalized or too implementation technology dependent. In our opinion, one of the most important criticisms should be the lack of features for representing the main characteristics of OO approaches. These metrics only calculate the complexity of operations in the method, which is similar to the complexity calculation for procedural language programs, and therefore do not capture the features of OO system. It was also the case in one of our previous works (Misra &Akman 2008a). This seems to be the main reason for the failure of the conventional metrics used on the method level for complexity measure of OO code.

In addition, the available OO metrics do not consider the cognitive characteristics (i.e., the cognitive complexity) in calculating the code complexity. The cognitive complexity is defined as the mental burden on the user who deals with the code, for example the developer, tester and maintenance staff. The cognitive complexity can be calculated in terms of cognitive weights. Cognitive weights (Wang & Shao 2003) are defined as the extent of difficulty or relative time and effort required for comprehending given software, and measure the complexity of logical structure of software. A higher weight indicates a higher level of effort required to understand the software. A high cognitive complexity is undesirable due to several reasons, such as increased fault-proneness and reduced maintainability. Additionally, the cognitive complexity also provides valuable information for the design of OO systems. High cognitive complexity indicates poor design, which sometimes can be unmanageable (Briand *et al* 2001). In such cases, maintenance effort increases drastically.

This work proposes a new metric for evaluating the design of OO code to eliminate the drawbacks given above. The proposed metric includes the cognitive complexity of operations in a method in terms of cognitive weights. It also considers the inheritance property to be an important feature of the OO systems. To the best of our knowledge, none of the available object-oriented metrics calculate the total complexity of the code by considering the complexity due

to the internal architecture of the code except our earlier work, which calculates the complexity of class by considering attributes and methods (Misra & Akman 2008b). On the other hand, these metrics ((Misra & Akman 2008a, b) failed to consider the inheritance properties of OO programs and cognitive aspects together. In addition, none of them are empirically validated, and without empirical validation, the practical usefulness of a new metric can not be proved. All these issues are needed for the quantification of the ease of maintainability since they are closely related to the design of the system and play an extremely important role in the software development process. Therefore, all these issues are considered in this proposal. The preliminary work of this study was introduced in RSKT 2008 (Misra & Akman 2008c). In this paper, we extended our previous work, and evaluated and validated our metric through practical, theoretical and empirical validations. In addition, a comparative study with similar measures is given.

The next section presents the proposal of the new complexity metric. The theoretical validation of the proposed metric through measurement theory is given in section 3. Section 4 provides the results of a case study, empirical validation and comparative study. The pros and cons with future work are discussed in section 5, and, finally, a conclusion is given in section 6.

## 2. The proposed metric for object-oriented programming

An object is a class instance and an OO system should be treated as a number of objects which collaborate through message exchanges. An OO code consists of one or more classes which may be related to each other by composition or by inheritance and contains related attributes and operations (methods) in the classes. The complexity metrics developed for OO languages are mainly based on the complexity of individual classes like number of methods, number of messages, etc. However, the complexity of the entire code is also important, and for calculating the complexity of the entire system, we have not only to find the complexity for each component of the system but also to consider the type of the relations between them.

The proposed metric is first interested in calculating the complexity of methods considering corresponding cognitive weights for each method of the class of the system (Eq 1). Cognitive weights are used to measure the complexity of the logical structures of the software in terms of Basic Control Structures (BCSs). These logical structures reside in the method (code) and are classified as sequence, branch, iteration and message call with the corresponding weights of one, two, three and two, respectively. Actually, these weights are assigned on the classification of cognitive phenomenon as discussed by Wang & Shao (2003). They proved and assigned the weights for subconscious function, meta cognitive function and higher cognitive function as 1, 2 and 3, respectively.

The complexity due to method calls is also considered at this stage. If there is a message call to one of the methods of other classes, the complexity of that message in the method is the sum of the weights of the called method and the weight due to that call. On the other hand, if the message call is for a method in the same class, we only assign the weight due to the call. More formally, the method complexity (MC) is calculated as

$$MC = \sum_{j=1}^{q} \left[ \prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right], \tag{1}$$

where $W_c$ is the cognitive weight of the concerned basic control structure (BCS). The method complexity of a software component is defined as the sum of cognitive weights of its q linear

blocks composed of individual BCSs, since each block may consist of m layers of nested BCSs, and each layer with n linear BCSs.

Some methods in an object-oriented code may include recursive method calls. Each recursive method call is considered as a new call and taken into account during the calculation of method complexity. If the recursively called method is inside the same class of method which initiates the first call, then we add only the complexity arisen because of method calls, not the complexity of called method. If the recursively called method is from another class, we include the method complexity only once. Because, the cognitive complexity burden to developers/programmers by recursive method is not repetitive.

The second stage (Eq 2) of the proposed metric calculates the complexity of each class. Equation 1 gives the complexity of the single method. If there are several methods in a class then complexity of an individual class is calculated by the summation of the weights of all methods. Accordingly the class complexity (CC) is given by;

$$\text{Class complexity (CC)} = \sum_{p=1}^{s} MC_p, \tag{2}$$

where s is the number of methods in a class.

The third stage (Eq 3) of the proposed metric calculates the complexity of the entire code by identifying the existing relations between classes. The complexity of the entire system (if the system consists of more than one class) is calculated considering the following two cases in the OO architecture:

(i) If the classes are in the same level then their weights are added.
(ii) If they are children of a class then their weights are multiplied due to inheritance property.

If there are m levels of depth in the OO code and level j has n classes then the cognitive code complexity (CCC) of the system is given by

$$\text{Cognitive code complexity (CCC)} = \prod_{j=1}^{m} \left[ \sum_{k=1}^{n} CC_{jk} \right]. \tag{3}$$

If there are more than one class hierarchies in a project, then we simply add CCCs of each hierarchy to calculate the complexity of the whole system.

The Class Complexity Unit (CCU) of a class is defined as the cognitive weight of the simplest software component (having a single class which includes single method and also the method includes only a linear structure). This corresponds to sequential structure in BCS and hence its cognitive weight is taken as 1. CCU is used as the basic unit for complexity.

