Computing and Informatics, Vol. 30, 2011, 267–293

NEW INHERITANCE COMPLEXITY METRICS FOR OBJECT-ORIENTED SOFTWARE SYSTEMS: AN EVALUATION WITH WEYUKER'S PROPERTIES

Deepti Mishra

Department of Computer Engineering
Atilim University
Ankara, Turkey
e-mail: deepti@atilim.edu.tr

Manuscript received 30 July 2008; revised 12 April 2010
Communicated by David Darcy

Abstract. Two inheritance complexity metrics, one at class level CCI (Class Complexity due to Inheritance) and another at program level ACI (Average Complexity of a program due to Inheritance), have been proposed for object-oriented software systems. These proposed metrics are evaluated with Weyuker's properties and compared with other well known object-oriented inheritance metrics. It has been found that the proposed metrics better represent the complexity, due to inheritance, of a class and a program. Weyuker's property 7 (Significance of Permutation) has received a negative response regarding its applicability to object-oriented software metrics. It has been observed that this property is not satisfied by any of the object-oriented inheritance metrics proposed so far. Contrary to past beliefs, the relevance of this property to object-oriented systems has been brought out in this paper. Examples with C++ code are also presented to support the applicability of this property.

Keywords: Weyuker's properties, software metrics, object-oriented systems, inheritance, complexity

1 INTRODUCTION

Object-oriented metrics have been successfully applied in various domains and programming languages in countries worldwide [15]. They have consistently demon-

strated relationships to quality factors such as costs, defects, reuse, and maintainability [15] which is due to the presence of good programming style [33]. Inheritance is one of the most powerful features of an object-oriented paradigm. Programming without inheritance is distinctly not object-oriented programming and may be called programming with abstract data type [35]. Many object-oriented metrics have been proposed over the last decade and some are actually being used by a number of organizations as part of an effort to manage quality [16]. Some of the well known object-oriented inheritance metrics are proposed by Chidamber and Kemerer [12], Abreu and Carapuca [2], Henderson-Sellers [22], Lorenz and Kid [29], and Li [27]. These inheritance metrics are based on

- depth of the inheritance tree
- total numbers of classes inherited in a program
- number of classes inherited (directly/indirectly) by a class
- number of classes inheriting (directly/indirectly) from a class
- number of methods inherited by a class
- number of methods overridden by a class etc.

Complexity due to inheritance is not only dependent on the number of classes inherited but also on the number of methods inherited and the complexity of these methods. This is also supported by Abreu and Carapuca [2] that the greater the inheritance relation is, the greater the number of methods a class is likely to inherit, making it more complex and therefore requiring more testing. A method with a complex decision structure will be harder to test and maintain and is more errorprone [2]. They suggested that complexity metrics based on the above criteria allow to pin-point potentially troublesome classes or methods, thus helping in the planning of the review and test efforts [2] which are fundamental components of the software quality process [34]. There is no comprehensive object oriented inheritance metric that takes into account all these issues. Therefore, the main motivation of this paper is to propose inheritance metrics that takes into account not only the number of classes inherited but also the number and complexity of the methods inherited.

In this paper, two metrics, one at class level CCI (Class Complexity due to Inheritance) and another at program level ACI (Average Complexity of a program due to Inheritance), have been proposed to evaluate complexity (due to inheritance) of object-oriented programs. These metrics will help identifying classes and programs having high complexity. This identification can lead to either channeling more testing efforts to improve quality or redesigning programs to reduce the complexity.

Any proposed metric should have sound theoretical and mathematical foundation and should also be relevant to practitioners in software development organizations. Weyuker [44] and other researchers [32, 37] have proposed sets of properties to serve as a basis for the evaluation of complexity metrics. Several researchers have evaluated their metric suites against Weyuker's properties [2, 11, 12, 19, 39, 40]

while others are skeptical about these properties [18, 24, 45]. Chidamber and Kemerer [12] proposed a metric suite for object oriented design and also evaluated it by Weyuker's set of properties. They claimed that Weyuker's property 7 should not be considered for object oriented metrics. Weyuker's property 7 states that changing the order of statements may affect the complexity of the program. They further supported their argument by stating that "Cherniavsky and Smith specifically suggest that this property is not appropriate for OOD metrics because the rationales used may be applicable only to traditional progamming" [12, p. 481]. In fact, Cherniavsky and Smith [11] stated this not specifically for property 7 but for all 9 Weyuker's properties that many of the rationale used for the properties apply only to traditional programming languages and object oriented programming languages will require a different set of properties [11, p. 638]. They concluded that, at best, satisfying all the nine properties is a necessary, but not sufficient, condition for a good complexity measure. The following reason has been given to support the non applicability of property 7 for object-oriented systems: In OOD, a class is an abstraction of the problem space, and the order of statements within the class definition has no impact on its eventual execution and use [12]. For example, changing the order in which methods are declared does not affect the order in which they are executed, since methods are triggered by the receipt of different messages from other objects [12]. We agree with this argument as far as order of the statements within a class are concerned. However, methods are parts of a class and the order of statements within a method will affect the execution and complexity of the method which will eventually affect the complexity of the class as a whole.

This paper is organized as follows: In the next section, related work is described. Two new inheritance complexity metrics and one method complexity metric have been proposed in Section 3. These proposed inheritance complexity metrics are evaluated against Weyuker's properties in Section 4. In Section 5, proposed metrics are compared with some well known inheritance metrics for object-oriented systems. Finally, the paper concludes in Section 6.

2 LITERATURE SURVEY

Many metrics have been proposed to evaluate the quality of object-oriented systems which can aid developers in understanding design complexity, in detecting design flaws, and in predicting certain quality outcomes such as software defects, testing and maintenance efforts [12, 22, 30]. Several empirical validation studies [1, 3, 38, 13, 26, 17, 5] of Chidamber and Kemerer's metrics suite [12] as well as Abreu and Carapuca's MOOD (Metrics for Object-Oriented Design) metric set [2] suggest that these metrics are important indicators of external quality factors. There is evidence that design metrics are related to a variety of quality characteristics of software products such as reliability, testability and maintainability [42].

