This is a preprint of the article: "S. Misra, M. Koyuncu, M. Crasso, C. Mateos and A. Zunino: "A Suite of Cognitive Complexity Metrics". Lecture Notes in Computer Science (3th International Workshop on Software Quality - ICCSA 2012). Vol. 7336, pp. 234-247. Springer-Verlag. 2012. ISSN 0302-9732."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_17

A Suite of Cognitive Complexity Metrics

Sanjay Misra[1], Murat Koyuncu[1] , Marco Crasso[2], Cristian Mateos[2] and Alejandro Zunino[2]

[1] Department of Computer Engineering, Atilim University, Ankara, Turkey
{smisra, mkoyuncu}@atilim.edu.tr

[2] ISISTAN Research Institute. UNICEN University, Tandil, Argentina. Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET). {mcrasso|cmateos|azunino}@conicet.gov.ar

Abstract. In this paper, we propose a suite of cognitive metrics for evaluating complexity of object-oriented (OO) codes. The proposed metric suite evaluates several important features of OO languages. Specifically, the proposed metrics are to measure method complexity, message complexity (coupling), attributes complexity and class complexity. We propose also a code complexity by considering the complexity due to inheritance for the whole system. All these proposed metrics (except attribute complexity) use the cognitive aspect of the code in terms of cognitive weight. All the metrics have critically examined through theoretical and empirical validation processes.

Keywords: software metrics, methods, messages, attributes, class, coupling, inheritance, cognitive complexity, validation.

1 Introduction

IEEE defines the software quality as 'Software quality is the degree to which software possesses a desired combination of attributes (e.g., reliability, interoperability)' [1]. Software quality is controlled by software metrics. Software metrics are tools to control the complexity of software. Through metrics, one can easily observe the several weaknesses of a software system and therefore, by means of it, quality can be estimated. This is the reason why metrics are indispensable tool in software development life cycle for achieving the quality.

The recent decades have witnessed the successes of the object-oriented (OO) languages. Most of the projects are being developed in JAVA, C++ or in Python. The need to control the complexity of the projects developed in these language is important. For this purpose, since the beginning of the 1990 several object-oriented metrics e.g. Chidamber and Kemerer (CK) metrics suite [2], MOOD metrics for OO Design [3], design metrics for testing [4], product metrics for object-oriented design [5-6], Lorenz and Kidd metrics [7], Henderson–Seller metrics [8], (slightly) modified

CK metrics [9], size estimation of OO systems[10] , weighted class complexity metric [11] and several other metrics[12-17] can be found in the literature. All the above metrics tried to cover some features of the OO languages and used for some quality attributes. The quality attributes, are such as correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability [18]. Amongst the given quality attributes, maintainability is treated as the most necessary attribute for software products [19]. In fact, majority of the metrics are developed for this most important attribute.

In our previous work, we have presented metrics for OO codes [11]. For inheritance complexity, we have first calculated the cognitive weights of all the methods of a class, sum up them and then multiply with the weight of their parent classes (due to inheritance). However, later, we have observed that while considering complexity due to inheritance, we should not only consider the method complexity but the complexity due to attributes. In this point of view, while estimating the complexity of the entire OO codes, we have to calculate the complexity of a class by considering the impact due to method complexity, message complexity and also due to attributes [11]. Then, we have to establish a relation between classes to capture the complexity due to inheritance property. The mentioned requirement is the starting point of this work and we present a suite of metrics which capture most of features of the OO programming paradigm in this paper.

The paper is organized as follows: The motivation of the work is given in the next section. Section 3 presents the proposal of the new suite of complexity metrics. The metrics are demonstrated with an OO example in Section 4. Finally, a conclusion is given in Section 5.