## 3. Theoretical validation

A newly proposed metric is acceptable only when its usefulness has been proved by a validation process. For theoretical validation several researchers proposed different criteria (Briand *et al* 1996; Fenton 1993, 1994; IEEE Computer Society 1998; Kaner 2004; Kitchenham *et al* 1995; Morasca 2003; Wang 2003; Zuse 1991, 1992, 1998), to which the proposed software metric should adhere. However, in general all those aforementioned criteria suggest that the metric should fulfill some basic requirements based on measurement theory perspective. In order to

make the software more discipline and more mature, tools provided by Measurement Theory (MT) should be used. As a consequence, we define the basics of MT and evaluate the proposed metric formally from the MT perspective.

Amongst available validation criteria, the framework given by Briand *et al* (1996) is reported to be more practical and used by several researchers (Costagliola *et al* 2005). In this section, we adopt this framework since it also validates a given metric for various measurement concepts like size, length, complexity, cohesion and coupling.

Before assessing our proposed metric against this framework, it seems appropriate to provide the basic definitions and the desirable properties for complexity measures given in this framework.

***Definition*** (Representation of Systems and Modules): A system S is represented as a pair $<E, R>$, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements. Given a system $S = <E, R>$, a system $m = <E_m, R_m>$ is a module of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$ and $R_m \subseteq R$. The elements of a module are connected to the elements of the rest of the system by incoming and outgoing relationships. The set Input R(m) of relationships from elements outside module $m = <E_m, R_m>$ to those of module m is defined as

$$\text{Input R (m)} = \{< e_1,\ e_2 >\in\ R | e_2 \in E_m \text{ and } e_1 \in E - E_m\}.$$

The set Output R(m) of relationships from the elements of a module $m = <E_m, R_m>$ to those of the rest of the system is defined as

$$\text{Output R (m)} = \{< e_1,\ e_2 >\in R | e_1 \in E_m \text{ and } e_2 \in E - E_m\}.$$

For the proposed complexity metric, the entities are classes, i.e., E is a set of classes in S, and R represents a set of binary relations between classes.

Briand *et al* (1996) give the complexity definition as follows.

***Definition*** (Complexity): The complexity of a system S is a function Complexity (S) that is characterized by the properties non-negativity, null value, symmetry, module monotonicity and disjoint module additivity.

In order to make it easier to follow the theoretical evaluation of our metric for the reader, the description of properties of Briand *et al* (1996) and corresponding evaluation of the proposed metric are given below.

***Property complexity 1*** (Non-negative): The complexity of a system $S = <E, R>$ is non-negative if complexity (S) $\geq 0$.

***Proof***: Since the proposed metric is obtained by the sum of weights of non-negative numbers this property is satisfied.

***Property complexity 2*** (Null value): The complexity of a system $S = <E, R>$ is null if R is empty. This can be formulated as:

$$R = \varnothing \Rightarrow \text{complexity (S)} = 0.$$

***Proof***: Since no BCS is present in the system, the complexity value in terms of cognitive weight is trivially null and therefore this property is also satisfied by the proposed metric. In other words, if a simple OO code does not contain any method then naturally it will have no complexity in terms of weights.

***Property complexity 3*** (Symmetry): The complexity of a system S =<E, R> does not depend on the convention chosen to represent the relationships between its elements.

$$\left(\text{Let S} =< \text{E},\ \text{R} > \text{ and } \text{S}^{\{-1\}} =< \text{E},\ \text{R}^{\{-1\}} >\right) \Rightarrow \text{Complexity (S)} = \text{Complexity}\left(\text{S}^{-1}\right).$$

*Proof*: In the proposed metric, there is no effect on complexity value by changing its order or changing its representation because weights assigned to the class or the method cannot depend on the order or way of representation. Therefore, this property is also satisfied by the proposed metric.

***Property complexity 4*** (Module Monotonicity): The complexity of a system S =<E, R> is no less than the sum of the complexities of any two of its modules with no relationships in common.

$$\begin{aligned}&(\text{Let S} = \text{<E, R> and for all } m_1 = \text{<}E_{m1},\ R_{m1}\text{> and } m_2 = \text{<}E_{m2},\ R_{m2}\text{> and } m_1 \cup m_2 \subseteq \\ &\qquad \text{S and } R_{\_\{m1\}} \cap R_{\_\{m2\}} = \varnothing) \\ &\Rightarrow \text{Complexity (S)} \geq \text{Complexity } (m_1) + \text{Complexity } (m_2)\,.\end{aligned}$$

***Proof***: The conditions $m_1 \subseteq S$, $m_2 \subseteq S$ and $E = E_{m1} \cup E_{m2}$ imply that no modification is made to the classes of S when the system is partitioned into modules $m_1$ and $m_2$.

In this metric, if any class is partitioned into two classes, the sum of the complexity values of its partitioned classes will never be greater than the weights of the joined class. In other words, the complexity values for the whole will never be less than the sum of the complexity value of its module. This theorem can easily be illustrated by taking the three examples given in appendices 1, 2 and 5. In the first example, the Computer–Hardware–Software–Desktop–Notebook class (figure 1) is partitioned into two sub-classes Computer–Software (Appendix II) and Computer–Hardware–Desktop–Notebook (Appendix V) and their corresponding cognitive complexity values are 152, 24 and 128 (see table 2). Clearly, the complexity of the class Computer–Hardware–Software–Desktop–Notebook is equal to the sum of the complexities of its components, i.e., 24+128 = 152. Therefore, this property also holds by the proposed complexity metric.

***Property complexity 5*** (Disjoint Module Additivity): The complexity of a system S =<E, R> composed of two disjoint modules $m_1$, $m_2$, is equal to the sum of the complexities of the two modules.

$$\begin{aligned}&(\text{S} = \text{<E, R> and } \text{S} = m_1 \cup m_2,\ \text{and } m_1 \cap m_2 = \varnothing) \\ &\Rightarrow \text{Complexity (S)} = \text{Complexity } (m_1) + \text{Complexity } (m_2)\,.\end{aligned}$$

***Proof***: For the metric presented in this research, it can be said that the complexity value of the class which is obtained by concatenation of $m_1$ and $m_2$ is equal to the sum of their calculated