It is not possible to conduct an exhaustive inspection of code or a thorough testing of all modules due to lack of resources such as people and time. Therefore,

available resources for quality control must be cleverly applied to cover at least 20 % of the total modules that usually stand for more than 80 % of the faults (the "old" Paretto Law) [23]. Experimental data confirms that the phenomenon of defect clustering still holds for object-oriented systems [43]. Complexity metrics can help finding those 20 % modules [2]. Complexity metric for a program code is a well known metric since it is a good indicator of a well-designed, understandable, and easy to modify program. Inheritance is one of the key features of object-oriented paradigm as it promotes reuse. Many studies [4, 10, 12, 28] have claimed that the use of inheritance reduces the amount of software maintenance necessary and eases the burden of testing. The reuse of software through inheritance is claimed to produce more maintainable, understandable and reliable software [5, 6, 7]; but Harrison, Counsell and Nithi [21] contradict this through their experimental assessment that systems without inheritance are easier to modify than corresponding systems containing three or five levels of inheritance. Also, they found [21] that systems without inheritance are easier to understand than corresponding systems containing three levels of inheritance. Lattanzi and Henry [25] found in their comparative study that inheritance relationship seems to be detrimental to software productivity. They found a correlation between reuse through inheritance and the number of integration errors. Ordinarily, it is agreed that the deeper the inheritance hierarchy, the better the reusability of classes, but the higher the coupling between inherited classes, the harder it is to maintain the system [41]. The designers may tend to keep the inheritance hierarchies shallow, forsaking reusability through inheritance for simplicity of understanding [12]. Therefore, it was concluded that deriving new classes from existing library classes makes the system harder to integrate correctly. The obvious reason is that to inherit a new class, the parent's implementation must be, at least partially, understood as well as any of the parent's ancestors. Although inheritance within an object-oriented system is a great way to enhance the readability and internal organization of a large program, inheriting from classes designed and written by other programmers (library classes) can prove too costly in terms of time and effort required to understand the implementation of the library classes.

3 PROPOSED INHERITANCE METRICS

Two metrics for inheritance, Class Complexity due to Inheritance (CCI) and Average Complexity of a program due to Inheritance (ACI) are proposed.

Also, one more metric is proposed to calculate the complexity of a method (MC) which is based on McCabe's cyclomatic complexity [31] but it also takes into account the depth of control structures. McCabe's cyclomatic complexity of two programs, one having two sequential loops and the other having same loops nested, is the same. This is not an ideal situation because the complexity of a program increases with nesting. This is also supported by Piwowarski [36].

Method Complexity (MC)

$$MC = P + D + 1, \tag{1}$$

where

- P is the number of predicates in a method,
- D is the maximum depth of control structures in a method; if there is no nested control structures then $D = 0$; if there is one inside another then $D = 1$ and so on.

Class Complexity due to Inheritance (CCI)

$$CCI_i = \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j, \tag{2}$$

where

- CCI_i is the complexity of an ith class due to inheritance,
- k is the number of classes, i^{th} class is inheriting,
- CCI_{ifrom} is the complexity of a parent class, i^{th} class is inheriting,
- l is the number of methods in i^{th} class,
- MC_j is the complexity of jth method in i^{th} class.

Average Complexity of a program due to Inheritance (ACI)

$$ACI = \frac{\sum_{i=1}^{n} CCI_i}{n}, \tag{3}$$

where

- n is the total number of classes in the program,
- CCI_i is the complexity, due to inheritance, of ith class in the program.

Inheritance metrics should not only consider the number of classes a particular class is inheriting but also the complexity of the inherited classes. Complexity of an inherited class can be calculated by considering the number of methods and complexities of these methods. However, constructors and destructors are not inherited but the default constructor and destructor of the base class is always called whenever an object of the derived class is created or destroyed. Also, other constructors of the base class can be called in the constructor of the derived class. So, constructors and destructors are included in the calculations. It should also be considered whether a class is inherited as it is or some of the methods are overridden. This can be taken into account by adding the method complexities of the inheriting class. Consider program 1 and program 2 given in the appendix. Let us just consider the base class Person and one derived class Employee in program 1. Similarly, let us just consider

base class Shape and one derived class Triangle in program 2. Now, both these programs are similar as both have one base class and one derived class. Complexity due to inheritance is as follows:

Program 1: Complexity of base class Person can be calculated as

$$\begin{aligned} CCI_i &= \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j \\ &= 0 + (MC_1 + MC_2 + MC_3) \\ &= 3 \end{aligned}$$

$$\left(\sum_{ifrom=1}^{k} CCI_{ifrom} = 0 \text{ as base class is not inheriting any class} \right).$$

Complexity of derived class Employee can be calculated as

$$\begin{aligned} CCI_i &= \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j \\ &= 3 + (MC_1 + MC_2 + MC_3) \\ &= 3 + 3 = 6 \end{aligned}$$

$$\left(\sum_{ifrom=1}^{k} CCI_{ifrom} = 3 \text{ as it is only inheriting class Person with } CCI = 3 \right).$$

Average Complexity of full program due to inheritance (ACI)

$$= \frac{\sum_{i=1}^{n} CCI_i}{n} = (3+6)/2 = 9/2 = 4.5.$$

Program 2: Complexity of base class Shape can be calculated as

$$\begin{aligned} CCI_i &= \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j \\ &= 0 + (MC_1 + MC_2 + MC_3 + MC_4 + MC_5 \\ &\quad + MC_6 + MC_7 + MC_8 + MC_9 + MC_{10}) \\ &= 10 \end{aligned}$$

$$\left(\sum_{ifrom=1}^{k} CCI_{ifrom} = 0 \text{ as base class is not inheriting any class} \right).$$

Complexity of derived class Triangle can be calculated as

$$CCI_i = \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j$$

$$= 10 + (MC_1 + MC_2 + MC_3 + MC_4)$$
$$= 10 + 4 = 14$$

$$\left(\sum_{ifrom=1}^{k} CCI_{ifrom} = 10 \text{ as it is only inheriting class Shape with } CCI = 10 \right).$$

Average Complexity of full program due to inheritance (ACI)

$$= \frac{\sum_{i=1}^{n} CCI_i}{n} = (10 + 14)/2 = 24/2 = 12.$$

If the inheritance metric only considers the number of classes inherited then it gives value 1 for both examples; but if the number of classes inherited, the complexity of the inherited classes as well as the complexity of the derived class are all taken into consideration then the value of inheritance metric for program 1 and program 2 are 4.5 and 12, respectively. These values are more reasonable as program 1 is simpler than program 2 in terms of inheritance. The derived class is inheriting fewer methods from base class in program 1 whereas the derived class is inheriting more methods in program 2.

4 EVALUATION OF PROPOSED METRICS BY WEYUKER'S PROPERTIES

Weyuker [44] has proposed a set of properties of syntactic software complexity measures to serve as a basis for the evaluation of such measures. These types of formalized evaluations help clarify the strengths and weaknesses of existing and proposed complexity measures, aid in the selection of appropriate measures and ultimately lead to the definition of a better measure by emphasizing important properties [44]. A good complexity measure should satisfy most of Weyuker's properties.