2 Motivation

After the CK metric suite, no further attempts have been made seriously in this direction to develop a more effective suite of metrics for OO languages [2]. All the metrics in CK metric suite are straight forward and simple to compute. On the other hand, these metrics do not cover the following issues:

1. The overall complexity of a class due to all possible factors
2. The internal architecture of the class
3. The impact of the relationship due to inheritance in the class hierarchy
4. The number of messages between classes and their complexities (CK metrics suite counts only the methods coupled with other classes)
5. Cognitive complexity, which is a measure of understandability and therefore has a great impact on maintainability of the system

The lack of the above features in CK metric suite motivated us to produce a new suite of metrics, which can be a complimentary set of the CK metric suite. In fact, our proposed metrics suggest examining the OO properties in more detail. For example, CBO (one of metrics in CK metric suite) is a measure to show interactions between objects by counting the number of other classes to which the class is coupled. In our proposal, coupling is computed by considering the message calls to other classes and the weight of the called methods. One class may have “1” for CBO showing that it

interacts with only one class, but may include many messages to that class which causes a more complex code (which is considered in our metric). Therefore, we believe that our metric gives more accurate information about coupling of a class.

3 Proposed Suite of Metrics for Object-Oriented Programming

An object is a class instance and an object-oriented system consists of objects which collaborate through message exchanges. An object-oriented code includes one or more classes which may be related to each other by composition or inheritance and contains related attributes and operations (methods) in the classes. The complexity metrics developed for object-oriented languages are mainly based on the complexity of individual classes like number of methods, number of messages etc. However, not only the numbers of different components are important, but also the internal complexities of these components are equally important. Furthermore, for calculating the complexity of the entire system, we have to consider the special features of OO programs and type of the relations between classes. Accordingly, we propose the following suite of metrics:

Method Complexity(MC): Method complexity is calculated by considering corresponding cognitive weights of structures in a method of a class. Cognitive weights are used to measure the complexity of the logical structures of the software in terms of Basic Control Structures (BCSs). These logical structures reside in the method (code) and are classified as sequence, branch, iteration and call with the corresponding weights of one, two, three and two, respectively. Actually, these weights are assigned on the classification of cognitive phenomenon as discussed by Wang [20]. We calculate method complexity in a class by associating a number (weight) with each member function (method), and then simply add the weights of all the methods. More formally, the method complexity (MC) is calculated as;

$$MC = \sum_{j=1}^{q} \left\lceil \prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right\rceil, \quad (1)$$

where, W_c is the cognitive weight of the concerned Basic Control Structure (BCS). The method complexity of a software component is defined as the sum of cognitive weights of its q linear blocks composed of individual BCSs, since each block may consist of m layers of nested BCSs, and each layer with n linear BCSs. Equation 1 gives the complexity of a single method.

Message complexity (Coupling Weight for a Class (CWC)): Two classes are coupled when there is a message call in one class for the other class. In our proposal, if there are message calls for other classes, we not only count the total number of such messages, but also we add the weight of the called methods. Accordingly, complexities due to message calls are the sum of weights of call and the weight of called methods. i.e.

$$CWC = \sum_{i=1}^{n} (2 + MC_i), \quad (2)$$

where, 2 is the weight of the message to an external method and W_i is the weight of the called method. If there are n numbers of external calls, then the CWC is calculated as the sum of weights of all message calls.

Attribute Complexity(AC): It reflects the complexity due to data members (attributes). We simply assign the total number of attributes associated with class as the complexity due to data members. The attributes are not local to one procedure but local to objects and can be accessed by several procedures. Accordingly, the attribute complexity of a class (AC) is given by:

$$AC = \sum_{i=1}^{n} 1, \quad (3)$$

where n is the total number of attributes.

Weighted Class Complexity(WCC): OO software development is based on classes and subclasses whose elements are attributes and methods (including messages). These elements are identified in class declarations and are responsible for the complexity of a class. Therefore, the complexity of a class is a function of the methods and the data attributes. More formally, we suggest the following formula to calculate the Weighted Class Complexity (WCC):

$$WCC = AC + \sum_{p=1}^{n} MC_p \quad (4)$$

WCC is the sum of the attribute complexity and the sum of all the method complexities of a class.

Code Complexity (Inheritance): For calculating the complexity of the entire system, we have to consider not only the complexity of all the classes, but also the relations among them. That is, we are giving emphasis on the inheritance property because classes may be either parent or children classes of others. In the case of a child class, it inherits the features from the parent class. By keeping this property of OO paradigm, we propose to calculate the code complexity of an entire system as follows:

- If the classes are of the same level then their weights are added.
- If they are subclasses or children of their parent then their weights are multiplied.