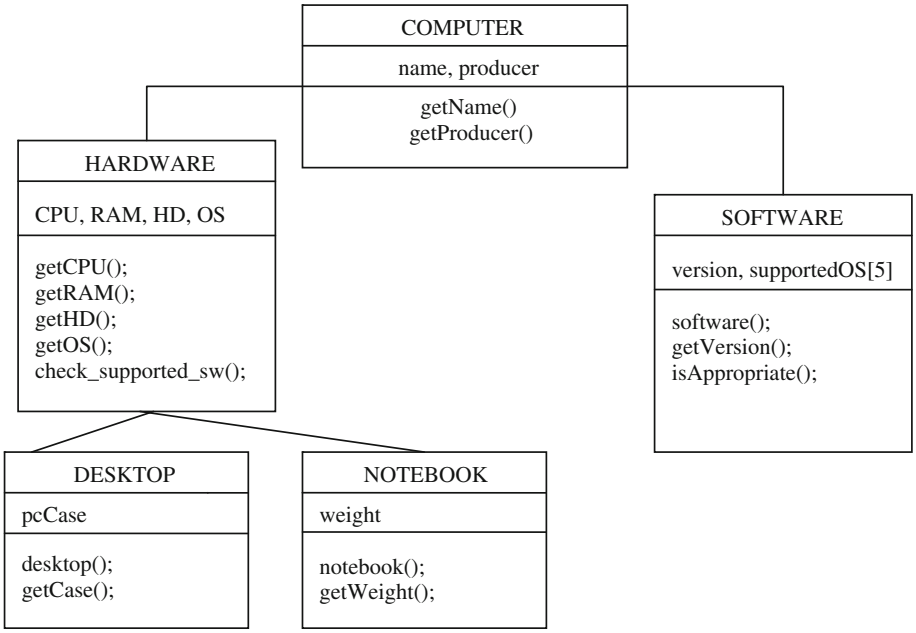



**Figure 1.** An example of an object-oriented system.

complexity values. If two independent classes are combined into a single class then the weights of the individual classes will be combined. Therefore, this property is also satisfied by the proposed metric. For a practical implementation, consider the same example given in the previous property. It can easily be said that this property is satisfied by our complexity measure since the combined CCC for code in Appendix I is 152 (code of the class hierarchy given in figure 1), which is sum of 24 (code for Appendix II) and 128 (code for Appendix V).

As consequences of the properties Complexity 1–5 given above, it is shown that adding relationships between elements of a system does not decrease its complexity. Furthermore, the proposed complexity metric holds the properties complexity 1–5, therefore it is also applicable to the admissible transformation for the ratio scale. In other terms, by fulfilling these properties, one may say that the proposed complexity metric is on the ratio scale which is the most desirable property of complexity measure from measurement theory point of view.

## 4. Experimentation and test cases

### 4.1 *Demonstration of the metric*

The applicability of the proposed metric has been checked by applying it to an OO programming that its class hierarchy is given in figure 1. The complete code of the figure is given in Appendix I. This example processes a computer database hierarchy. It has one main class Computer and two subclasses, Hardware and Software. The class Hardware has again two subclasses, Desktop and Notebook. We demonstrate how we can calculate the class complexity for an OO system. The complexity values corresponding to each class of figure 1 (see Appendix I for code and

**Table 1.** Calculated CC values for CLASSES and SUBCLASSES (see Appendix I).

| Name of Class | COMPUTER | HARDWARE | DESKTOP | NOTEBOOOK | SOFTWARE |
|---|---|---|---|---|---|
| CC | 2 | 16 | 2 | 2 | 12 |

weights) is summarized in table 1. Table 2. Consists of the cognitive code complexity (CCC) of different codes (class structures), which are the results of different combinations of the classes given in Appendices II–V.

The class complexity (CC) of each class is calculated as follows:

COMPUTER class has two methods, then $CC = \Sigma MC = W_{c1} + W_{c2} = 1 + 1 = 2$.

SOFTWARE class has three methods, then $CC = \Sigma MC = W_{S1} + W_{S2} + W_{S3} = 4 + 1 + 7 = 12$.

HARDWARE class has five methods, then $CC = \Sigma MC = W_{H1} + W_{H2} + W_{H3} + W_{H4} + W_{H5} = 1 + 1 + 1 + 1 + 12 = 16$.

DESKTOP class has two methods, then $CC = \Sigma MC = W_{D1} + W_{D2} = 1 + 1 = 2$.

NOTEBOOK class has two methods, then $CC = \Sigma MC = W_{N1} + W_{N2} = 1 + 1 = 2$

Calculation of $W_{H5}$ can be used to further clarify the calculation of complexity of methods as follows:

$W_{H5} = 1 + 2 + 7 + 2 = 12$, where 1 is due to sequential structure of this method, 2 belongs to external message call, 7 is the weight of called message ($W_{S3}$) and 2 is for 'if' statement (please see the class HARDWARE in Appendix I). This example is an indication of how we handle message calls in our proposal.

The Cognitive Code Complexity (CCC) is then calculated by using equation 3 as follows;

$$\begin{aligned} CCC &= \text{CC of class COMPUTER} * ((\text{CC of Class HARDWARE} * (\text{CC of Class} \\ &\quad \text{DESKTOP} + \text{CC of Class NOTEBOOOK}) + \text{CC of Class SOFTWARE}) \\ &= 2 * (16 * (2 + 2) + 12) \\ &= 152 \text{ CCU}. \end{aligned}$$

An analysis of these programs provides useful information about the metric. If the example in Appendix V is considered, we can find that DESKTOP and NOTEBOOK classes are on the same level and inherit the property from HARDWARE. Therefore, we add the CC of DESKTOP and NOTEBOOK (i.e., 2 + 2) and then multiply by the CC of HARDWARE (i.e. (2 + 2)*16) in order to get the complexity for HARDWARE–DESKTOP–NOTEBOOK.

**Table 2.** Calculated CCC values for different OO codes (see Appendix I–V).

| Appendix | I | II | III | IV | V |
|---|---|---|---|---|---|
| Name of classes in hierarchies | COMPUTER-HARDWARE-SOFTWARE-DESKTOP-NOTEBOOK | COMPUTER-SOFTWARE | COMPUTER-HARDWARE-DESKTOP | COMPUTER-HARDWARE-NOTEBOOK | COMPUTER-HARDWARE-DESKTOP-NOTEBOOK |
| CCC | 152 | 24 | 64 | 64 | 128 |

HARDWARE is a subclass of COMPUTER and inherits the properties from COMPUTER, therefore we multiply once more the complexity value of HARDWARE–DESKTOP–NOTEBOOK with CC of COMPUTER (i.e., ((2 + 2)*16)*2 = 128) to find the CCC of the code given in Appendix V (COMPUTER–HARDWARE–DESKTOP–NOTEBOOK). Similarly, to find CCC of entire code (figure 1), first we calculate the complexity values of classes HARDWARE–DESKTOP–NOTEBOOK (i.e., (2 + 2)*16) and SOFTWARE (i.e., ((2 + 2)*16) +12), and secondly we multiply this value with CC of the COMPUTER class (i.e., (((2 + 2)*16) +12)*2 = 152). It is because of the HARDWARE and SOFTWARE classes are on the same level and both inherit from the class COMPUTER. This example shows the usage of inheritance property of the classes in calculations. Further, when we combine programs of Appendices III and IV, and get a new program in Appendix V, we find that the complexity of the combined classes is the sum of class complexity of its components. Similarly, if we add the class complexity of Appendices II and V, we get the same complexity value of the combined classes presented in Appendix I. This is a practical example for the additive nature of the proposed metric. This also shows the scale of the metric on the ratio scale as discussed in section 3.

