Weighted Class Complexity: A Measure of Complexity for Object Oriented System

JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 24, 1689-1708 (2008)

Weighted Class Complexity: A Measure of Complexity for Object Oriented System

SANJAY MISRA AND K. IBRAHIM AKMAN
Department of Computer Engineering
Atilim University
Ankara, Turkey

Software complexity metrics are used to predict critical information about reliability and maintainability of software systems. Object oriented software development requires a different approach to software complexity metrics. In this paper, we propose a metric to compute the structural and cognitive complexity of class by associating a weight to the class, called as Weighted Class Complexity (WCC). On the contrary, of the other metrics used for object oriented systems, proposed metric calculates the complexity of a class due to methods and attributes in terms of cognitive weight. The proposed metric has been demonstrated with OO examples. The theoretical and practical evaluations based on the information theory have shown that the proposed metric is on ratio scale and satisfies most of the parameters required by the measurement theory.

Keywords: software metrics, object oriented programming, class complexity, cognitive complexity, cognitive weights, validation criteria

1. INTRODUCTION

Software engineers generally use indirect measures that lead to metrics which provide a quantitative basis for understanding the underlying information in software development processes. Software metrics have always been important for software developers to assure the quality of some representation of software, and organizations are achieving promising results through their use. The quality of software must be defined in terms that are meaningful to the users. Generally, quality objectives [1] may be listed as performance, reliability, availability and maintainability, which are closely related to software complexity.

Object Oriented (OO) techniques have come to dominate software engineering over the last two decades. The improvement and modification in these techniques are still undergoing research [2-4]. More and more organizations are adopting these techniques into their software development practices. A result of the growth in popularity of object oriented programming is the introduction of number of software design metrics. Object-oriented design complexity metrics are used to predict critical information about reliability and maintainability [5] of software systems and therefore help to evaluate and improve the quality of the design. Today, the relevant literature provides a variety of object oriented metrics [6-13], to compute the complexity of software. Further, selecting a particular metric is again a problem, as every metric has its own advantages and disadvantages. There is continuous effort to find a comprehensive complexity metric, which addresses most of the parameters of software.

Received February 27, 2007; revised August 28 & December 26, 2007 & May 22, 2008; accepted June 12, 2008.
Communicated by Jonathan Lee.

OO software development is based on classes, subclasses and objects whose elements are attributes, methods and messages. These elements are identified in class declarations and are responsible for the complexity of a class. Abbot [7] defines the complexity to be a function of the interactions of its set of properties and in the case of objects and classes, the methods and data attributes are the set of properties. Therefore, complexity of a class is a function of the methods and the data attributes. 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 method have not yet been studied carefully. Further, at the class level, most of the metrics do not consider the internal architecture of methods, messages and attributes together.

On the other hand, most of the object oriented metrics do not consider the cognitive characteristics in calculating the complexity of code, which directly affects the cognitive complexity. Complexity of the method directly affects understandability of the code. Understandability of the code is known as program comprehension and is a cognitive process and related to cognitive 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. In our proposal, we calculated cognitive complexity in terms of cognitive weights [14]. Cognitive weights 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, cognitive complexity also provides valuable information for the design of object oriented systems. High cognitive complexity indicates poor design, which sometimes can be unmanageable [6]. In such cases, maintenance effort increases drastically.

Earlier, complexity of the method could only be calculated by applying conventional metrics, but there are several criticisms [16, 17] of these metrics. These criticisms are mainly based on a lack of theoretical basis [18], lacking in desirable measurement principles [19] and mathematical [20] properties, being insufficiently generalized or too implementation technology dependent [16]. On the other hand, the traditional metrics only investigate the complexity of operation in a method and do not provide a proper way to calculate the complexity of class and the entire code. It was also the case in our previous [15] work.

Against this backdrop, the present work proposes a new metric on class level of OO systems. The proposed metric computes the structural and cognitive complexity of class by associating a weight to the class, called as Weighted Class Complexity (WCC). The proposed metric includes the complexity of operations and messages in a method in terms of cognitive weights. It also considers the complexity due to data members (attributes). The data members are equally important in design consideration. To the best of our knowledge, none of the object oriented metrics constructed up to the present date to calculate the total complexity of a class by considering the complexity due to the internal architecture of the code (methods and message), and attributes. In addition, the proposed metric also consider the structure and cognitive aspects of the code, since the structural complexity is defined as [21] “the organization of program elements within program.” The cognitive weights are used to measure the complexity of the logical structures of the software. Further, all of 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 software development process. Therefore; we considered all these issues in our proposal. In our previous work, [15] we presented a metric (in ICCI'2006) based on cognitive weights at method level. It is similar to calculate the complexity of procedural language, because it only considers the complexity of operation inside the method. In the present paper, we extended our metric [15] at class level. Cognition of a class requires a comprehension of class, which is not only due to methods but also due to attributes.

The proposal of WCC is given in next section. We evaluated and validated our measure through Weyuker's properties and measurement theory in section 3. Section 4 includes the practical evaluation of measure through a framework. Empirical validations through a case study and comparative study with similar measures have also been done in this section. The summary and future work is given in section 5. The last section includes the conclusion drawn.

2. PROPOSED METRIC: WEIGHTED CLASS COMPLEXITY (WCC)

Complexity [22] is defined as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" *i.e.* complexity of a code is directly dependent on the understandability. All the factors that makes program difficult to understand are responsible for complexity. Object orientation is a form of expression relation between the data and function and the class can be assumed as a set of data and set of method accessing them. Hence, the complexity of the class should be measured by complexity of methods and attributes. In our proposed measure, the complexity of a class is the sum of complexity of the operation in methods, complexity due to data members (attributes) and complexity due to message call. Further, complexity of method is calculated by complexity of the code of operation in method and as well as on the number of messages in the method. Now, we can make the relation more clear and introduce the WCC.