Property 1: Non-coarseness – $(\exists P)(\exists Q)(|P| \neq |Q|)$ where P and Q are two different classes.

This means different classes should have different values for metrics, as far as possible. This property is satisfied by proposed metrics CCI and ACI.

Class Employee (program 1) and class Rectangle (program 2) have different CCI values 6 and 14, respectively, although they both are inheriting just one class and their level is also the same. Also, the ACI for two programs (program 1 has ACI = 8.67 and program 2 has ACI = 13.5) is different.

Property 2: Granularity – Let c be a non-negative number. Then there are only finitely many programs of complexity c.

This property states that there will be a finite number of cases having the same metric value. Since the universe of discourse deals with at most a finite set of applications, each of which has a finite number of classes, this property will be

met by any metric measured at the class level [2]. Therefore, this property is satisfied by proposed class level metric CCI. The other proposed metric at program level (ACI) is the sum of the complexities of all classes in a program divided by the number of classes in that program. Programs are made by combining classes and there are a finite number of classes of the same complexity. Therefore, ACI satisfies this property.

Property 3: Non-uniqueness – There are distinct classes P and Q such that $|P| = |Q|$.

This means that two different classes may have the same metric value. This property is satisfied by proposed metrics CCI and ACI. Class Sal_Emp (program 1) and class Shape (program 2) have the same CCI = 10. Also, classes within a program can have the same complexity, e.g. class Employee and class Student both have CCI = 6 (program 1). Moreover, the ACI for two programs can also be the same.

Property 4: Design Implication – $(\exists P)(\exists Q)(P \equiv Q \; and \; |P| \neq |Q|)$.

This means that if two designers implement the same class or program, the metric values need not be identical. This property is satisfied by proposed metrics CCI and ACI. CCI depends on the internal structure of the class such as the number and complexity of inherited classes, number of methods within the class and the complexity of each method. Even if a class is producing the same output, its internal structure may be different when implemented by two different people and therefore will produce a different CCI value. Similarly at program level, ACI gives different values depending on the internal structure of a program such as the number of classes, the complexities of these classes, the type of inheritance, the level of inheritance, etc. Both these metrics are implementation based so their value will be different for different implementations even if they provide the same functionality.

Property 5: Monotonicity – $(\exists P)(\exists Q)(|P| \leq |P;Q| \;\&\; |Q| \leq |P;Q|)$.

This means that if we concatenate two classes, the combined metric value must be greater than (or at least equal to) the greatest value from the two base classes. When any two classes P and Q are combined, there are three possible cases:

1. When P and Q are siblings.
   Consider program 1, class Employee and class Student are siblings of each other and both have CCI = 6. If these two classes are combined, there are two ways of combination:
   (a) Keep class Employee methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as they are. In this case, the combined class $(P + Q)$ will have four methods of complexity 1 each and also their constructors. Therefore, CCI$(P + Q)$ = Complexity of inherited class Person + complexities of all methods = 3 + 6 = 9 which is greater than both CCI(P) and CCI(Q).

(b) Combine Employee methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as one single get_info() and put_info() method with one "if .. else", so that the method can selectively get or put either employee or student information. In this case, the combined class $(P+Q)$ will have two methods of complexity; 2 each and also their constructors. Therefore CCI$(P+Q) = 3+6 = 9$ which is greater than both CCI(P) and CCI(Q).

2. When one is the child of another.

   Consider program 1, Class Person (CCI = 3) is the parent of the class Employee (CCI = 6). If we combine these two classes, then there are again the two possibilities mentioned above as both classes have methods get_info() and put_info().

   In case (a), combined class $(P+Q)$ will have four methods of complexity; 1 each and also their constructors. Therefore CCI$(P+Q) = 6$ which is equal to CCI(Q).

   In case (b), combined class $(P+Q)$ will have two methods (namely get_info(), put_info()) of complexity; 2 each and also their constructors. Therefore CCI$(P+Q) = 6$ which is equal to CCI(Q).

3. When P and Q are neither children nor siblings of each other.

   Consider program 1, class Student (CCI = 6) and class Sal_Emp (CCI = 10). If these two classes are combined, then the right place for $(P+Q)$ is the current place of class Sal_Emp so that it can inherit from both class Person as well as class Employee. These classes can be combined in two ways:

(a) Either keep class Sal_Emp methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as they are. In this case, the combined class $(P+Q)$ will have four methods and also their constructors. Therefore, CCI$(P+Q)$ = Complexity of inherited class Employee + complexities of all methods = $6+7=13$ which is greater than both CCI(P) and CCI(Q).

(b) Combine Sal_Emp methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as one single get_info(), put_info() method with one "if .. else", so that the method can selectively get or put either Sal_Emp or Student information. In this case, combined class $(P+Q)$ will have two methods get_info() with CCI value 4 $(P+D+1 = 2+1+1 = 4)$ and put_info() with CCI value 2 $(P+D+1 = 1+0+1 = 2)$ each and also their constructors. Therefore CCI$(P+Q) = 6+8 = 14$ which is greater than both CCI(P) and CCI(Q).

Based on the above explanations, this property is satisfied by proposed metric CCI.

This property can also be satisfied by program level metric ACI by considering two programs and then combining them.

Property 6: Non-equivalence of Interaction –

(a) $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |P;R| \neq |Q;R|)$
(b) $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |R;P| \neq |R;Q|)$

This means that if two classes, P and Q, have equal metric values, then combining a third class, say R, with each of them may produce different metric values (i.e., concatenation of P and R need not be equal to concatenation of Q and R, as far as metrics are concerned).

This property is satisfied by both proposed metrics CCI and ACI. Consider program 2, class Triangle and class Rectangle both have CCI = 14. If we add class NmTriangle to both of them, Class Triangle CCI will be 16, whereas class Rectangle CCI will be 15. This is because class NmTriangle has a method displayName() which is already there in class Rectangle whereas it is not present in class Triangle. So, class Triangle will have displayName() method whereas class Rectangle will not have this method as it is already present. The same can be proven for metric ACI as adding a new program to two existing programs will have a different effect.

Property 7: Significance of Permutation – There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $|P| \neq |Q|$. This means that if two classes differ only in the ordering of elements within them, it cannot be assumed that the metric values will be identical.

Consider program 1 (ACI = 8.67), class Hour_Emp (CCI = 14) has a method get_info() ($MC_2 = P + D + 1 = 4 + 1 + 1 = 6$) that contains a "nested if .. else" with depth $D = 1$. If we change the order of the statement, i.e. 2 sequential "if .. else" are used instead of one "nested if .. else" then get_info() method complexity will be $MC_2 = P + D + 1 = 4 + 0 + 1 = 5$.