### 4.2 *Empirical validation*

In the previous section, we demonstrated how our metric can be applied to an OO code. However, it is not sufficient to prove the worth of a proposed metric unless it is applied on real examples. For the empirical validation of the proposed method, we preferred an open source code software project developed in C++, since the user of open source software can study it, gets knowledge of all the details and can work with it as the original author would. The Apache Xerces project is selected and used in this study. Apache Xerces is a collaborative software development project to develop free available XML parsers. This project was managed in cooperation with various individuals worldwide (both independent and company-affiliated experts), who use the internet to communicate, plan, and develop XML software and related documentation. This project is advantageous for the empirical validation of the proposed complexity measure since it contains every characteristic of an OO project, is practical for validation and easy to access for the reader. More information and the source code are available on the Internet at the address http://xerces.apache.org.

Although the whole project includes about 400 classes, we evaluated only 30 classes of this project because of the unavailability of a software tool to calculate the complexity calculations automatically. The selected 22 classes belong to a specific module (under the subdirectory 'internal' of the source code) and the 8 classes, which are connected through inheritance to the classes of that specific module, are from other modules. In addition, in calculating the class complexities of this specific module, we also included the complexity of methods from the classes of other module, which are connected through message calls. As noted in section 5, the development of a tool is the task of future work. We believe that the selected 30 classes are significant for comparison since they constitute a module (a module can also be treated as sub project) and, therefore, contain most of the characteristics of an OO system required for the validation of the proposed measure. Further, the applicability of our metric to a module also proves its scalability for large project. The cognitive complexity of the selected 30 classes and the parameters affecting the CC of classes are shown in table 3. The classes are sorted according to their CC values.

The cognitive complexity calculation of one method of the class 28 (the scanRawAttrListforNameSpaces method of the XSAXMLScanner class) is given as a tree structure in figure 2 to further clarify the CC calculations by visualizing it. In the given method, there are one loop (the "FOR" statement) and one branching (the "IF" statement) structures. In the loop, there are two

**Table 3.** Cognitive complexity of classes.

| Classes | CC | # of methods | # of method calls | # of iterations | # of branches | TOTAL |
|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 0 | 0 | 0 | 3 |
| 2 | 4 | 2 | 2 | 0 | 0 | 4 |
| 3 | 4 | 4 | 0 | 0 | 0 | 4 |
| 4 | 4 | 4 | 0 | 0 | 0 | 4 |
| 5 | 6 | 6 | 0 | 0 | 0 | 6 |
| 6 | 6 | 4 | 0 | 0 | 2 | 6 |
| 7 | 6 | 6 | 0 | 0 | 0 | 6 |
| 8 | 6 | 6 | 0 | 0 | 0 | 6 |
| 9 | 6 | 6 | 0 | 0 | 0 | 6 |
| 10 | 7 | 4 | 0 | 0 | 2 | 6 |
| 11 | 8 | 4 | 2 | 0 | 0 | 6 |
| 12 | 14 | 7 | 5 | 0 | 1 | 13 |
| 13 | 14 | 14 | 0 | 0 | 0 | 14 |
| 14 | 15 | 15 | 0 | 0 | 0 | 15 |
| 15 | 16 | 6 | 3 | 2 | 0 | 11 |
| 16 | 18 | 9 | 3 | 0 | 3 | 15 |
| 17 | 22 | 21 | 1 | 0 | 0 | 22 |
| 18 | 28 | 8 | 5 | 1 | 2 | 16 |
| 19 | 30 | 10 | 8 | 0 | 2 | 20 |
| 20 | 34 | 3 | 10 | 0 | 2 | 15 |
| 21 | 36 | 2 | 11 | 0 | 3 | 16 |
| 22 | 66 | 15 | 13 | 2 | 9 | 39 |
| 23 | 73 | 15 | 14 | 0 | 11 | 40 |
| 24 | 76 | 10 | 9 | 2 | 9 | 30 |
| 25 | 105 | 15 | 24 | 2 | 23 | 64 |
| 26 | 165 | 20 | 19 | 0 | 27 | 66 |
| 27 | 203 | 35 | 24 | 5 | 23 | 87 |
| 28 | 677 | 8 | 103 | 2 | 55 | 168 |
| 29 | 972 | 54 | 111 | 15 | 85 | 265 |
| 30 | 2706 | 56 | 236 | 84 | 104 | 480 |

external method calls and one branching, and so on. The number given in circles represents the weight of the called methods in this figure. The calculation done on each node is given around the node. For example, the value '2 + 1' given for the first external call should be considered such that '2' is the weight for the method call and '1' is the weight of the called method. The value '3 * 20' given at the right of the 'for' node shows that 20 is the total weight of the sub-branches of that node and 3 is for the iteration structure. Finally, the value '60 + 268' at the top node shows that 60 comes from the left branch and 268 comes from the right branch of the three. Other calculations are done in similar ways according to the Equation 1.