The proposed metric is first interested in calculating the complexity of operations by considering corresponding cognitive weights. The cognitive weights are used to measure the complexity of the logical structures of the software. These logical structures reside in the method (code) and are classified as sequence, branch, iteration and call (message in OO). The corresponding weights of these basic control structures are one, two, three and two [14]. Actually, the weights are assigned on the classification of cognitive phenomenon as discussed by Wang [14]. He proved and assigned the weights for sub conscious function, meta cognitive function and higher cognitive function as 1, 2 and 3 respectively.

The second and third stages of the proposed metric calculate the complexity of each class and find the complexity of the entire code respectively.

Accordingly, we first calculate the weight of individual method in a class by associating a number (weight) with each member function (method), and then we simply add all the weights of all methods. This gives the complexity (weight) due to methods. The total weight of a single method, called method complexity (MC) is defined as the sum of cognitive weights of its *q* linear blocks composed in individual BCS's. Since each block may consists of *m* layers of nested BCS's, and each layer with *n* linear BCS's, the total cognitive weight of a method can be calculated by:

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

where W_c is the cognitive weight of the concerned Basic Control Structure (BCS).

If there are *s* methods in a class, then complexity due to all methods of the class is given by total method complexity

$$= \sum_{p=1}^{s} MC_p. \tag{2}$$

Further we count the total number of attributes in that class. It reflects the complexity due to data members (attributes). In other words, the complexity due to data members for a class equals to total number of data members in that class. The attributes are not local to one procedure but local to objects and can be accessed by several procedures. We represent the complexity due to attributes is N_a.

Using above consideration, we suggest a formula for calculating the complexity of a single class, called Weighted Class Complexity (WCC),

$$WCC = N_a + \sum_{p=1}^{s} MC_p. \tag{3}$$

If there are *y* classes in an object oriented code, then the total complexity of the code is given by the sum of weights of individual classes.

$$TotalWeightedClassComplexity = \sum_{x=1}^{y} WCC_x \tag{4}$$

The unit of WCC is defined as the cognitive weight of the simplest software component *i.e.* only a linear structure BCS is taken as one Weighted Class Complexity unit (WCCU).

It is important to note that this approach includes the complexity of the class due to messages, automatically. In the case of a message between two classes, the complexity of the message is the sum of the weight of the called procedure and the weight due to that call (*i.e.* two). Messages between the classes are the indication of coupling. Although, WCC is not a measure of coupling and cohesion; it provides some indication for the level of coupling. If the number of messages between the classes increases, the overall complexity increases. Clearly, a high complexity value represents the high coupling between classes because of greater number of messages, which is undesirable according to quality design principles [23].

The above measure has been illustrated with the help of an example in section 4.

3. THEORETICAL EVALUATION OF PROPOSED MEASURE

Any new measure must be validated and evaluated both formally and practically. Further, the newly proposed measure is acceptable only when its usefulness has been proved by a validation process. The purpose of the validation is also to prove the usefulness of attribute, which is measured by the proposed metric. For this purpose, in section 3.1, we examined our proposed metric against the nine well known Weyuker's properties [19]. Although, these properties are very much criticized by several researchers, they are still in use and the topic of research [24, 25]. Further, the measurement process is known to be critical in both science and engineering. In order to make the software discipline more and more mature we can use the tools provided by Measurement Theory (MT). As a consequence, a proposal of new software metric can be validated through the application of MT basics. In section 3.2, we define the basics of MT and look at the proposed metric from a theoretical measurement perspective.

3.1 WCC and Weyuker's Properties

Weyuker [19] properties are well established evaluation criterion for software complexity measures. A good complexity measure should satisfy the Weyuker's properties. Although Weyuker proposed the properties at the time when procedural languages were dominant, these properties are also valuable to evaluate object-oriented metrics at present. A significant number [10, 11] of researchers evaluated the object-oriented metrics by the complexity properties proposed by Weyuker [19]. We also used these properties to evaluate our proposed measure.

Property 1: $(\exists P)(\exists Q)(|P| \neq |Q|)$. Where P and Q are the two different classes.

This property states that a measure should not rank all classes as equally complex. Now consider two examples given in Appendixes 2 and 3. The WCC values for these programs are 19 and 18 respectively. Therefore, this property is satisfied by WCC.

Property 2: Let c be a non-negative number, and then there are only finite number of classes and programs of complexity c. In other words, it states that there is only finite number of classes of the same weight. All object-oriented languages consist only finite number of cognitive weights of basic control structures and attributes. Since, the total complexity is defined as the sum of cognitive weights of all the methods and attributes in a program, a possible largest number can be assumed, without harm, to be the upper bound. Therefore, for given a given number of complexity value, there are only finitely many programs. Hence, this proposed complexity does hold this property.

Property 3: There are distinct class P and Q such that $|P| = |Q|$.

This property states that there are multiple classes of the same complexity.

Consider two programs given in Appendixes 2 and 4. WCC values for both these programs are 19. Therefore, WCC also satisfies this property.

Property 4: $(\exists P)(\exists Q)(P \equiv Q$ and $|P| \neq |Q|)$.

If there exists classes P and Q such that they produce, the same output given the same input. In other words, this property states that even if two classes have same functionality, they are different in details of implementation, and means that even if two programs, which consist of many classes, have the same functionality, they are different in the details of implementations. Since the cognitive weights depend on the internal architecture, therefore cognitive weights for the two classes for the same output may be different. Therefore, this property is also satisfied by the given measure.

Property 5: $(\forall P)(\forall Q)(|P| \le |P; Q|$ and $|Q| \le |P; Q|)$.

Since the class complexity is given by, the associated cognitive weight, which is an integer, and the set of integers with operator holds the following property,

$(\forall P)(\forall Q)(P \le P + Q)$ and $(Q \le P + Q)$.

Since this equation and Weyuker's property 5 are analogous and such property 5 is satisfied by the proposed measure.