If there are m levels of depth in the object-oriented code and level j has n classes then the Code Complexity (CC) of the system is given by,

$$CC = \prod_{j=1}^{m} \left[\sum_{k=1}^{n} WCC_{jk} \right] \quad (5)$$

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

In addition to these metrics, we are also proposing the associated metrics which are extracted from the above metrics. These metrics may be useful indications for general information regarding the projects.

Average Method Complexity(AMC): It gives an average method complexity for a class and is calculated by dividing the sum of complexities of all the methods of a class to the total number of methods in that class.

$$AMC = \sum_{p=1}^{n} MC_p / n, \quad (6)$$

where MC is the complexity of a particular method, n is total number of methods in a class.

Average Method Complexity per Class(AMCC): It is defined as the average method complexity for the entire system.

$$AMCC = \sum_{p=1}^{m} AMC / m, \quad (7)$$

where m is total number of classes in a project.

Average Class Complexity(ACC): It is the average complexity of classes in a project and it is calculated by dividing the sum of the complexity of the classes to the total number of classes.

$$ACC = \sum_{p=1}^{m} WCC / m, \quad (8)$$

where WCC is the complexity a class and m is total number of classes.

Average Coupling Factor(ACF): It is defined as the complexity of all the external method calls (i.e. coupling weights) to the total number of messages.

$$ACF = \sum_{i=1}^{k} CWC / k \quad (9)$$

where k is number of messages to other classes.

Average Attributes per Class(AAC): It shows the average number of attributes per class in a project and it is calculated by dividing the sum of attribute complexity of all classes to the total number of classes.

$$AAC = \sum_{i=1}^{m} AC / m , \quad (10)$$

where, m is the total number of classes.

4 Demonstration of the Metrics

The proposed complexity metrics given in Section 3 is demonstrated with a programming example in this section. The class hierarchy of the example program is illustrated in Fig.1 and the complete C++ code for the example is given in Appendix.

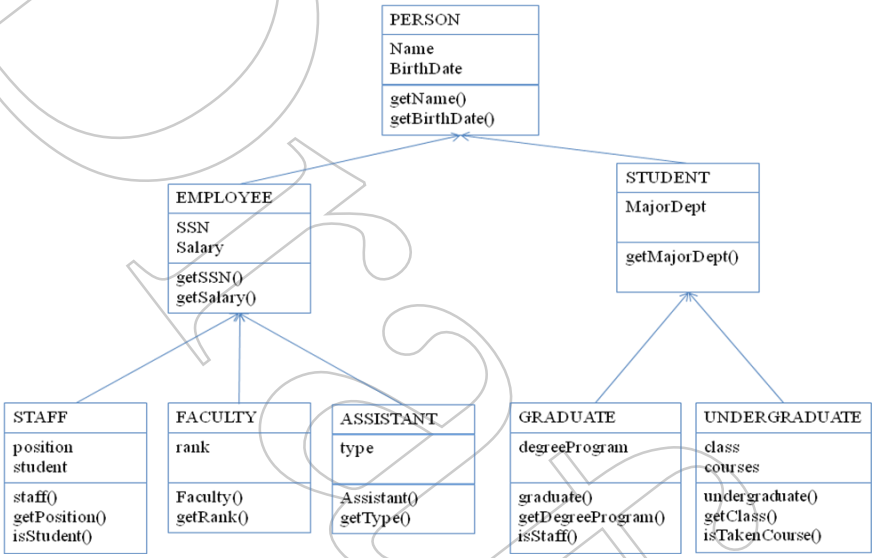

Fig. 1. An Example Class Hierarchy

The given example processes a personnel database hierarchy. It has one main class *Person* and two subclasses, *Employee* and *Student*. The class *Employee* has again three subclasses, *Staff*, *Faculty*, and *Assistant*. The *student* class has two subclasses, *Graduate* and *Undergraduate*. This section demonstrates how we calculate the complexities according to the metrics given in section 3 for an object-oriented program.