So, class Hour_Emp CCI can be calculated as

$$\begin{aligned} & \sum_{ifrom=1}^{k} CCI_{ifrom} + \sum_{j=1}^{l} MC_j \\ = \ & 6 + (MC_1 + MC_2 + MC_3) \\ = \ & 6 + (1 + 5 + 1) = 6 + 7 = 13 \end{aligned}$$

which is now different than the previous value 14.

As class Hour_Emp CCI has changed so ACI (Average complexity of program due to inheritance) will also change. So, ACI will be $(3+6+6+10+13+13)/6 = 51/6 = 8.5$.

This property is satisfied by proposed metric CCI as well as ACI.

Property 8: No change on renaming – If P is a renaming of Q, then $|P| = |Q|$.

This property is satisfied by proposed metric CCI as well as ACI as both are implementation based. Changing the name of class will not affect CCI and therefore ACI will not be affected too.

Property 9: Interaction Complexity – This means that the complexity of two interacting classes will be greater than (or at least equal to) the sum of the metrics of individual classes. $(\exists P)(\exists Q)(|P| + |Q| < |P; Q|)$ This property is satisfied by proposed metric ACI. We will consider a program having two classes P and Q. We will consider ACI values before and after the composition of P and Q. When any two classes P and Q are combined, there are three possible cases:

1. When P and Q are siblings.

   Consider program 1 (ACI = 8.67), class Employee and class Student are siblings of each other and both have CCI = 6. If these two classes are combined, there are two ways for combination:

   (a) Keep class Employee methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as they are. In this case, combined class $(P + Q)$ will have four methods of complexity; 1 each and also their constructors. Therefore, CCI$(P + Q)$ = Complexity of inherited class Person + complexities of all methods $= 3 + 6 = 9$.

   (b) Combine Employee methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as one single get_info() and put_info() methods with one "if .. else", so that the method can selectively get or put either employee or student information. In this case, combined class $(P + Q)$ will have two methods of complexity; 2 each and also their constructors. Therefore CCI$(P + Q) = 3 + 6 = 9$.
   Now, there are 5 classes left instead of 6, i.e. Person, class $(P + Q)$, Sal_Emp, Hour_Emp,Comm_Emp. But the CCI value of classes Sal_Emp, Hour_Emp, Comm_Emp will change now because they are now inheriting class $(P + Q)$ instead of class Employee. Changed value of CCI for Sal_Emp, Hour_Emp, Comm_Emp is 13, 17, and 16, respectively. Therefore, ACI after composition of P and Q $= (3 + 9 + 13 + 17 + 16)/5 = 11.6$ which is greater than ACI (8.67) before composition of P and Q.

2. When one is child of another.

   Consider program 1 (ACI = 8.67), Class Person (CCI = 3) is parent of class Employee (CCI = 6). If we combine these two classes, then there are again the two possibilities mentioned above as both classes have methods get_info() and put_info().

   In case (a), combined class $(P + Q)$ will have four methods of complexity; 1 each and also their constructors. Therefore, CCI$(P + Q) = 6$.

   In case (b), combined class $(P+Q)$ will have two methods (namely get_info(), put_info()) of complexity; 2 each and also their constructors. Therefore CCI$(P + Q) = 6$.

Now, there are 5 classes left instead of 6 i.e. class $(P+Q)$, Student, Sal_Emp, Hour_Emp, Comm_Emp; but the CCI value of Student will change now because it is now inheriting class $(P+Q)$ instead of class Employee. The changed value of CCI for Student is 9. Therefore, ACI after composition of P and $Q = (6+9+10+14+13)/5 = 10.4$ which is greater than ACI (8.67) before composition of P and Q.

3. When P and Q are neither children nor siblings of each other.

   Consider program 1 (ACI = 8.67), class Student (CCI = 6) and class Sal_Emp (CCI = 10). If these two classes are combined, then the right place for $(P+Q)$ is the current place of class Sal_Emp so that it can inherit from both class Person as well as class Employee. These classes can be combined in two ways:

   (a) Keep class Sal_Emp methods (get_info(), put_info()) and class Student methods (get_info(),put_info()) as they are. In this case, combined class $(P+Q)$ will have four methods and also their constructors. Therefore, CCI$(P+Q)$ = Complexity of inherited class Employee + complexities of all methods $= 6+7 = 13$.
   Now, there are 5 classes left instead of 6, i.e. Person, Employee, class $(P+Q)$, Hour_Emp,Comm_Emp. Therefore, ACI after composition of P and $Q = (3+6+13+14+13)/5 = 9.8$ which is greater than ACI (8.67) before composition of P and Q.

   (b) Combine Sal_Emp methods (get_info(), put_info()) and class Student methods (get_info(), put_info()) as one single get_info(), put_info() methods with one "if .. else", so that the method can selectively get or put either Sal_Emp or Student information. In this case, combined class $(P+Q)$ will have two methods get_info() with CCI value 4 $(P+D+1 = 2+1+1 = 4)$ and put_info() with CCI value 2 $(P+D+1 = 1+0+1 = 2)$ each and also their constructors. Therefore CCI$(P+Q) = 6+8 = 14$. Now, there are 5 classes left instead of 6, i.e. Person, Employee, class $(P+Q)$, Hour_Emp, Comm_Emp. Therefore, ACI after composition of P and $Q = (3+6+14+14+13)/5 = 10$ which is greater than ACI (8.67) before composition of P and Q.

Therefore, we can say that this property is satisfied by proposed metric ACI.

5 COMPARISON WITH OTHER INHERITANCE METRICS

Some well known inheritance metrics are summarized in Table 1. Metrics which are validated theoretically and most importantly empirically are considered in this study. Chidamber and Kemerer's metrics suite [12] as well as Abreu and Carapuca's MOOD (Metrics for Object-Oriented Design) [2] are probably most widely referenced set of object oriented metrics. Some of the studies which validated the metrics considered in this study are: Chidamber and Kemerer metrics suite [12, 5], Abreu

and Carapuca [1, 20], Henderson-Sellers [41, 14, 8, 9], Lorenz and Kid [8, 9], and Li [27, 28]. These metrics values are calculated for two programs (attached in the appendix) and the result is presented in Tables 2 and 3.