The method calls (internal and external) and the iteration and branching structures which are given in table 3 are directly related to CC, as explained in the previous sections. Even if there is no any method call, branching structure and iteration structure in a method, we assign 1 as its weight as already explained in section 2. This means, the number of methods in a class is also a factor affecting CC. For example, the CC is 3 for the first class in table 3, and there is no any method call, iteration statement and branching statement in the methods of that class. In other

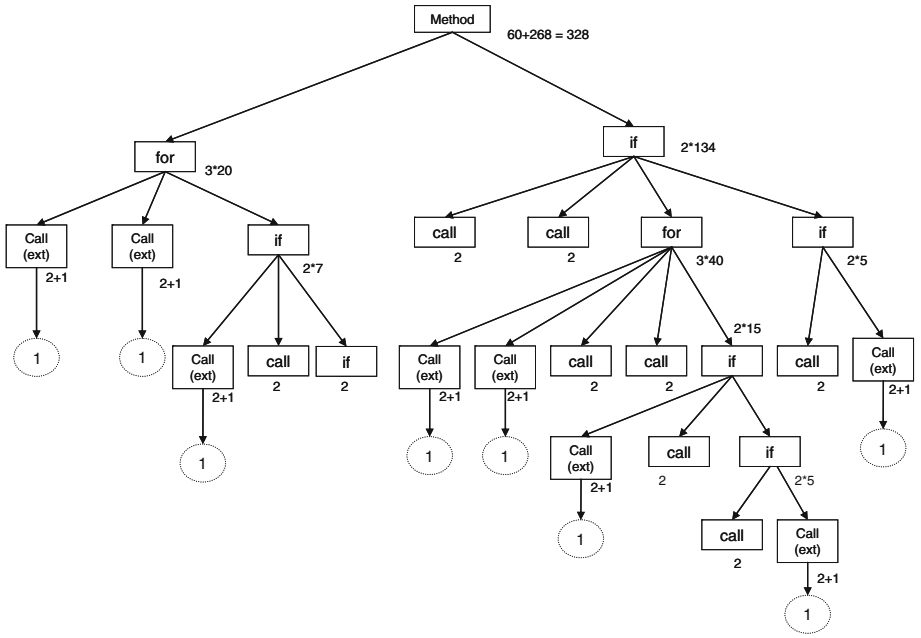



**Figure 2.** An example for cognitive complexity calculations.

words, the methods of that class include only simple sequential statements, i.e., the weight of each method is 1.

The relation between CC and affecting parameters (number of methods, number of method calls, number of loops and number of branches) can be seen in table 3. When we check the content of classes having high CC values, we see that either they may have many methods, many method calls in method definitions, many iteration and branching structures or all of them together, which increase the complexity of classes. The high complexity of classes causes difficulty in testing, debugging and maintaining the OO codes and our method gives valuable ideas about the complexity of the classes as shown in table 3. The last column of table 3 is the sum of the number of methods, method calls, iterations and branches. This column is just given to show the positive relation between the cognitive complexity and the related parameters. The CC column and total column show very similar trends since CC is computed from those parameters. When the total number of parameters increases, CC increases accordingly.

A comparative study has been done with most widely accepted Chidamber & Kemerer (CK) metrics suite (1994) using the class hierarchy given in figure 1 and also the classes given in table 3, and the results are given in tables 4 and 5, respectively.

The results show that none of the CK metrics calculates the complexity of the whole system, which is one of the differences between our metric and the CK metrics. In one of the CK metrics, namely Weighted Methods per Class (WMC), they suggested that one can calculate the weight of the method by using any procedural metric. This is similar to our approach only for calculating the weight of the method. However, our approach is one step ahead of WMC, since it establishes a proper relation between the classes by taking the most important property of OO systems: Inheritance. An additional advantage of our metric is that, unlike the CK equivalent, it

**Table 4.** Complexity values of classes given in figure 1.

| Name of CLASS / Metrics | COMPUTER | HARDWARE | SOFTWARE | DESKTOP | NOTEBOOK | Complexity for software system |
|---|---|---|---|---|---|---|
| CCC | 2 | 16 | 12 | 2 | 2 | 152 |
| WMC(1) | 2 | 5 | 3 | 2 | 2 | 14 |
| WMC(2) | 2 | 16 | 12 | 2 | 2 | 34 |
| RFC | 2 | 7 | 5 | 9 | 9 | – |
| DIT | 0 | 1 | 1 | 2 | 2 | — |
| NOC | 2 | 2 | 0 | 0 | 0 | – |
| LCOM | 0 | 2 | 3 | 2 | 2 | – |
| CBO | 0 | 1 | 0 | 0 | 0 | – |

CCC: Cognitive code complexity, WMC (1): Weighted method per class (weight of each method is assumed to be one), WMC (2): calculated WMC by cognitive weights, RFC: Response for a class, DIT: Depth of Inheritance, NOC: Number of children, LCOM: Lack of cohesion in methods, CBO: coupling between objects

takes cognitive weights into consideration. Three different complexity values (CCC, WMC (1), WMC (2)), for the whole OO hierarchy given in figure 1, are presented in table 4 (see the last column of the table). We calculated the weight of each method by using cognitive weights for WMC (2) (based on our method) and the approach suggested by Chidamber and Kemerer for WMC (1). We found that the resulting value of WMC (2) is higher than the original WMC (2). This is because, in WMC (2), the weight of each method is assumed to be one. However, including cognitive weights in calculating method complexity (WMC (2)) is more realistic because it considers the complexity of the internal architecture of methods. As seen in the table, our method (CCC) produces higher value for the whole system than the system complexity calculated by the approaches WMC (1) and WMC (1). The reason here is that, CCC gives more emphasizes to inheritance, since deep inheritance causes increased complexity and unpredictable behaviour in OO codes. Our approach is more distinguishing compared to WMC, since it considers both the internal structure of methods and OO class hierarchies. That is, although the WMC metrics may calculate the same value for two similar codes, the proposed approach obtains different results considering detailed structures.

The Depth of Inheritance Tree (DIT) and the Number Of Children (NOC) are two important CK measures. The former represents the maximum length from the node to the root of the tree and the latter is the number of immediate subclasses subordinated to a class in class hierarchy. The complexity values for both metrics vary from class to class depending on the position of class in the hierarchy. Generally, these metrics have values between 0 and 3, and they give very limited information about the complexity of classes as shown in table 5. It is difficult to get an idea about complexity of an OO code just considering DIT or NOC. This implies that DIT and NOC do not consider architectural hierarchy fully. However, in this proposal, if the depth of the inheritance tree or the number of children is high it is reflected directly in metric calculation, since we multiply the complexity of the children by their parent class. In other words, the CCC metric includes complexity due to number of children and their depth in the class hierarchy. That is, the CCC metric covers both DIT and NOC. Although this naturally results in higher complexity values for CCC, it provides valuable information about the design quality of the OO system for future maintenance without using any other measures.