We can also prove this property by taking the examples from appendix. The code given in Appendixes 2-5, are the sub components of code given in Appendix 1. The WCC values of these codes are 19, 18, 19 and 22, (see Table 1) which all are less than WCC value (*i.e.* 30) of code given in Appendix 1. Therefore, this property is also satisfied.

Table 1. Calculated WCC values for different OO codes (given in Appendixes 1-5).

Appendix	1	2	3	4	5
Name of class	PESON-STUDENT-EMPLOYEE-FACULTY-ADMINSTRATIVE	PERSON-STUDENT	PERSON-EMPLOYEE-FACULTY	PERSON-EMPLOYEE-ADMISTRATIVE	PERSON-EMPLOYEE-FACULTY-ADMINSTRATIVE
WCC	30	19	18	19	22

Property 6a: $(\exists P)(\exists Q)(\exists R)(|P| = |Q|)$ and $|P; R| \ne |Q; R|)$.

6b: $(\exists P)(\exists Q)(\exists R)(|P| = |Q|)$ and $|R; P| \ne |R; Q|)$.

This property shows the non-equivalence of interaction. It means that if a new class is appended to two classes which have the same class complexity, the class complexities of two new combined classes are different or the interaction between P and R can be different than interaction between Q and R resulting in different complexity values for $P + R$ and $Q + R$.

The cognitive weights of methods and number of attributes are fixed for a program. Therefore joining program R with P and Q adds same amount of complexity hence property 6 is not satisfied by this measure.

Property 7: There are program bodies P and Q such that Q is formed by permuting the order of the statements of P, and $(|P| \ne |Q|)$.

This property states that permutation of elements within the item being measured can change the metric values. The intent is to ensure that metric values change due to permutation of statements within the method under classes. However, in case of object-oriented programming in any class, changing the order in which methods or attribute is declared or even in operation the change in order of statement, do not affect the order in which they are executed. Therefore, our proposed measure is not satisfied by this property.

Property 8: If P is renaming of Q, then $|P| = Q|$.

This property requires that when the name of the class changes it will not affect the complexity of the class. Even if the member function or member data name in the class change, the class complexity should remain unchanged. As in our proposed complexity measure, there is no effect in complexity by renaming, so this property is also satisfied.

Property 9: $(\exists P)(\exists Q)(|P| + |Q|) < (|P; Q|)$.

This property shows interaction increases complexity. This property states that the class complexity of a new class combined from two classes is greater than the sum of two individual class complexities. For our case, this property is also not satisfied. For example, if we take two classes, PERSON ($WCC = 11$) and STUDENT ($WCC = 8$), when combine in a single class (Appendix 2) PERSON-STUDENT, their WCC values simply added (*i.e.* 19). Therefore, this complexity measure is not satisfied by the property nine.

It has been shown in this section that WCC satisfies six of Weyuker's nine properties. However, satisfying Weyuker's properties is a necessary but not sufficient condition for a good complexity measure [5]. Therefore, in the next section, we also evaluate WCC against the principle of measurement theory.

3.2 WCC and Measurement Theory

The relation between measurement theory and evaluation criteria for software complexity measure is well established by several researchers. A number of researchers proposed different criteria [27-32] to which the proposed software metric should adhere. However, in general all of the aforementioned criteria suggest that the metric should fulfill some basic requirements based on measurement theory perspective. Amongst available validation criteria, the framework given by Briand *et al.* [27] is reported to be the more practical and used by several researchers [9]. 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 applying our proposed measure against this framework, it seems appropriate to provide the basic definitions and the desirable properties for complexity measures given in the framework [27].

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.

For our proposed complexity measure, the entities are classes, *i.e.* E be a set of classes in S, the binary relation on classes is chosen to be greater than or equally complex.

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

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.* [27] and corresponding evaluation of the proposed metric are given below:

Property complexity 1 Nonnegative: The complexity of a system $S = <E, R>$ is non-negative if Complexity(S) ≥ 0.

Proof: Since our measure is obtained by the sum of weights of non-negative number 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 and attribute are present in the system, the complexity value in terms of weight is trivially null and therefore this property is satisfied by the proposed measure.

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.

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

Proof: In the proposed measure, 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 depends on the order or way of representation. Therefore, this property is satisfied by the proposed measure.

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.

$$(S = <E, R> \text{ and } m_1 = <E_{m1}, R_{m1}> \text{ and } m_2 = <E_{m2}, R_{m2}> \text{ and } m_1 \cup m_2 \subseteq S \text{ and } R_{m1} \cap R_{m2} = \varnothing) \Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2)$$

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 our case 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. We can easily prove this theorem by taking the example given in Appendix 1. In the first example**,** If the class PERSON-STUDENT-EMPLOYEE-FACULTY-ADMINSTRATIVE (Appendix 1) is partitioned into five sub classes PERSON, STUDENT, EMPLOYEE, FACULTY, ADMINSTRATIVE, (see Table 2), then it can easily observed that the complexity of the class PERSON-STUDENT-EMPLOYEE-

Table 2. Calculated WCC values for subclasses (code in Appendix 1).

Name of Class	STUDENT	ADMINSTRATIVE	FACULTY	EMPLOYEE	PERSON
WCC	8	4	3	4	11

FACULTY-ADMINSTRATIVE (*i.e.* 30) is the sum of complexity of their components. 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.

$(S = <E, R>$ and $S = m_1 \cup m_2$ and $m_1 \cap m_2 = \emptyset)$
$\Rightarrow$ Complexity(S) = Complexity(m_1) + Complexity(m_2)

Proof: For our case, we can say that the complexity value of class which is obtained by concatenation of m_1 and m_2 is equal to the sum of their calculated 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 proved by the proposed metric. We can also prove this theorem by taking the same example given in the proof of previous property. In other words, we can say that this property is also satisfied by our complexity measure since the WCC for code in Appendix 1 is 30, which is sum of complexity values for their components classes PERSON(11), STUDENT(8), EMPLOYEE (4), FACULTY(3), ADMINSTRATIVE(4).