Method Complexity (MC): Method complexity of each method is calculated using Formula 1. For example, the class *Person* has two methods named as *getName()* and

getBirthDate(). Each of these methods has simple structure, called as sequence basic control structure (BCS), therefore their weights are assigned as 1.

$$MC_{getName}=MC_{getBirthDate}=\sum_{j=1}^{q}\left\lceil\prod_{k=1}^{m}\sum_{i=1}^{n}W_c(j,k,i)\right\rceil=1$$

So, the method complexity for the class person:

$$MC_{PERSON}= MC_{getName} + MC_{getBirthDate} =1+1=2 \text{ CCU}$$

For another example, consider *isStudent()* method of the *Staff* class. The method includes an IF structure (branch). The method complexity is calculated as:

$MC_{isStudent}$= 1+2 = 3, where 1 is for sequence and 2 is for branch (IF) structure.

The method *isStaff()* of the class *Graduate* shows a more detailed example:

MCisStaff= 1+2((2+3)+2) = 15, where 1 is for sequence and 2 is for branch (IF) structure. The branch structure has a external method call and a nested branch inside. (2+3) is for the message sent to another class (i.e.,*Staff*), 2 is the weight of the message and 3 is the weight of the called method (i.e. *isStudent()*). The last 2 in the calculation is for the nested IF structure. Notice that if there is nested structure, we multiply the weights instead of summing them up.

The method complexity for the classes given in Fig.1 is calculated as follows:

$MC_{PERSON}= MC_{getName} + MC_{getBirthDate} =1+1=2$ CCU
$MC_{EMPLOYEE}= MC_{getSalary} + MC_{getSSN} =1+1=2$ CCU
$MC_{STUDENT}= MC_{getMajorDept} =1$ CCU
$MC_{STAFF}= MC_{staff} + MC_{getPosition} + MC_{isStaff} =1+1+3=5$ CCU
$MC_{FACULTY}= MC_{faculty} + MC_{getRank} =1+1=2$ CCU
$MC_{ASSISTANT}= MC_{assistant} + MC_{getType} =1+3=4$ CCU
$MC_{GRADUATE}= MC_{graduate} + MC_{getDegreeProgram} + MC_{isStaff} =1+1+15=17$ CCU
$MC_{UNDERGRADUATE}= MC_{undergraduate}+MC_{isTakenCourse}+MC_{getClass} =4+7+1=12$ CCU

All the method complexities can be seen in Appendix along with the code of each method.

Message complexity (Coupling Weight for a Class (CWC)): In the given example, there is only one class (*Graduate*) which includes one external message call to the *Staff* class. We can calculate the coupling weight of the class *Gradute* as the weight of the called methods. For this example, there is only one external method called from the *Graduate* class(isStaff()).

$$CWC = \sum_{i=1}^{n}(2+MC_i) = 2 + 3= 5 \text{ CCU}$$

Attribute Complexity (AC): Attribute complexity of a class can be calculated by counting the total number of attributes in that class. Accordingly, AC values for classes Person, Employee, Student, Faculty, Staff, Assistant, Graduate and Undergraduate are 2, 2, 1, 2, 1, 1, 1 and 2, respectively.

Weighted Class Complexity (WCC): WCC for each class can be calculated by Equation 4, i.e the sum of attribute complexity, method complexity and message complexity. It is worth mention that while calculating the WCC, we don't need to include the message complexity of classes, because, a message is a part of a method and already calculated in the method complexity. For the given example, WCCs are calculated as follows:

WCC_{PERSON}= 2+2= 4 CCU
$WCC_{EMPLOYEE}$= 2+2= 4 CCU
$WCC_{STUDENT}$= 1+1= 2 CCU
WCC_{STAFF}= 2+5= 7 CCU
$WCC_{FACULTY}$= 1+2= 3 CCU
$WCC_{ASSISTANT}$= 1+4= 5 CCU
$WCC_{GRADUATE}$= 1+17= 18 CCU
$WCC_{UNDERGRADUATE}$= 2+12= 14 CCU

Code Complexity(CC): The code complexity of the object-oriented code is calculated by using Equation 5 as given below:

CC = WCCPERSON *(WCCEMPLOYEE *(WCCSTAFF +
WCCFACULTY + WCCASSISTANT)+ WCCSTUDENT *
(WCCGRADUATE + WCCUNDERGRADUATE))
= 4*(4*(7+3+5) + 2*(18+14))
= 496 CCU

Average Method Complexity(AMC):

$AMC = \sum_{p=1}^{n} MC_p / n$,where W is the weight of a particular method, n is total number of method in a class.