**Table 5.** Different complexity values of classes given in table 3.

| Classes | # of Attributes | RFC | NOC | DIT | LCOM | CBO | CC |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 13 | 0 | 1 | 1 | 1 | 3 |
| 2 | 2 | 2 | 0 | 0 | 2 | 1 | 4 |
| 3 | 0 | 14 | 2 | 1 | 0 | 1 | 4 |
| 4 | 4 | 4 | 2 | 0 | 0 | 0 | 4 |
| 5 | 2 | 6 | 0 | 0 | 5 | 1 | 6 |
| 6 | 0 | 7 | 0 | 1 | 0 | 0 | 6 |
| 7 | 0 | 6 | 18 | 0 | 0 | 0 | 6 |
| 8 | 0 | 6 | 1 | 0 | 0 | 0 | 6 |
| 9 | 0 | 6 | 1 | 0 | 0 | 0 | 6 |
| 10 | 0 | 7 | 0 | 1 | 0 | 0 | 7 |
| 11 | 4 | 14 | 0 | 1 | 2 | 2 | 8 |
| 12 | 2 | 21 | 0 | 2 | 7 | 1 | 14 |
| 13 | 0 | 14 | 1 | 0 | 0 | 0 | 14 |
| 14 | 1 | 25 | 1 | 1 | 0 | 2 | 15 |
| 15 | 4 | 16 | 0 | 1 | 6 | 2 | 16 |
| 16 | 4 | 15 | 0 | 1 | 7 | 1 | 18 |
| 17 | 2 | 31 | 1 | 1 | 4 | 3 | 22 |
| 18 | 4 | 22 | 0 | 2 | 8 | 1 | 28 |
| 19 | 0 | 10 | 13 | 0 | 0 | 2 | 30 |
| 20 | 0 | 3 | 0 | 0 | 0 | 10 | 34 |
| 21 | 2 | 2 | 0 | 0 | 1 | 6 | 36 |
| 22 | 4 | 29 | 0 | 1 | 11 | 4 | 66 |
| 23 | 5 | 40 | 0 | 2 | 15 | 6 | 73 |
| 24 | 4 | 10 | 0 | 1 | 6 | 2 | 76 |
| 25 | 14 | 25 | 0 | 1 | 15 | 7 | 105 |
| 26 | 6 | 21 | 0 | 2 | 15 | 4 | 165 |
| 27 | 13 | 43 | 0 | 1 | 15 | 7 | 203 |
| 28 | 0 | 216 | 0 | 3 | 0 | 11 | 677 |
| 29 | 10 | 70 | 0 | 1 | 52 | 10 | 972 |
| 30 | 0 | 57 | 0 | 0 | 0 | 6 | 2706 |

Another CK metric is Response for the Class (RFC), which is defined as the total number of methods that can be executed in response to a message to a class. This count includes all the methods available in the class hierarchy. RFC is an important measure, because when RFC increases, the effort required for testing also increases (Pressman 2005). The difference between RFC and CC is due to the fact that RFC calculates only the number of methods in response to a message and our approach is sensitive to the complexity of the called method. Therefore, CC produces higher complexity values than RFC. We think that considering not only the number of methods, but also the whole complexity of methods gives more information about the maintainability of the OO code. However, a positive correlation (but not strictly) is observed between RFC and CC as shown in table 5.

The number of interactions have significant impact on the level of complexity which is directly related to modularity, maintenance, and testing of a system. Coupling Between Object classes (CBO) is a measure to show interactions between objects by counting the number of other classes to which the class is coupled. On the other hand, CC considers the message calls to other classes

and the weight of the called methods. One class may have 1 for CBO showing that it interacts with only one class, but may include many messages to that class which causes to more complex code. Therefore, we believe that CC gives more accurate information about coupling of a class. This implies that when CC increases, generally (but not always) CBO also increases as seen in table 5. This is because CC encompasses CBO.

The Lack of Cohesion in Methods (LCOM) metric is for cohesion and our method is not comparable with this metric. As seen in table 5, it is not possible to establish any relation between the LCOM values and CC values.

As a conclusion, it is observed that CCC can be used to calculate the complexity of OO codes of projects with different size. It is worth mentioning here that the features evaluated by our metric can be evaluated by different metrics but none of them is capable to indicate all these features using a single metric. The proposed metric also gives valuable idea about the design quality of OO codes. High CCC values indicate that understandability and maintainability of the code is weak. Ultimately, it helps the software developer for better design. For example, the developer, who can satisfy the user requirements through the usage of a lesser number of message calls to other classes, lesser number of inheritance classes, lesser number of branching and looping primitives, is assumed to be more skilful.

## 5. Pros-cons and recommendations for future work

A good metric is one that considers not only the number of methods, classes, subclasses and relations between them, but also the internal structure of the method. It is clear from the example that the proposed complexity metric is simple and fulfills the requirements of a good metric since it also considers the internal architecture of the member function (method). It is reported in the literature that this property is not satisfied by the other complexity metrics on method level (Chidamber & Kemerer 1994; Costagliola *et al* 2005).

The features of this metric are:

(i) It can be used to evaluate efficiency of the design. A low complexity value gives an indication of better design. A good design reduces the maintainability efforts.
(ii) It can also be used as component level design metrics. It is capable to calculate the complexity and coupling (to a certain extent) of the module.
(iii) It can be used to select the best design when more than one design alternatives are available for a software project.
(iv) It can be used to evaluate the performance of designers and developers.
(v) It calculates the cognitive complexity of the OO programs. Low cognitive complexity indicates a good design; therefore less maintenance efforts.
(vi) It can be used for the complexity of class by methods and thereby understandability of the code. It is obvious that more complex classes are less understandable and require more maintenance efforts.
(vii) This metric not only sees the complexity of the structure in method but it also considers the messages between the classes and inheritance property. In other words, it measures the important concepts of OO programs.
(viii) It is a language independent complexity metric since it uses cognitive weights, and cognitive weights of basic control structures are the same in all programming languages.

(ix) The metric is on the ratio scale, a fundamental requirement for a measure from the perspective of the measurement theory.

By considering the above features, the proposed metric can be implemented for calculating the complexity of OO systems. However, there are also some drawbacks to the proposed measures, as given below:

(i) The present method gives the complexity value in numerical terms, which are generally high for large programs. High complexity values are not desirable.
(ii) It is difficult to assign the upper and lower boundaries for the complexity values.
(iii) It is not possible to identify the underlying source of complexity with the proposed measure since it depends on several factors, such as; the number of methods, their internal architectures and the number of message calls.