As consequences of the above properties Complexity 1 to Complexity 5, it is shown that adding relationships between elements of a system does not decrease its complexity. Furthermore, our proposed complexity metric hold properties Complexity 1 to Complexity 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, the most desirable property for a complexity measure.

4. PRACTICAL EVALUATIONS OF PROPOSED MEASURE

Practical success of any proposed metric depends on the establishment of (1) its validation, (2) understandability by its users and (3) tight link between the metric and the attribute that it is intended to measure. In the previous section, we showed that our measure is a valid measure of complexity. An alternative approach to metric validation, which is more practical than the formal approach, is given by Kaner [33]. We follow this approach for practical evaluation. When we look our measure from the perspective given in [33], it is an indirect metric. It is a function of two components, which contributes to the measurement of software complexity. It should be clear that the propose measure is neither complete nor unique measure of complexity.

4.1 Evaluation through a Framework

For practical evaluation of our measure, we will apply the metric evaluation framework developed by [33]. The framework is based on the following points:

The purpose of the measure: Two main purposes of the measure are to contribute the judgment about design and product quality.

Scope of usage of the measure: The proposed measure can be categorized as an object oriented design complexity metric. Consequently, its scope of use is the software development group. It can be used to predict the maintenance effort.

Identified attribute to measure: The attributes measured by our metric are the quality of the product and the developer. More complex product makes it less understandable and consequently less maintainable for future development effort. In addition, the developer who can satisfy the user requirements through the usage of less number of branching and looping primitives (implying small time-complexity) is assumed more skillful.

Natural scale of the attribute: The existence of natural scale for the above attributes (but not the metrics) requires the development of a common, non-subjective view about them. We have no knowledge about the natural scale of attributes.

Natural variability of the attribute: If an attribute involves human performance then we can talk about its variability. The reason behind it; although one can develop a sound approach to handle such attribute it may not be complete because of the existence of many other factors that affects the attribute's variability. The difficulty of making sound and complete empirical observations about the product results in no knowledge about the variability of the attribute.

Definition of metric: The metric has been defined formally in section 2.

Measuring instrument to perform the measurement: It uses the instrument of *counting* by either human or by machine. The items to be counted are cognitive weights of different BCS's and attributes. For automated counting purpose, one can easily develop a token generator and use the string matching algorithms.

Natural scale for the metric: For the natural scale for the measure, we have to go through measurement theory. When we analyze our measure according to Morasca [27] it is on ratio scale.

The natural variability of readings from the instrument: Since the reading from our counting instrument is not subjective and does not require any interpretation, we can say that no variability (*i.e.* measurement error) on readings from the instrument can be expected. Note that, in case of automated counting, we assume that there is no bug in the devised algorithm.

Relationship between the attribute to the metric value: There is a direct relation between the quality of the product and our proposed measure. If the complexity value increases, the product quality will decrease, since it implies inefficient use of memory and time. Note that proposed measure is not the unique indicator of product quality. The same argument is also true for the relation between the complexity value and the developer quality attribute.

Natural and foreseeable side effects of using the instrument: Once we automate the complexity calculation, it will not require considerable additional workload of manpower of the company. The only cost will be due to automation.

4.2 Experimentation and a Case Study

The proposed complexity metric given by Eq. (4) is demonstrated with the programming example given by the following Fig. 1. The complete code for the following figure is given in Appendix 1.

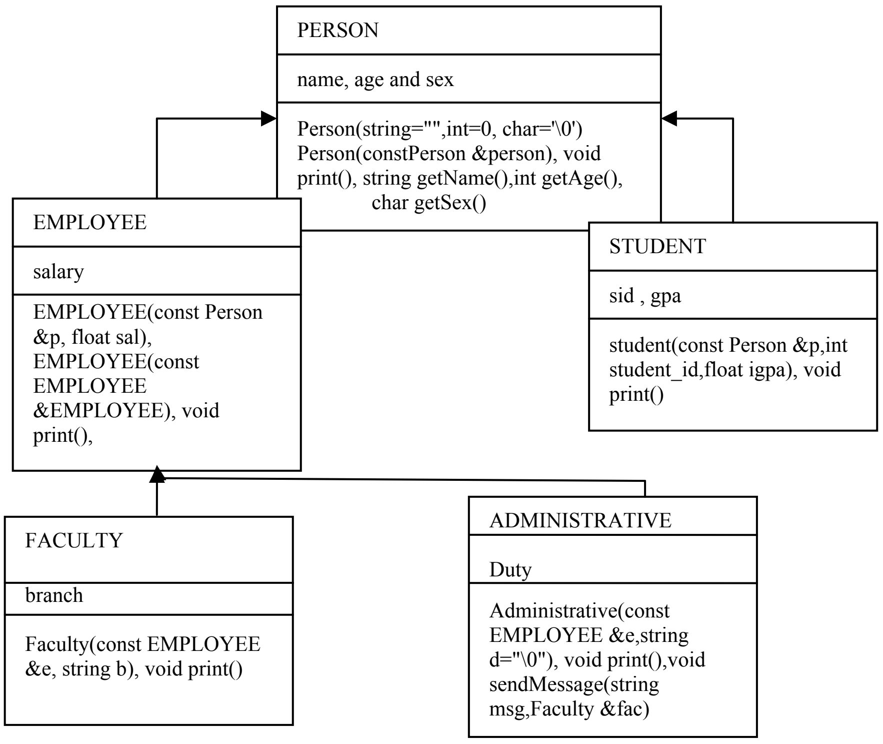

PERSON
name, age and sex
Person(string="",int=0, char='\0') Person(constPerson &person), void print(), string getName(),int getAge(), char getSex()
EMPLOYEE
salary
EMPLOYEE(const Person &p, float sal), EMPLOYEE(const EMPLOYEE &EMPLOYEE), void print(),
STUDENT
sid , gpa
student(const Person &p,int student_id,float igpa), void print()
FACULTY
branch
Faculty(const EMPLOYEE &e, string b), void print()
ADMINISTRATIVE
Duty
Administrative(const EMPLOYEE &e,string d="\0"), void print(),void sendMessage(string msg,Faculty &fac)

Fig. 1. An example of an object oriented system.