Metric	Description
Inheritance Metrics by Chidamber and Kemerer [12]	
Depth of Inheritance Tree (DIT)	Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.
Number of Children (NOC)	Number of immediate subclasses subordinated to a class in the class inheritance hierarchy is the the NOC for that class.
Inheritance Metrics by Abreu and Carapuca [2]	
Total Children Count (TCC)	Number of classes that inherit directly is the Total Children Count TCC of that class.
Total Progeny Count (TPC)	Number of classes that inherit directly or indirectly from a class is the Total Progeny Count (TPC) of that class.
Total Parent Count (TPAC)	The number of super classes from which a class inherits directly is the Total Parent Count (TPAC) of that class.
Total Ascendancy Count (TAC)	The number of super classes from which a class inherits directly or indirectly is the Total Ascendancy Count (TAC) of that class.
Total length of inheritance chain (TLI)	Total number of edges in the inheritance hierarchy graph.
Inheritance Metrics by Henderson-Sellers [22]	
Average Inheritance Depth (AID)	AID = Sum of depth of each class/Number of classes.
Inheritance Metrics by Lorenz and Kid [29]	
Number of Methods Inherited (NMI)	Number of methods inherited by a subclass.
Number of Methods Overridden (NMO)	Number of methods overridden by a subclass. superclass.
Number of New Methods (NNA)	Number of new methods in a subclass.
Inheritance Metrics by Li [27]	
Number of Ancestor Classes (NAC)	Total number of ancestor classes from which a class inherits is the NAC of that class.
Number of Descendent Classes (NDC)	Total number of Descendent classes (subclasses) of a class is the NDC of that class.

Table 1. Inheritance metrics

All metrics except TLI, AID and ACI are class level metrics. They may be used to determine the complexity of a class whereas TLI, AID and ACI can be used to determine the complexity of a program or module (consisting of many classes) as a whole.

5.1 Comparison of Class Level Metrics with Proposed Metric CCI

It is obvious that classes at lower level in the hierarchy in an inheritance tree are more complex because understanding these classes requires understanding, at least partially, parent's implementation as well as any of the parent's ancestors. In this case, DIT, TAC, NMI, NAC and the proposed class level metric CCI are more suitable to determine the complexity of a class since these metrics values are higher for lower level classes as given in Table 2. What makes the proposed metric different from DIT, TAC, and NAC is that values of DIT, TAC, and NAC for class Employee (program 1) and class Triangle (program 2) are the same as they are both inheriting from one class; but their parent classes are different in complexity and they are inheriting different methods in terms of number and complexity, so their values should be different. NMI and CCI values are different for class Employee and class Triangle; but NMI only considers the number of methods inherited by a class. It does not consider the complexity of the methods inherited. So, NMI value for two classes inheriting the same number of methods will be the same but their CCI values will be different if the complexities of the inherited methods are different. So, proposed metric CCI better represents the complexity of a class due to inheritance.

Pr.	Class	DIT	NOC	TCC	TPC	TPAC	TAC	NAC	NDC	NMI	NMO	NNA	CCI
1	Person	0	2	2	5	0	0	0	5	N.A.	N.A.	N.A.	3
	Employee	1	3	3	3	1	1	1	3	2	2	0	6
	Student	1	0	0	0	1	1	1	0	2	2	0	6
	Sal_Emp	2	0	0	0	1	2	2	0	4	2	0	10
	Comm_Emp	2	0	0	0	1	2	2	0	4	2	0	13
	Hour_Emp	2	0	0	0	1	2	2	0	4	2	0	14
2	Shape	0	2	2	3	0	0	0	3	N.A.	N.A.	N.A	10
	Triangle	1	1	1	1	1	1	1	1	7	1	1	14
	Rectangle	1	0	0	0	1	1	1	0	7	1	1	14
	NmTriangle	2	0	0	0	1	2	2	0	9	0	1	16

Table 2. Class level inheritance metrics values for program 1 and program 2

5.2 Comparison of program level metric AID with proposed metric ACI

According to Table 3, TLI, AID and the proposed metric ACI values are contradictory. TLI considers the numbers of classes inherited whereas AID just considers the average number of classes inherited in an inheritance tree. These metrics do not consider how complex the inherited classes are, how many methods are inherited or how complex the inherited methods are. As the number of classes inherited is more in program 1 than in program 2, TLI and AID values for program 1 are higher.

Although the number of classes inherited is less in program 2, the complexity of the inherited classes is higher in program 2. The inherited classes in program 2 have more methods than the inherited classes in program 1. So, the complexity, due to inheritance, of program 2 is higher than that of program 1 and therefore ACI better represents the complexity (due to inheritance) of a program.

Program	TLI	AID	ACI
1	5	1.33	8.67
2	3	1.00	13.5

Table 3. Program level inheritance metrics values for program 1 and program 2

5.3 Comparison with Respect to Weyuker's Properties

A good complexity measure should satisfy most of Weyuker's properties. Some of the object oriented inheritance metrics (DIT, NOC, TPC, TPAC, TAC) were already evaluated with Weyuker's properties. Other metrics (TCC, TLI, AID, NMI, NMO, NNA, NAC, NDC) are evaluated against all 9 Weyuker's properties during this study. The result is summarized in Table 4. It was found that none of the inheritance metrics except proposed metrics CCI and ACI satisfies Weyuker's property 7. Also, Weyuker's property 9 is only satisfied by ACI metric.

Property	DIT	NOC	TCC	TPC	TPAC	TAC	NAC	NDC	NMI	NMO	NNA	CCI	TLI	AID	ACI
1	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
2	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
3	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
5	×	√	√	×	√	×	×	×	×	√	√	√	×[1]	×[1]	√[1]
6	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
7	×	×	×	×	×	×	×	×	×	×	×	√	×	×	√
8	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
9	×	×	×	×	×	×	×	×	×	×	×	×	×	×	√

Table 4. Evaluation of inheritance metrics against Weyuker's properties

√ indicates that the metric satisfies the corresponding property.

× indicates that the metric does not satisfy the corresponding property.

[1] for calculations: individual P and Q inheritance depth is considered against $(P+Q)$ inheritance depth in 3 different cases: a) P and Q are siblings, b) one is the child of another, c) they are neither siblings not child to each other.

6 CONCLUSIONS

Two new object-oriented inheritance complexity metrics CCI (Class Complexity due to Inheritance) and ACI (Average Complexity of a program due to Inheritance) have been proposed in this paper. These proposed metrics are evaluated with Weyuker's properties and also compared with other well known object-oriented inheritance metrics. It has been found that the proposed metrics CCI and ACI better represent the complexity of a class and program due to inheritance.