AMC_{PERSON}= 2/2= 1 CCU
$AMC_{EMPLOYEE}$= 2/2= 1 CCU
$AMC_{STUDENT}$= 1/1= 1 CCU
AMC_{STAFF}= 5/3= 1.66 CCU
$AMC_{FACULTY}$= 2/2= 1 CCU
$AMC_{ASSISTANT}$= 4/2= 2 CCU
$AMC_{GRADUATE}$= 17/3= 5.66 CCU
$AMC_{UNDERGRADUATE}$= 12/3= 4 CCU

Average Method Complexity per Class(AMCC):

$AMCC = \sum_{p=1}^{m} AMC / m$, where m is total number of classes in a project

AMCC= (1 + 1+ 1+ 1.66 + 1+ 2+ 5.66 + 4)/8 = 2.165 CCU

The average method complexity of a class is 2.165. It is worth mentioning that this number is not the average number of methods per class but it represents the average complexity/weight of method per class

Average Class Complexity(ACC):

$ACC = \sum_{p=1}^{m} WCC / m$, where WCC is the complexity a class and m is total number of classes.

ACC = (4 + 4 + 2 + 7 + 3 + 5 + 18 + 14) / 8 = 7.125 CCU

i.e. the average class complexity of this project is 7.125 CCU.

Average Coupling Factor(ACF):

$ACF = \sum_{i=1}^{k} CWC / k$, where CWC is the Coupling Weight for a Class and k is number of messages to other classes.

$ACF_{GRADUATE}$ = 5/1 = 5

The average coupling factor for class Graduate is 3. There is only one method call to the outside in that class.

Average Attributes per Class(AAC):

$AAC = \sum_{i=1}^{m} AC / m$, where AC is the attribute complexity and m in the total number of classes.

AAC = (2 + 2 + 1 + 2 + 1 + 1 + 1 + 2)/8 = 1.5

i.e. the average number of attributes per class is 1.5.

5. Conclusions

A suite of object-oriented metrics are proposed in this study. The application of metric suite is shown on an example object-oriented code. These metrics are capable to capture most of the features existing in object-oriented codes such as method, attribute, class, inheritance and coupling. Further, the objective to produce such a metric suite is to combine most of the feature responsible for complexity. These

metrics calculate the complexity at each level of the code and the code complexity represents the structural and cognitive complexity of an OO system.

References

1. IEEE Standard 1061-1992: Standard for a Software Quality Metrics Methodology. New York: Institute of Electrical and Electronics Engineers (1992)
2. Chidamber S.R., and Kermerer, C. F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 6, 476-493 (1994)
3. Harrison, R., Counsell, S.J., and Nithi, R.V.: An Evaluation of the MOOD Set of Object Oriented Software Metrics. IEEE Transactions on Software Engineering, 24(6), 491-496 (1998)
4. Binder, R.V.: Object-Oriented Software Testing. Communications of the ACM, 37(9), 28-29 (1994)
5. Vaishnavi, V. K., Purao,S., and Liegle J.: Object-Oriented Product Metrics: A Generic Framework. Information Science, 177, 587-606 (2007)
6. Purao S., and Vaishnavi, V.K.: Product Metrics for Object Oriented Systems. ACM Computing Surveys, 35(2), 191-221 (2003)
7. Lorenz, M., Kidd, J.: Object-Oriented Software Metrics. Prentice Hall, Englewood Cliffs, New Jersey (1994).
8. Henderson-Selles, B.: Object-Oriented Metrics, Measure of Complexity. Prentice-Hall Englewood Cliffs, New Jersey (1996)
9. Basily, V.R., Briand, L.C., and Melo, W.L. : A Validation of Object Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22(1), 751-761 (1996)
10. Costagliola, G., Ferrucci, F., Tortora, G., Vitiello G.: Class Points: An Approach for the Size Estimation of Object-Oriented Systems. IEEE Transactions on Software Engineering, 31(1), 52-74 (2005)
11. Misra, S. and Akman, I.: Weighted Class Complexity: A Measure of Complexity for Object-Oriented System. Jour. of Information Science and Engineering, 24, 1689-1708 (2008)
12. Stephen H. Kan: Metrics and Lessons Learned for OO Projects, Chapter 12: Metrics and Models in Software Quality Engineering. Addison-Wesley (2003)
13. Babsiya, J. and Davis, C.G.: A Hierarchical Model for Object Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, 28(1), 4-17 (2002)
14. Briand L., Wust, J.: Modeling Development Effort in Object Oriented System Using Design Properties. IEEE Transactions on Software Engineering, 27(11), 963-986 (2001)
15. Kim,K., Shin,Y., and Wu,C. : Complexity Measures for Object-Oriented Program Based on the Entropy. In: Proc. Asia Pacific Software Engineering , pp. 127–136 (1995)
16. Kim, J., and Lerch, J.F.: Cognitive Processes in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies. Carnegie Mellon University, Graduate School of Industrial Administration, Working Paper (1991).
17. Olague, H.M., Etzkorn, L.H., Gholston, S. and Quattlebaum, S.: Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes**.** IEEE Transactions on Software Engineering, 33(6), 402-419 (2007)
18. Pfleeger, S.L., Atlee, J.M.: Software Engineering – Theory and Practice. Prentice Hall (2006)
19. Sommerville, I.: Software Engineering, Addison Wesley (2004)