PERSON class has six methods and three attributes.
The total method complexity of class PERSON = $\sum MC = W_{p1} + W_{p2} + W_{p3} + W_{p4} + W_{p5} + W_{p6} = 1 + 1 + 3 + 1 + 1 + 1 = 8$.
WCC of PERSON class is given by, $WCC_p = N_a + W_c = 8 + 3 = 11$ *WCCU*.

EMPLOYEE class has three methods and one attribute.
The total method complexity of class EMPLOYEE = $\sum MC = W_{E1} + W_{E2} + W_{E3} = 1 + 1 + 1 = 3$.
WCC of EMPLOYEE class is given by, $WCC_E = N_a + W_c = 3 + 1 = 4$ *WCCU.*

STUDENT class has two methods and two attributes.
The total method complexity for class STUDENT = $\sum MC = W_{S1} + W_{S2} = 1 + 5 = 6$.
WCC of STUDENT class is given by, $WCC_S = N_a + W_c = 6 + 2 = 8$ *WCCU.*

FACULTY class has two methods and one attribute.
The total method complexity = $\sum MC = W_{F1} + W_{F2} = 1 + 1 = 2$.
WCC of FACULTY class is given by, $WCC_F = N_a + W_c = 2 + 1 = 3$ *WCCU.*

ADMINSTRATIVE class has three methods and one attribute.
The total method complexity for ADMINSTRATIVE class = $\sum MC = W_{A1} + W_{A2} + W_{A3} = 1 + 1 + 1 = 3$.
WCC of ADMINSTRATIVE class is given by, $WCC_A = N_a + W_c = 3 + 1 = 4$ *WCCU.*

Total Weighted Class Complexity of the above object-oriented code is given by;
$WCC = \sum WCC = WCC_p + WCC_E + WCC_S + WCC_F + WCC_A = 11 + 4 + 8 + 3 + 4 = 30$ *WCCU.*

4.3 Comparison with Other Measures

A comparative study has been done with most widely accepted Chidamber and Kemerer (CK) metric suits [10]. If we compare our metric with CK metric suits, we find that none of the CK metrics calculates the total complexity of the class by considering the complexity due to internal architecture of the code (methods and messages) and attributes. This differentiates our metric from the CK metrics. However, in one of the CK metrics, 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 also considers the complexity due to attributes. A further advantage of our metric is that, unlike the CK equivalent, it takes cognitive weights into consideration. In the following Table 3, a comparison has been demonstrated with CK metric suits. We calculated the weight of each method by using cognitive weights (WMC(2)) and the approach suggested by Chidamber *et al.* (WMC(1)). We found that the resulting value of WMC(2) is higher than the original WMC(1). This is because, in WMC(1), the weight of each method is assumed to be one. However, including cognitive weights for calculation of the method complexity (WMC(2)) is more realistic because it provides for the complexity of the internal architecture of method.

The depth of inheritance tree (DIT) and the number of children (NOC) are two important CK measures (Table 3). 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 zero to two and vary from class to class depending on the position of class in the hierarchy.

Table 3. Complexity values for different metrics.

Name of CLASS / Metrics	STUDENT	ADMINSTRATIVE	FACULTY	EMPLOYEE	PERSON	Complexity for software system
WCC	8	4	3	4	11	30
WMC(1)	2	3	2	3	6	17
WMC(2)	6	3	2	3	8	22
RFC	2	3	2	3	6	--
DIT	1	2	2	1	0	---
NOC	0	0	0	2	2	--
LCOM	2	0	0	3	6	--
CBO	0	0	0	0	0	--

WCC: Weighted Class 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; NOC: Number of children; LCOM: Lack of cohesion in methods; CBO: coupling between objects.

However, in our proposal, if the depth of the inheritance tree or the number of children is high, it is reflected directly in our metric calculation since we add complexity of the children by their parent class.

Another CK metric is response for the class (RFC). This represents a set of methods that can, potentially, be executed in response to a message received by an object of that class. Since, in our example, there is not any method call for another class, RFC metrics are just the total numbers of methods for that particular class. The difference between RFC and WCC 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, WCC produces higher complexity values than RFC.

The remaining two CK metrics, LCOM and CBO, (Table 3) are related to coupling and cohesion. Therefore they are not comparable with our metric. As a conclusion we can say that CCC can be used to calculate the complexity of object oriented code.

We also compared the proposed measure with other complexity measures in terms of all nine Weyuker's properties (see Table 4). We refer the reader to [10] and [11] for the details of these metrics. We have closely examined seven proposed syntactic complexity measures to see which properties, they have in common and which properties distinguish them. When we compare our measure through Weyuker's properties, we found that the proposed measure satisfies six Weyuker's property out of nine, which established this measure as well structured one. The proposed measure does not satisfy property seven. It is because of changing the order in which methods or attributes are declared does not affect the order in which they are executed. This property is more meaningful in traditional program design where the ordering or if-then-else blocks could alter the program logic and consequent complexity. In OOP, a class is an abstraction of the problem space, and the order of statements within the class definition has no impact on eventual execution or use. This is the reason that no complexity measure does satisfied by this property. Property 9 is also not satisfied by the proposed metric. This property allows the possibility of increased complexity due to interaction. Failing to meet this property implies that complexity metric could increase, rather than reduce, if a class is divided into more classes. Weyuker's Property 9 has received a mixed response regarding