It has been mentioned many times that Weyuker's property 7 is not applicable for object-oriented systems. Weyuker's property 7 states that changing the order of statements may affect the complexity of the program. Both proposed object-oriented inheritance complexity metrics CCI (Class Complexity due to Inheritance) and ACI (Average Complexity of a program due to Inheritance) satisfy Weyuker's property 7. This has also been illustrated by help of examples. It has been found that class level inheritance metric CCI satisfies all Weyuker's properties except property 9, whereas program level inheritance metric ACI satisfies all 9 of Weyuker's properties (including property 7). To the best of my knowledge, ACI (Average Complexity of a program due to Inheritance) is the only metric based on inheritance of an object-oriented program that satisfies all 9 Weyuker's properties.

Classes with higher CCI values are complex and therefore can be considered to be more error prone. So, testing efforts should be directed more towards these classes than classes with lower CCI values. Similarly, programs with higher ACI values have complex design so these programs will require more efforts during testing, specifically integration testing. As a future work, these metrics can be used for fault prediction in a quantitative way. It is also planned to develop a tool to calculate the proposed metrics.

7 APPENDICES

Program 1

```
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
//base class Person
class Person
{
char name[50],gen[6];
int age;
public:
Person(){}                //MC1 = 1
void get_info();
void put_info();
};
```

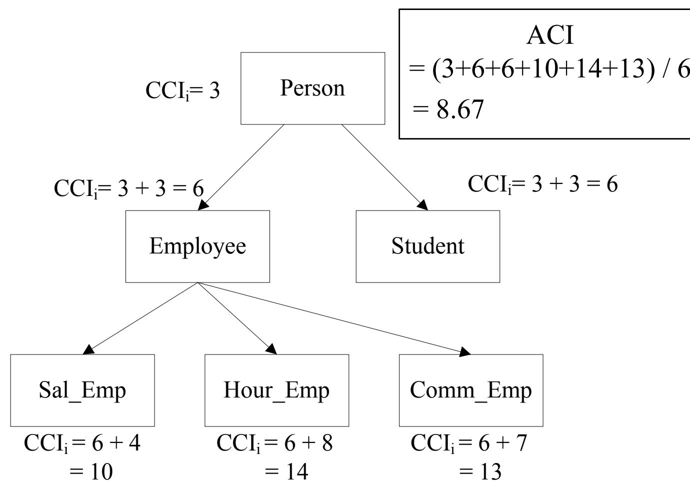

Fig. 1. Inheritance structure of program 1

```
void Person::get_info()
{
cout<<"Enter the name of person?";
gets(name);
cout<<"Enter the gender of person?";  //MC2 = 1
cin>>gen;
cout<<"Enter the age of person?";
cin>>age;
}
void Person::put_info()
{
cout<<name<<"\t";
cout<<gen<<"\t";                //MC3 = 1
cout<<age<<"\t";
}
//*******************************************
// employee class derived from person
class Employee:public Person
{
int empid;
char des[20];
public:
Employee(){}                     //MC1 = 1
void get_info();
void put_info();
```

```
};
void Employee::get_info()
{
Person::get_info();
cout<<"Enter emp id?";     //MC2 = 1
cin>>empid;
cout<<"Enter designation?";
gets(des);
}
void Employee::put_info()
{
Person::put_info();
cout<<empid<<"\t";       //MC3 = 1
cout<<des<<"\n";
}
//*****************************************
//Salary_Emp class derived from Employee
class Sal_Emp: public Employee
{
double salary;
public:
Sal_Emp(){}                   //MC1 = 1
void get_info();
void put_info();
};
void Sal_Emp::get_info()
{
Employee::get_info();
cout<<"Enter salary";      //MC2 = 2
cin>>salary;
if (salary < 0.0)
salary = 0.0;
}
void Sal_Emp::put_info()
{
Employee::put_info();        //MC3 = 1
cout<<salary<<"\n";
}
//******************************************
//Hourly_Emp class derived from Employee
class Hour_Emp: public Employee
{
int hoursWorked;
double wage;
```

```
double earning;
public:
Hour_Emp(){}                //MC1 = 1
void get_info();
void put_info();
};
void Hour_Emp:: get_info()
{
Employee::get_info();
cout<<"Enter number of hours worked";
cin>>hoursWorked;
cout<<"Enter hourly wage";
cin>>wage;
if (hoursWorked >= 0.0)
{
if (hoursWorked <= 168.0)
hoursWorked = hoursWorked;
else
hoursWorked = 0.0;
}
else
hoursWorked = 0.0;
if (wage < 0.0)                          //MC2 = P+D+1
wage = 0.0;                              // = 4+1+1= 6
if (hoursWorked <= 40.0)
earning = hoursWorked*wage;
else
earning =(40*wage)+((hoursWorked-40)*wage*1.5);
}
void Hour_Emp::put_info()
{
Employee::put_info();
cout<<hoursWorked<<"\t";       //MC3 = 1
cout<<wage<<"\t";
cout<<earning<<"\n";
}
//****************************************
//Commission_Emp class derived from Employee
class Comm_Emp: public Employee
{
int sale;
double commRate;
double earning;
public:
```

```
Comm_Emp(){}                        //MC1 = 1
void get_info();
void put_info();
};
void Comm_Emp::get_info()
{
Employee::get_info();
cout<<"Enter gross sale amount";
cin>>sale;
cout<<"Enter commission rate";
cin>>commRate;
if (sale < 0.0)
sale = 0.0;
if (commRate > 0.0)                 //MC2 = P+D+1
{                                   //    = 3+1+1=5
if (commRate < 1.0)
commRate = commRate;
else
commRate = 0.0;
}
else
commRate = 0.0;
earning = sale*commRate;
}
void Comm_Emp::put_info()
{
Employee::put_info();               //MC3 = 1
cout<<sale<<"\t";
cout<<CommRate<<"\t";
cout<<earning<<"\n";
}
//*******************************************
//Student class derived from Person
class Student:public Person
{
int studid;
char class_name[10];
public:
Student() {}                       //MC1 = 1
void get_info();
void put_info();
};
void Student::get_info()
{
```

```
Person::get_info();
cout<<"Enter stud id?";
cin>>studid;
cout<<"enter student class name?";    //MC2 = 1
gets(class_name);
}
void Student::put_info()
{
Person::put_info();
cout<<studid<<"\t";                      //MC3 = 1
cout<<class_name<<"\n";
}
void main()
{
clrscr();
Employee e;
cout<<"\n\nENTER EMPLOYEE INFORMATION\n\n";
e.get_info();
Student p;
cout<<"\n\nENTER STUDENT INFORMATION \n\n";
p.get_info();
cout<<"\n NAME\tGENDER\tAGE\tEMPID\tSALARY\tDESIGNATION\n";
e.put_info();
cout<<"\n NAME\tGENDER\tAGE\tSTUDID\tCLASS NAME\n";
p.put_info();
}
```