20. Wang Y., Shao, J.: A New Measure of Software Complexity Based On Cognitive Weights. Canadian Journal of Electrical and Computer Engineering, 28, 69-74 (2003)

Appendix: Classes for the Case Study

```
#include <iostream.h>

/*****************CLASS PERSON******************/
class person {
public:
    char * getName(){return name;};                 //W_getName=1
    char * getBirthDate() {return birthDate;};      //W_getBirthDate=1
protected:
    char * name;
    char * birthDate;};

/*****************CLASS EMPLOYEE******************/
class employee :  public person
{
public:
    int getSalary(){return salary;};                //W_getSalary=1
    char * getSSN(){return SSN;};                   //W_getSSN=1
protected:
    int salary;
    char * SSN;};

/*****************CLASS STUDENT******************/
class student :  public person
{
public:
    char * getMajorDept(){return majorDept;};       //W_getMajorDept=1
protected:
    char * majorDept;};

/*****************CLASS STAFF******************/

class staff:  public employee
{
public:
    staff(char * tname, char * tSSN, char * tbirthDate,
                int tsalary, char * tposition, bool tstudent);
    char * getPosition(){return position;};         //W_getPosition=1
        bool isStudent();
protected:
    char * position;
    bool student;};

staff::staff(char * tname, char * tSSN, char * tbirthDate,
             int tsalary, char * tposition, bool tstudent){
     name= tname;                                   //W_staff=1
     SSN=tSSN;
```

This is a preprint of the article: "S. Misra, M. Koyuncu, M. Crasso, C. Mateos and A. Zunino: "A Suite of Cognitive Complexity Metrics". Lecture Notes in Computer Science (3th International Workshop on Software Quality - ICCSA 2012). Vol. 7336, pp. 234-247. Springer-Verlag. 2012. ISSN 0302-9732."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_17

```
     birthDate=tbirthDate;
     salary=tsalary;
     position=tposition;
     student=tstudent;};

bool staff::isStudent(){                    //W_isStudent=1+2=3
        if (student==0)
           return false;
        else
           return true;};

/*****************CLASS FACULTY ******************/
class faculty:  public employee
{
public:
    faculty(char * tname, char * tSSN, char * tbirthDate,
                                    int tsalary, char * trank);
    char * getRank(){return rank;};                //W_getRank=1
protected:
    char * rank;};

faculty::faculty(char * tname, char * tSSN, char * tbirthDate,
                int tsalary, char * trank){        //W_faculty=1
     name= tname;
     SSN=tSSN;
     birthDate=tbirthDate;
     salary=tsalary;
     rank=trank; };

/*****************CLASS ASSISTANT ******************/
class assistant:  public employee
{
public:
    assistant(char * tname, char * tSSN, char * tbirthDate,
                 int tsalary, short type);
    char * getType();
protected:
    short type;};

assistant::assistant(char * tname, char * tSSN,
                char * tbirthDate, int tsalary, short ttype){
     name= tname;                                  //W_assistant=1
     SSN=tSSN;
     birthDate=tbirthDate;
     salary=tsalary;
     type=ttype; };

char * assistant::getType(){                //W_getType=1+2=3
     if (type==1)
         return("Research assistant");
     else
         return("Teaching assistant");
         };
```