Table 4. Comparison in terms of Weyuker's Properties

P.N.	WMC	DIT	NOC	CBO	RFC	LCOM	CMBOE[*]	WCC
1	Y	Y	Y	Y	Y	Y	Y	Y
2	Y	Y	Y	Y	Y	Y	Y	Y
3	Y	Y	Y	Y	Y	Y	Y	Y
4	Y	Y	Y	Y	Y	Y	Y	Y
5	Y	Y/Y/N	Y	Y	Y	N	N	Y
6	Y	Y	Y	Y	Y	Y	Y	N
7	N	N	N	N	N	N	N	N
8	Y	Y	Y	Y	Y	Y	Y	Y
9	N	N	N	N	N	N	Y	N

[*] CMBOE = Complexity Measures for Object-Oriented Program Based on Entropy.

its applicability to object oriented software metrics and on the contrary to past beliefs, the relevance of this property to object oriented systems is brought out [10]. The proposed complexity metric is therefore can be considered as suitable for OOP.

As a result of case study and comparative study, we can say that WCC can be used to calculate the complexity of OO code 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. Our metric gives valuable idea about the design quality of object oriented codes. High WCC value indicates that understandability and maintainability of the code is weak. Ultimately, it helps the software developer for better design information. 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 attributes, lesser number of branching & looping primitives, is assumed to be more skilful.

5. SUMMARY AND FUTURE WORK

Class is a coherent set of attributes and the method working on the attributes therefore; the complexity of the class depends on these two factors: methods and the attributes. We calculate the cognitive complexity of the code due to these factors. It considers the internal architecture of method and the complexity due to attributes. It is reported in the literature that this property is not satisfied for the other complexity metrics on class level [9, 10]. Some of the features of metric are:

1. It can be used for the cognitive complexity of class by methods and attributes and thereof understandability of the code.
2. It can be used to evaluate efficiency of the design. Low complexity value gives better design information and less maintenance effort.
3. This metric not only see the complexity of the procedure in method but it also consider the attributes and message between the classes. In other words, it measures the important concepts of OOPs like methods, class and coupling.
4. It is language independent complexity metric since it uses cognitive weights and attributes which are the same in all programming languages.

5. The metric is on the ratio scale, a fundamental requirement for a measure from the measurement theory perspective.

Therefore, the proposed metric can be implemented for calculating the cognitive complexity of OO systems. However, there are also some drawbacks as given below.

1. The present method gives the complexity value in number, which are generally high for large programs. High complexity values are not desirable.
2. It is difficult to assign the upper bound for the complexity values.
3. It is not possible to identify the underlying source of complexity with the proposed measure since it depends on several factors such as number of methods, their internal architectures, number of attribute and the number of message calls.

In the light of the experiences we propose the future work to include the following:

1. Assignment of the upper and lower bounds of the complexity values should be investigated.
2. Further analysis is needed for the assessment of class complexity.
3. Apart from the preliminary evaluation, more test cases and typical examples (data from the industry) should be applied for the empirical validation.
4. Improvement of proposed metric should be studied for consideration of remaining features of OOPs.
5. Algorithm development to calculate the class complexity automatically should be considered.

6. CONCLUSIONS

A cognitive complexity metric for OO systems at class level has been formulated. Cognition of a class requires a comprehension of class, which is due to methods and attributes. It is the basic motivation for proposing such a metric, which is capable of calculating the cognitive complexity of class by considering internal architecture of the methods and as well as due to attributes. The metric is theoretically 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.

REFERENCES

1. I. Sommerville, *Software Engineering*, 8th ed., Addison-Wesley, China, 2007.
2. http://www.minds.nuim.ie/~jmcq/NUIM-CS-TR-2006-03.pdf.
3. L. Briand and J. Wust, "Modeling development effort in object oriented system using design properties," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 963-986.
4. M. Choi and E. Cho, "Component identification methods applying method call type between classes," *Journal of Information Science and Engineering*, Vol. 22, 2006, pp. 247-267.

5. R. K. Bandi, V. K. Vaishnavi, and D. E. Turk, "Predicting maintenance performance using object-oriented design complexity metrics," *IEEE Transactions on Software Engineering*, Vol. 29, 2003, pp. 77-87.
6. L. C. Briand, C. Bunse, and J. W. Daly, "A controlled experiments for evaluating quality guidelines on the maintainability of object-oriented design," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 513-530.
7. D. Abbot, "A design complexity metric for object-oriented development," Masters Thesis, Department of Computer Science, Clemson University, U.S.A., 1993.
8. M. Lorenz and J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
9. G. Costagliola and G. Tortora, "Class points: an approach for the size estimation of object-oriented systems," *IEEE Transactions on Software Engineering*, Vol. 31, 2005, pp. 52-74.
10. S. R. Chidamber and C. F. Kermer, "A metric suite for object oriented design," *IEEE Transactions Software Engineering*, Vol. 6, 1994, pp. 476-493.
11. K. Kim, Y. Shin, and C. Wu, "Complexity measures for object-oriented program based on the entropy," in *Proceedings of the 2nd Asia-Pacific Software Engineering Conference*, 1995, pp. 127-136.
12. J. Kim and J. F. Lerch, "Cognitive processes in logical design: comparing object-oriented and traditional functional decomposition software methodologies," Working Paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburg, 1991.
13. J. A. Mc Quillan and J. F. Power, "On the application of software metrics to UML model," *Lecture Notes in Computer Science*, Vol. 4364, 2007, pp. 217-226.
14. Y. Wang and J. Shao, "A new measure of software complexity based on cognitive Weights." *IEEE Canadian Journal of Electrical and Computer Engineering*, 2003, pp. 69-74.
15. S. Misra, "An object-oriented complexity metric based on cognitive weight" in *Proceedings of IEEE International Conference on Cognitive Informatics*, 2006, pp. 134-139.
16. Y. Wand and R. Weber, "Toward a theory of the deep structure of information systems," in *Proceedings of International Conference on Information Systems*, 1990, pp. 61-71.
17. N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Transactions on Software Engineering*, Vol. 18, 1992, pp. 1038-1044.
18. I. Vessey and R. Weber, "Research on structured programming: an empiricist's evaluation," *IEEE Transactions on Software Engineering*, Vol. 10, 1984, pp. 394-407.
19. E. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, Vol. 14, 1988, pp. 1357-1365.
20. R. E. Prather, "An axiomatic theory of software complexity measures," *Computer Journal*, Vol. 27, 1984, pp. 340-346.
21. I. N. Gorla and R. Ramakrishnan, "Effect of software structure attributes software development productivity," *Journal of System and Software*, Vol. 36, 1997 pp. 191-199.
22. IEEE Std. 1061-1998 – IEEE Computer Society: Standard for Software Quality Metrics Methodology, 1998.