Program 2

```
#include <iostream>
#include <cstring>
using namespace std;
// base class Shape
class Shape {
double width;
double height;
char name[20];
public:
Shape() {
width = height = 0.0;                   //MC1 = 1
strcpy(name, "unknown");
}
Shape(double w, double h, char *n) {
width = w;
```

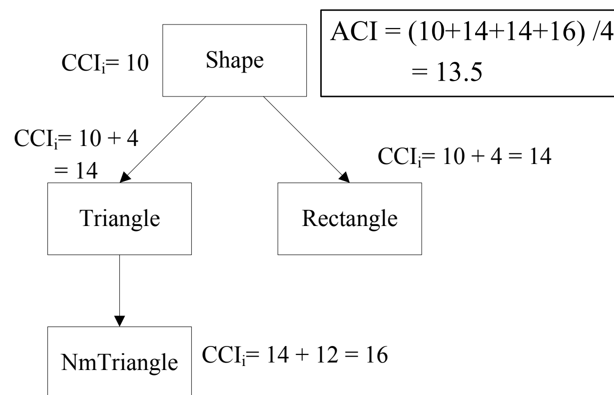

Fig. 2. Inheritance structure of program 2

```
height = h;                              //MC2 = 1
strcpy(name, n);
}
Shape(double x, char *n) {
width = height = x;               //MC3 = 1
strcpy(name, n);
}
void display() {
cout << "Width and height are " << width << " and "
          << height << "\n";               //MC4 = 1
}
double getWidth() { return width; }        //MC5 = 1
double getHeight() { return height; }     //MC6 = 1
void setWidth(double w) { width = w; }    //MC7 = 1
void setHeight(double h) { height = h; }    //MC8 = 1
char *getName() { return name; }           //MC9 = 1
virtual double area() {
cout << "Error: area() must be overridden.\n";
return 0.0;                                       //MC10 = 1
}
};
//*******************************************
//class Triangle derived from Shape
class Triangle : public Shape {
char style[20];
public:
Triangle(char *str, double w, double h) : Shape(w, h, "triangle") {
strcpy(style, str);                            //MC1 = 1
```

```
}
Triangle(double x) : Shape(x, "triangle") {
strcpy(style, "isosceles");                  //MC2 = 1
}
double area() {
return getWidth() * getHeight() / 2;         //MC3 = 1
}
void showStyle() {
cout << "Triangle is " << style << "\n";       //MC4 = 1
}
};
//****************************************
// class NameTriangle derived from Triangle
class NmTriangle : public Triangle {
char name[20];
public:
NmTriangle(char *clr, char *style, double w, double h) :
Triangle(style, w, h) {
strcpy(name, clr);                          //MC1 = 1
}
void displayName() {
cout << "Name is " << name << "\n";        //MC2 = 1
}
};
//****************************************
//class Rectangle derived from Shape
class Rectangle : public Shape {
public:
Rectangle(double w, double h) : Shape(w, h, "rectangle") { }//MC1 = 1
Rectangle(double x) : Shape(x, "rectangle") { }       //MC2 = 1
void displayName() {
cout << "Name is " << getName() << "\n";
}                                                     //MC3 = 1
double area() {
return getWidth() * getHeight();                    //MC4 = 1
}
};
int main() {
Shape *shapes[5];
shapes[0] = &Triangle("right", 8.0, 12.0);
shapes[1] = &Rectangle(10);
shapes[2] = &Rectangle(10, 4);
shapes[3] = &Triangle(7.0);
shapes[4] = &Shape(10, 20, "generic");
```

```
for(int i=0; i < 5; i++) {
cout << "object is " << shapes[i]->getName() << "\n";
cout << "Area is " << shapes[i]->area() << "\n\n";
}
NmTriangle t1("A", "right", 8.0, 12.0);
NmTriangle t2("B", "isosceles", 2.0, 2.0);
t1.showStyle();
t1.display();
t1.displayName();
cout << "Area is " << t1.area() << "\n";
t2.showStyle();
t2.display();
t2.displayName();
cout << "Area is " << t2.area() << "\n";
return 0;
}
```

REFERENCES

[1] Abreu, F. B.—Melo, W.: Evaluating the Impact of Object-Oriented Design on Software Quality. In METRICS '96: Proceedings of the 3rd International Symposium on Software Metrics, p. 90, IEEE Computer Society Washington, DC, USA 1996.

[2] Abreu, F. B.—Carapuca, R.: Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Journal of System Software, Vol. 26, 1994, pp. 87–96.

[3] Arisholm, E.—Briand, L. C.—Foyen, A.: Dynamic Coupling Measurement for Object-Oriented Software. IEEE Trans. Softw. Eng., Vol. 30, 2004, No. 8, pp. 491–506.

[4] Basili, V. R.:Viewing Maintenance As Reuse Oriented Software Development. IEEE Software, Vol. 7, 1990, No. 1, pp. 19–25.

[5] Basili, V. R.—Briand, L. C.—Melo, L. W.: A Validation of Object-Oriented Design Metrics As Quality Indicators. IEEE Transactions on Software Engineering, Vol. 22, 1996, No. 10, pp. 751–761.

[6] Basili, V. R.—Briand, L. C.—Melo, W. L.: How Reuse Influences Productivity in Object-Oriented System. Commun. ACM, Vol. 39, 1996, No. 10, pp. 104–116.

[7] Briand, L.—Bunse, L.—Daly, J.—Differding, C.: An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. In: Proceedings of Empirical Assessment in Software Engineering (EASE), Keele, UK 1997.

[8] Briand, L. C.—Daly, J. W.—Porter, V.—Wst, J.: A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems. Technical Report ISERN-98-07, 1998. available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.152.

[9] Briand, L. C.—Wst, J.—Daly, J. W.—Porter, D. V.: Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. Journal of Systems and Software, Vol. 51, 2000, No. 3, pp. 245–273.

[10] Cartwright, M.—Shepperd, M.: An Empirical Analysis of Object Oriented Software in Industry. In: Bournemouth Metrics Workshop, April, Bournemouth, UK 1996.

[11] Cherniavsky, J.—Smith, C.: OnWeyukers Axioms for Software Complexity Measures. IEEE Transaction on Software Engineering, Vol. 17, 1991, No. 6, pp. 636–638.

[12] Chidamber, S. R.—Kemerer, C. F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Vol. 20, 1994, No. 6, pp. 476–493.