In the light of experience, we propose that future work should include the following topics:

(i) Proposed metric can be extended to calculate dynamic complexity of an OO code.
(ii) A software tool should be developed to calculate our metric automatically.
(iii) Assignment of the upper and lower boundaries of the complexity values should be investigated.
(iv) Further analysis is needed for the assessment of complexity for component-based software development.
(v) More test cases and typical examples (data from the industry) should be applied to further empirical evaluation.
(vi) The proposed metric should be studied in the light of making improvements to the remaining features of OO programs.

## 6. Conclusions

A cognitive complexity metric for OO systems has been introduced. The basic motivation for proposing such a metric is to be able to calculate the cognitive complexity of the internal architecture of the methods, by considering the special feature of OO programs: inheritance. It can be used to evaluate efficiency of the design and, therefore, can be applied to early phases of software development. A good design reduces the maintainability efforts in later stages, therefore our metric provides valuable information about maintainability of the software system. The metric is evaluated through measurement theory and practically through a framework. It is found that the proposed metric is on ratio scale and satisfies most of the parameters required by practical evaluation framework. The comparative study and the application on a real project prove the robustness of the measure.

**Appendices: Classes for the case study**

**Appendix I. Classes:** COMPUTER-HARDWARE-SOFTWARE-DESKTOP-NOTEBOOK

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
```

```
        /*****************CLASS COMPUTER******************/
class computer
{
public:
    char * getName(){return name;};                    //Wc1=1
    char * getProducer() {return producer;};           //Wc2=1
protected:
    char * name;
    char * producer;
    };
```

```
        /*****************CLASS SOFTWARE******************/
class software :  public computer
{
public:
    software::software(char * cname, char * cproducer, char *
cversion, char * csupportedOS[5]);                  //WS1= WS11+WS12=1+3=4
    char * getVersion(){return version;};             //WS2=1
    short isAppropriate(char * COS);        //WS3= WS31+(WS32*WS33)=1+3*2=7
protected:
    char * version;
    char * supportedOS[5];
};
software::software(char * cname, char * cproducer, char * cversion,
char * csupportedOS[5]){                             //WS11=1(sequence)
     name= cname;
     producer=cproducer;
     version=cversion;
     for (int i=0; i<5; i++)                        //WS12=3 (for loop)
         supportedOS[i]=csupportedOS[i];
      }
short software::isAppropriate(char * COS){           //WS31=1 (sequence)
    for (int i=0;i<5;i++) {                          //WS32=3(for loop)
       if (strcmp(COS,supportedOS[i])==0)           //WS33=2(if statement)
           return true;
       }
    return false;
    }
```

```
        /*****************CLASS HARDWARE******************/
class hardware :  public computer
{
public:
    char * getCPU(){return CPU;};              //WH1=1
    int getRAM(){return RAM;};                 //WH2=1
    int getHD(){return HD;}                    //WH3=1
     char * getOS(){return OS;};               //WH4=1
    void check_supported_sw(software * s);     //WH5=WH51+WH52+WH53 =1+9+2=12
protected:
    char * CPU;
    int RAM;
```

```
    int HD;
    char * OS;
};
void hardware::check_supported_sw(software * s){//WH51=1(sequence)
    short result=s->isAppropriate(OS);        //WH52=2(call)+7(weight
                                                of Called method(WS3))
    if (result)                                 //WH53=2 (if statement)
       cout<<"This is an appropriate software for this
computer"<<'\n';
    else
       cout<<"This is NOT an appropriate software for this
computer"<<'\n';
    }
```

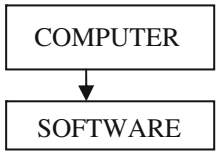
```
          /*****************CLASS DESKTOP******************/

class desktop :  public hardware
{
public:
    desktop(char * cname, char * cproducer, char * ccpu, int cram, int
chd, char * cos, char * ccase);                    //WD1=1
    char * getCase(){return pccase;};              //WD1=1
protected:
    char * pccase;
};
desktop::desktop(char * cname, char * cproducer, char * ccpu, int
cram, int chd, char * cos, char * ccase){
     name= cname;
     producer=cproducer;
     CPU=ccpu;
     RAM=cram;
     HD=chd;
     OS=cos;
     pccase=ccase; }
```

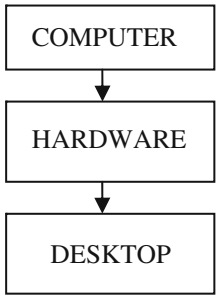
```
          /*****************CLASS NOTEBOOK******************/
class notebook :  public hardware
{
public:
    notebook(char * cname, char * cproducer, char * ccpu, int cram,
int chd, char * cos, float weight);                //WN1=1
    float getWeight(){return weight;};             //WN2=1
protected:
    float weight;
};
notebook::notebook(char * cname, char * cproducer, char * ccpu, int
cram, int chd, char * cos, float cweight){
     name=cname;
     producer=cproducer;
     CPU=ccpu;
     RAM=cram;
     HD=chd;
     OS=cos;
     weight=cweight; }
```