23. P. Coad and E. Yourdon, *Object Oriented Analysis*, 2nd ed., Prentice-Hall, New Jersey, 1991.
24. N. Sharma, P. Joshi, and R. K. Joshi, "Applicability of Weyuker's property 9 to object oriented metrics," *IEEE Transactions on Software Engineering*, Vol. 32, 2006, pp. 209-211.
25. G. Saran and G. Roy, "On the applicability of Weyuker property nine to object oriented structural inheritance complexity metrics," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 361-364.
26. J. C. Cherniavsky and C. H. Smith, "On Weyuker's axioms for software complexity measures," *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp. 636-638.
27. L. C. Briand, S. Morasca, and V. R. Basily, "Property based software engineering measurement," *IEEE Transactions on Software Engineering*, Vol. 22, 1996, pp. 68-86.
28. N. Fenton, "Software measurement: a necessary scientific basis," *IEEE Transactions on Software Engineering*, Vol. 20, 1994, pp. 199-206.
29. B. Kitchenham and N. Fenton, "Towards a framework for software measurement validation," *IEEE Transactions on Software Engineering*, Vol. 21, 1995, pp. 929-943.
30. S. Morasca, "Foundations of a weak measurement-theoretic approach to software measurement," *Lecture Notes in Computer Science*, Vol. 2621, 2003, pp. 200-215.
31. Y. Wang, "The measurement theory for software engineering," in *Proceedings of Canadian Conference on Electrical and Computer Engineering*, 2003, pp. 1321-1324.
32. H. Zuse, "Properties of software measures," *Software Quality Journal*, Vol. 1, 1992, pp. 225-260.
33. C. Kaner, "Software engineering metrics: what do they measure and how do we know?" in *Proceedings of the 10th International Software Metrics Symposium*, 2004. pp. 1-12

APPENDIX: CLASSES AND SUBCLASSES FOR THE CASE STUDY

Appendix 1. PERSON-STUDENT-EMPLOY-FACULTY-ADMINSTRATIVE

```
#include <iostream>
#include <string>
using namespace std;
/* Person Class is base class.Student&EMPLOYEE both inherits Person Class */
```

```
                         /* ******** PERSON CLASS ********** */
class Person{
             string name; int age; char sex;
  public:
      Person(string="" ,int=0, char='\0');                    // W p1=1
      Person(const Person &person); //copy constructorW p2=1
void print()const;                                            //Wp3=Wp31+Wp32=2+1=3
      string getName(){                                       // Wp4=1
      return name;   }
      int getAge(){                                           //Wp5=1
      return age;}
      char getSex(){                                          //Wp6=1
      return sex; }
         };
                 //Person-default constructor
    Person :: Person(string in, int ia, char is)
           { name = in; age = ia; sex = is; }
                //Person-copy constructor
```

```
Person :: Person(const Person &p)
{                   name = p.name;    age = page;  sex = p.sex;        }
void Person :: print()const
{                   cout<<"Name\t : "<<name<<'\n' ;                        //Wp31=1
                    cout<<"Age\t : "<<age<<'\n' ;
                    if (sex=='F')                                          //Wp32=2
                        cout<<"Sex\t : Female" <<'\n' ;
        else            cout<<"Sex\t : male" <<'\n' ;                     }
```

```
                           /* ******** STUDENT CLASS ********** */
class Student: public Person{   int sid; float gpa;
public:Student(const Person &p,int student_id,float igpa): Person(p)          //WS1=1
                    {  sid = student_id;
                    gpa = igpa;   }
                    void print()const;     };                                //WS2=WS21+WS22*WS23=1+2*2=5
                   void Student :: print()const
{           Person :: print();
           cout<<"S.ID\t:"<<sid<<"\nGPA\t:"<<gpa<<endl;                       //WS21=1
        if (gpa>=2.0)                                                        //WS22=2
            cout<<" Student is successful"<<endl;
        else {if (gpa>=1.7)                                                  //WS23=2
                        cout<<"Student must improve GPA" <<endl;
        else
                cout<<"Student must repeat" <<endl;}}
```

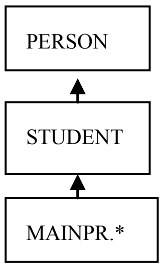
```
                           /* ******** EMPLOYEE CLASS ********** */
class EMPLOYEE: public Person{  float salary;
 public: EMPLOYEE::EMPLOYEE(const Person &p, float sal):Person(p) ,salary(sal){} //WE1=1
EMPLOYEE(const EMPLOYEE &EMPLOYEE):Person(EMPLOYEE){
                    salary=EMPLOYEE.salary;      }                           //WE2=1
                    void print()const;                   };                  //WE3=1
     void EMPLOYEE::print() const{ Person::print();
                    cout<<"salary: "<<salary<<endl; }
```

```
                           /* ******** FACULTY CLASS ********** */
class Faculty: public EMPLOYEE{
                    string branch; //Physics , Math, or Ceng, etc .
       public:      Faculty(const EMPLOYEE &e, string                        //WF1=1
b):EMPLOYEE(e),branch(b){}
                    void print()const;               };                      //WF2=1
```

```
                        /* ******** ADMINISTRATIVE CLASS ********** */
class Administrative: public EMPLOYEE{string duty;                           //Secretary, Accountant
public:
Administrative(const EMPLOYEE &e,string d="\0"):EMPLOYEE(e){duty=d;}      //WA1=1
                      void print() const;          };                           //WA2=1
void sendMessage(string msg,Faculty &fac)                                       //WA3=1
                    {cout<<"The incoming message :"<<msg<<". \nMessage to";    cout<<fac.getName();}
```