[13] Chidamber, S. R.—Darcy, D. P.—Kemerer, C. F.: Managerial Use of Metrics for Objectoriented Software: An Exploratory Analysis. IEEE Trans. Softw. Eng., Vol. 24, 1998, No. 8, pp. 629–639.

[14] Dagpinar, M.—Jahnke, J. H.: Predicting Maintainability with Object-Oriented Metrics – An Empirical Comparison. In Proceedings of the 10th Working Conference on Reverse Engineering (November 13–17, 2003), WCRE. IEEE Computer Society, Washington, DC, p. 155.

[15] Darcy, D. P.—Kemerer, C. F.: OO Metrics in Practice. IEEE Softw. 22, 6 November 2005, pp. 17–19. DOI: `http://dx.doi.org/10.1109/MS.2005`.

[16] El Emam, K.: A Primer on Object-Oriented Measurement. In Proceedings of the 7th international Symposium on Software Metrics (April 04–06, 2001), METRICS. IEEE Computer Society, Washington, DC, p. 185.

[17] El Emam, K.—Melo, W.—Machado, J. C.: The Prediction of Faulty Classes Using Object-Oriented Design Metrics. J. Syst. Softw., Vol. 56, 2001, No. 1, pp. 63–75.

[18] Fenton, N.: Software Measurement: A Necessary Scientific Basis. IEEE Trans. on Software Engg., Vol. 20, 1994, No. 6, pp. 199–206.

[19] Gursaran, G. R.: On the Applicability of Weyuker Property Nine to Object-Oriented Structural Inheritance Complexity Metrics. IEEE Transaction on Software Engineering, Vol. 27, 2001, No. 4, pp. 361–364.

[20] Harrison, R.–Counsell, S. J.: An Evaluation of the Mood Set of Object-Oriented Software Metrics. IEEE Transactions on Software Engineering, Vol. 21, 1995, No. 12, pp. 929–944.

[21] Harrison, R.—Counsell, S.—Nithi, R.: Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. Journal of Systems and Software, Vol. 52, 2000, pp. 173–179.

[22] Henderson-Sellers, B.: Object Oriented Metrics: Measures of Complexity. Prentice Hall PTR: Englewood Cliffs, NJ, 1996; pp. 130–132.

[23] Humphrey, W. S.: Managing the Software Process. SEI Series in Software Engineering, Addison-Wesley Publishing Company 1989.

[24] Kitchenham, B.—Pfleeger, S. L.—Fenton, N.: Towards a Framework for Software Measurement Validation. IEEE Trans. on Software Engg., Vol. 21, 1995, No. 12, pp. 929-944.

[25] Lattanzi, M.—Henry, S.: Software Reuse Using C++ Classes. The Question of Inheritance Journal of Systems and Software, Vol. 41, 1998, pp. 127–132.

[26] Lavazza, L.—Denaro, G.—Pezze, M.: An Empirical Evaluation of Object Oriented Metrics in Industrial Setting. In The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal, November 2003.

[27] Li, W.: Another Metric Suite for Object-Oriented Programming. Journal of Systems and Software, Vol. 44, 1998, pp. 155–162.

[28] Li, W.—Henry, S.: Object-Oriented Metrics That Predict Maintainability. Journal of Systems and Software, Vol. 23, 1994, No. 2, pp. 111–122.

[29] Lorenz, M.—Kidd, J.: Object-Oriented Software Metrics. Prentice Hall 1994, ISBN: 013179292X.

[30] Martin, R. C.: Agile Software Development: Principles, patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA 2002.

[31] McCabe, T. J.: A Complexity Measure. IEEE Transactions on Software Engineering, Vol. SE-2, 1976, No. 4, pp. 308–320,

[32] Melton, A.—Gustafson, D.—Bieman, J.—Baker, A.: A Mathematical Perspective for Software Measure Research. Journal of Software Eng., Vol. 5, 1990, No. 5, pp. 246–254.

[33] Mishra, D.—Mishra, A.: An Efficient Software Review Process for Small & Medium Enterprises. IET Software, Vol. 1, 2007, No. 4, pp. 132–142.

[34] Mishra, D.—Mishra, A.: Simplified Software Inspection in Compliance with International Standards. Computer Standards and Interfaces, Vol. 31, No. 4, pp. 763–771.

[35] Pason, D.: Object-Oriented Programming. DP Publication 1994.

[36] Piwowarski, P.: A Nesting Level Complexity Measure. SIGPLAN Notices, Vol. 17, 1982, No. 9, pp. 44–50.

[37] Prather, R. E.: An Axiomatic Theory of Software Complexity Measurement. Computing Journal, Vol. 27, 1984, No. 4, pp. 340–346.

[38] Prechelt, L.—Unger, B.—Philippsen, M.—Tichy, W.: A Controlled Experiment on Inheritance Depth As a Cost Factor for Code Maintenance. Journal Syst. Softw., Vol. 65, 2003, No. 2, pp. 115–126.

[39] Roy, G.: On the Applicability of Weyuker Property Nine to Object-Oriented Structural Inheritance Complexity Metrics. M. Tech. Minor Project Report, Faculty of Eng., Dayalbagh Educational Inst., Agra 1997.

[40] Sharma, N.—Joshi, P.—Joshi, R. K.: Applicability of Weyuker's Property 9 to Object-Oriented Metrics. IEEE Transaction on Software Engineering, Vol. 32, 2006, No. 3, pp. 209–211.

[41] Sheldon, F. T.—Jerath, K.—Chung, H.: Metrics for Maintainability of Class Inheritance Hierarchies. Journal of Software Maintenance 14, 3 May 2002, pp. 147–160.

[42] Shepperd, M.: Products, Processes and Metrics. Information and Software Technology, Vol. 34, 1992, No. 10, pp. 674–680.

[43] Walsh, J. F.: Preliminary Defect Data from the Iterative Development of a Large C++ program (Experience Report). In Proc. of OOPSLA92, pp. 178–183, 1992.

[44] Weyuker, E. J.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, Vol. 14, 1988, No. 9, pp. 1357–1365.

[45] Zhang, L.—Xie, D.: Comments on On the applicability of Weyuker Property Nine to Object-Oriented Structural Inheritance Complexity Metrics. IEEE Transactions on Software Engineering, Vol. 28, 2002, No. 5, pp. 526–527.

Deepti Mishra is an Assistant Professor of Computer Engineering at Atilim University, Ankara, Turkey. She has received her Ph. D. in Computer Science (Software Engineering) and Masters in Computer Science and Applications. Her research interests include software process improvement, software quality, requirement engineering and databases. She has published many research papers and book chapters at international and national levels. She has been granted the Department of Information Technology Scholarship of Government of India.