```
/* ==========================Main Program======================== */
int main ()
{
    char * supportedOS[5];
    supportedOS[0]="MS Windows";
    supportedOS[1]="Linux";
    supportedOS[2]="";
    supportedOS[3]="";
    supportedOS[4]="";
    software * sw1 = new software ("Rational", "IBM"', "7.1",
supportedOS);
    desktop * ds1 = new desktop("vaio", "Sony", "Intel core duo",
1024, 120, "MS Vista","All-in-One");
    notebook * nb1 = new notebook ("Pavillion", "HP", "Intel core
duo", 1024, 120, "Linux",2.5);
    cout<<ds1->getCase()<<'\n';
    ds1->check_supported_sw(sw1);
    cout<<nb1->getHD()<<'\n';
}
```

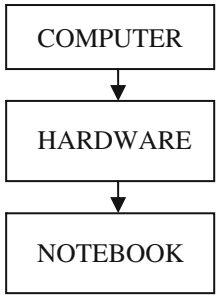
**Appendix II. Classes:** COMPUTER- SOFTWARE



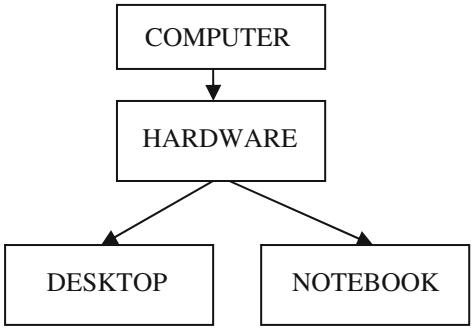

**Appendix III. Classes:** COMPUTER-HARDWARE- DESKTOP



**Appendix IV. Classes:** COMPUTER-HARDWARE-NOTEBOOK




**Appendix V. Classes:** COMPUTER-HARDWARE- DESKTOP-NOTEBOOK




## References

Babsiya J, Davis C G 2002 A hierarchical model for object oriented design quality assessment. *IEEE Transactions on Software Eng.* 28: 4–17

Basci D, Misra S 2009a Measuring and Evaluating a Design Complexity Metric for XML Schema Documents. *J. Information Sci. and Eng.* 2009; 25(5): 1405–1425

Basci D, Misra S 2009b Data Complexity Metrics for Web-Services. *Advances in Electrical and Computer Eng.* 9(2): 9–15

Basily V R, Briand L C, Melo W L 1996 A validation of object oriented design metrics as quality indicators. *IEEE Transactions on Software Eng.* 22(1): 751–761

Binder RV 1994 Object-oriented software testing. *Communications of the ACM;* 37(9): 28–29

Briand L C, Wust J 2001 Modelling development effort in object oriented system using design properties. *IEEE Transactions on Software Eng.* 27(11): 963–986

Briand L C, Bunse C, Daly J W 2001 A controlled experiment for evaluating quality guidelines on maintainability of object oriented design. *IEEE Transactions on Software Eng.* 2(6): 513–530

Briand L C, Morasca S, Basily V R 1996 Property based software engineering measurement. *IEEE Transactions on Software Eng.* 22(1): 68–86

Chidamber S R, Kemerer C F 1994 A metrics suite for object oriented design. *IEEE Transactions on Software Eng.* 6: 476–493

Costagliola G, Ferrucci F, Tortora G, Vitiello G 2005 Class points: An approach for the size estimation of object-oriented systems. *IEEE Transactions on Software Eng.* 31(1): 52–74

Fenton N 1993 New software quality metrics methodology standards fills measurement needs. *IEEE Computer* 26(4): 105–106

Fenton N 1994 Software measurement: A necessary scientific Basis. *IEEE Transactions on Software Eng.* 20(3): 199–206

Gupta V, Chhabra J K 2009 Package coupling measurement in object-oriented software. *J. of Computer Sci. and Technol.* 24(2): 273–283

Harrison R, Counsell S J, Nithi R V (1998) An evaluation of the MOOD set of Object Oriented Software Metrics. *IEEE Transactions on Software Eng.* 24(6): 491–496

Henderson-Sellers B 1996 *Object-oriented metrics, measure of complexity.* (Englewood Cliffs, N.J.: PTR, Prentice-Hall)

IEEE Computer Society 1998 *Standard for software quality metrics methodology.* Revision IEEE Standard, 1061–1998

Kaner C 2004 Software Engineering Metrics: What do they measure and how do we know?. *Proc. 10th Int. Software Metrics Symposium* (Metrics 2004), 1–10

Kim J, Lerch J F 1991 Cognitive processes in logical design: comparing object-oriented and traditional functional decomposition software methodologies. Carnegie Mellon University, Graduate School of Industrial Administration, Working Paper

Kim K, Shin Y, Wu C 1995 Complexity measures for object-oriented program based on the entropy. *Proc. Asia Pacific Software Eng.* 127–136

Kitchenham B, Pfleeger S L, Fenton N 1995 Towards a framework for software Measurement Validation. *IEEE Transactions on Software Eng.* 21(12): 929–943

Lorenz M, Kidd J 1994 *Object-oriented software metrics.* Englewood Cliffs, New Jersey: Prentice Hall

Marinescu R 2005 Measurement and quality in Object–oriented design. *Proc. 21st IEEE International Conference on Software Maintenance,* Beijing, 701–704

Misra S, Akman I 2008a Measuring complexity of object oriented programs. *Proc. Int. Conf. Computational Science and Application,* Perigua, Italy: 652–667

Misra S, Akman I 2008b Weighted class complexity: A measure of complexity for object –oriented system. *J. Information Sci. Eng.* 24: 1689–1708

Misra S, Akman I 2008c A complexity metric based on cognitive informatics. *Proc. RSKT 2008,* Chengdu, China; 620–627

Morasca S 2003 Foundations of a weak measurement-theoretic approach to software measurement. *Lecturer Notes in Computer Science* (LNCS); 2621: 200–215

Olague H M, Etzkorn L H, Gholston S, Quattlebaum S 2007 Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Eng.* 33(6): 402–419

Purao S, Vaishnavi V K 2003 Product metrics for object oriented systems. *ACM Computing Surveys* 35(2): 191–221

Pressman R S 2005 *Software engineering: A practitioner's approach,* sixth edition, New York: McGraw Hill

Reißing R 2001 Towards a model for object-oriented design measurement. In *Proc. of the 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 71–84. Budapest, Hungary

Stephen H K 2003 Metrics and lessons learned for OO projects. Chapter 12, in Metrics and Models in Software Quality Engineering, Second Edition, Addison-Wesley

Vaishnavi V K, Purao S, Liegle J 2007 Object-oriented product metrics: A generic framework. *Information Science* 177: 587–606

Wand Y, Weber R 1990 Toward a theory of the deep structure of information systems. *Proc. Int. Conf. on Information Systems,* Copenhagen, Denmark: 61–71

Wang Y 2003 The measurement theory for software engineering. *Proc. Canadian Conference on Electrical and Computer Eng.* (CCECE 2003); 1321–1324

Wang Y, Shao J 2003 A new measure of software complexity based on cognitive Weights. *Canadian J. of Electrical and Computer Eng.* 28(2): 69–74

Weyuker E 1988 Evaluating software complexity measures. *IEEE Transactions on Software Eng.* 14: 1357–1365

Wilde N, Huitt R 1992 Maintenance support for object-oriented programs. *IEEE Transactions on Software Eng.* 18: 1038–1044

Zuse H 1991 *Software complexity measures and methods.* Berline: Walter de Gruyter

Zuse H 1992 Properties of software measures. *Software Quality Journal* 1: 225–260

Zuse H 1998 *A framework of Software Measurement.* Berline: Walter de Gruyter