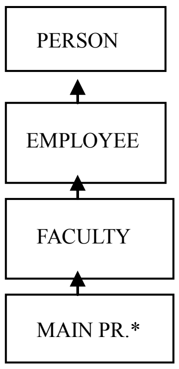
```
                     /* ********************* MAIN ********************** */
int main(void)
{                   Person * per[3];
                    per[0]=new Person ("Aysegul",27,'f');
                    per[1]=new Person ("Remzi",23,'m');
                    per[2]=new Person ("Ali",30,'m');
                    EMPLOYEE EMPLOYEE1(* per[0],1000);
                    EMPLOYEE1.print();
                    Student student1(* per[1],9299,3.5);
                    student1.print();
                    EMPLOYEE EMPLOYEE2(* per[0],2000);
                    Administrative admEMPLOYEE(EMPLOYEE1,"Secretary");
                    Faculty facEMPLOYEE(EMPLOYEE2,"Computer");
                    admEMPLOYEE.sendMessage("Today there is a seminar at your university. You are in
                   vited",facEMPLOYEE);}
```

Appendix 2. CLASS: PERSON-STUDENT



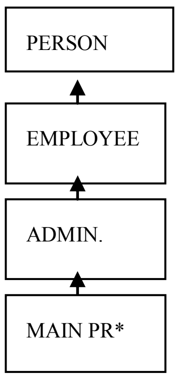
```
*int main(void)
{Person * per[3];
        per[0]=new Person ("Aysegul",27,'f');
        per[1]=new Person ("Remzi",23,'m');
        per[2]=new Person ("Ali",30,'m');
        Person person1("fatmagul",27,'f');
        Student student1(* per[1],9299,3.5);
        student1.print(); }
```

Appendix 3. CLASS: PERSON-EMPLOYEE-FACULTY



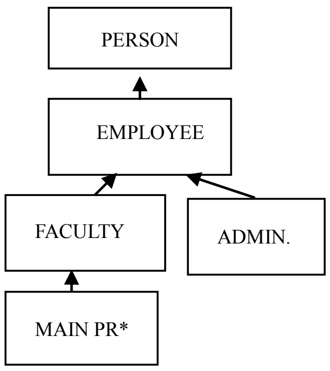
```
*int main(void){Person * per[3];
            per[0]=new Person ("Aysegul",27,'f');
            per[1]=new Person ("Remzi",23,'m');
            per[2]=new Person ("Ali",30,'m');
            EMPLOYEE EMPLOYEE2(*
            per[0],2000);
            Faculty facEM PLOYEE (EM
            PLOYEE2,"Computer");}
```

Appendix 4. CLASS: PERSON-EMPLOYEE-ADMINISTRATIVE



```
*int main(void) { Person * per[3];
 per[0]=new Person ("Aysegul",27,'f');
 per[1]=new Person ("Remzi",23,'m');
 per[2]=new Person ("Ali",30,'m');
 EMPLOYEE EMPLOYEE1(* per[0],1000);
 EMPLOYEE1.print();
 AdministrativeadmEM PLOYEE (EM
 PLOYEE1,"Secretary");
 admEMPLOYEE.sendMessage("Today there
 is a seminar at your university. You are in
 vited",facEMPLOYEE);}
```

Appendix 5. CLASS: PERSON-EMPLOYEE-FACULTY-ADMINISTRATIVE



```
*int main(void) {Person * per[3];
 per[0]=new Person ("Aysegul",27,'f');
 per[1]=new Person ("Remzi",23,'m');
 per[2]=new Person ("Ali",30,'m');
 EMPLOYEE EMPLOYEE1(* per[0],1000);
 EMPLOYEE1.print();
 EMPLOYEE EMPLOYEE2(* per[0],2000);
 Administrative admEMPLOYEE(EMPLOYEE1,"Secretary");
 Faculty facEMPLOYEE(EMPLOYEE2, "Computer");
 admEMPLOYEE.sendMessage ("Today there is a seminar
 at your university. You are invited", facEMPLOYEE)
```

Sanjay Misra is Assistant Professor in Department of Computer Engineering, Atilim University, Ankara, Turkey. He obtained M.Tech. degree in Software Engineering from Motilal Nehru National Institute of Technology, India and Ph.D. from University of Allahabad, India. He has a wide experience (more than 16 years) of research, teaching and academic administration. His areas of interests are software measurement, object oriented technologies, cognitive informatics, SOA, XML technologies and web-services. He published more than 40 research papers in these areas. Presently, he is chief editor of International Journal of Computer Engineering and Software Technology (IJCSST).

K. Ibrahim Akman is Professor and the Chair of Computer Engineering Department at Atilim University. He obtained his Ph.D. in operations research from Lancaster University (U.K.) in 1984. Dr. Akman has served on the editorial boards of Electronic Journal of e-Government and International Journal of Information Technology and Management (IJITM). Software piracy, e-Government, human resource management and data compression are amongst his current fields of interest. He published over 50 articles in conferences and journals including Studies in Educational Evaluation, Computational Statistics and Data analysis, Microelectronics and Reliability, Journal of Information Science, and Government Information Quarterly.