This is a preprint of the article: "S. Misra, M. Koyuncu, M. Crasso, C. Mateos and A. Zunino: "A Suite of Cognitive Complexity Metrics". Lecture Notes in Computer Science (3th International Workshop on Software Quality - ICCSA 2012). Vol. 7336, pp. 234-247. Springer-Verlag. 2012. ISSN 0302-9732."

The original publication is available at http://dx.doi.org/10.1007/978-3-642-31128-4_17

```
/*****************CLASS GRADUATE ******************/
class graduate:  public student
{
public:
    graduate(char * tname, char * tbirthDate, char * tmajorDept,
                                char * tdegreeProgram);
    char * getDegreeProgram(){return degreeProgram;};
                                                    //W_degreeProgram=1
        bool isStaff(staff * s);
protected:
    char * degreeProgram;};

graduate::graduate(char * tname, char * tbirthDate,
       char * tmajorDept, char * tdegreeProgram){ //W_graduate=1
     name= tname;
     birthDate=tbirthDate;
     majorDept=tmajorDept;
     degreeProgram=tdegreeProgram; };

bool graduate::isStaff(staff * s){    //W_isStaff=1+2((2+3)+2)=15
        if (strcmp(name,s->getName())==0){
           bool result=s->isStudent();
           if (result)
              cout<<"This graduate student is an
employee"<<'\n';
           else
              cout<<"This graduate student is not an
employee"<<'\n';
           return(1);}
        else
           return(0);};

/*****************CLASS UNDERGRADUATE ******************/
class undergraduate:  public student
{
public:
    undergraduate(char * tname, char * tbirthDate,
           char * tmajorDept, short tclass, char * courses[6]);
    short getClass(){return sclass;};              //W_getClass1=1
    short isTakenCourse(char * course);
protected:
    short sclass;
    char * courses [6];};

undergraduate::undergraduate(char * tname, char * tbirthDate,
      char * tmajorDept, short tclass, char * tcourses[6]){
     name= tname;                              //W_undergraduate=1+3=4
     birthDate=tbirthDate;
     majorDept=tmajorDept;
     sclass=tclass;
     for (int i=0;i<6;i++)
         courses[i]= tcourses[i];};
```

```
short undergraduate::isTakenCourse(char * tcourse){
   for (int i=0;i<6;i++){         //W_isTakenCourse=1+3*2=7
      if (strcmp(tcourse,courses[i])==0)
         return true;
      }
   return false;};

/* ==================Main Program=================*/

int main ()
{
    char * courses[6];
    courses[0]="Database";
    courses[1]="OS";
    courses[2]="Programming in C";
    courses[3]="Networking";
    courses[4]="Data Structure";
    courses[5]="";
    staff * staff1 = new staff ("Aysegul Ozeke", "123456789",
                "10/05/1964", 1000, "secratery", 1);
    faculty * faculty1 = new faculty("Murat Koyuncu",
         "987654321", "10/04/1964", 4000, "Yardımcı Doçent");
    assistant * assistant1 = new assistant("Seda Camalan",
         "9876789", "10/04/1992", 1500, 2);
    graduate * graduate1 = new graduate("Aysegul Ozeke",
          "10/04/1995", "Computer", "Networking");
    undergraduate * undergraduate1 = new undergraduate("Can
          Kara", "10/04/1994", "Computer", 3, courses);
    cout<<staff1->getName()<<staff1->getSalary()<<'\n';
    cout<<faculty1->getName()<<faculty1->getSalary()<<'\n';
    cout<<assistant1->getName()<<assistant1->getType()<<'\n';
    cout<<graduate1->getName()<<graduate1->
         getDegreeProgram()<<graduate1->isStaff(staff1)<<'\n';
    cout<<undergraduate1->getName()<<undergraduate1->
         getClass()<<'\n';
    cout<<undergraduate1->isTakenCourse("Database")<<'\n';
}
```