International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 5 Issue 9, September-2016

Cognitive Complexity for Object Oriented Code: A Measure Perspective

Dharmbir[1], Mr. Anil Vadhwa[2]

[1]M.Tech Student, RPSGOI, Mohindergarh
[2]Assistant Professor, RPSGOI, Mohindergarh

ABSTRACT

One of the most important issues in object-oriented software is improving the quality of object-oriented design. Various object-oriented metrics were proposed to improve the quality of the software design such as the maintainability and the complexity of classes. Many software managers are advised to use metrics to identify outlying values .Software metrics is used to serve many purposes for software engineers. Many software metrics have been successfully validated theoretically and empirically as good predictors of maintance factors. The object-oriented metrics software provides information to developers and managers that is important about the object oriented structure and quality of the design and code, but without proper guidelines metrics are of little importance. Many object-oriented metrics proposed in literature lack in theoretical basis, while other has not been validated yet. This paper describes how object oriented metrics given by CK is useful to illustrate complexity of the system. Once complexity are detected, then they can be easily corrected and quality and maintenance of the software can be improved. In this paper, we have measured the cognitive class complexity by the help widely known object-oriented CK metrics. After that, we can easily calculate the maintance effort acorrect out the object-oriented design.

Keywords: CBO, DIT, LCOM, NOC, RFC, WMC

I. INTRODUCTION

The CK metrics are analyzed to assess their usefulness for practicing managers. In order to evaluate the maintainability of OO software, the quality of their design must also be evaluated using adequate quantification mean. This means that the design should be good from starting of the software development and software metrics are the tools to evaluate the quality of the design[1]. One way to calculate the complexity of the method is through traditional metrics. However, the applicability of these metrics is under several criticisms in OO code [2]. These criticisms are mainly based on lack of theoretical basis, lack of desirable measurement and mathematical properties, being insufficiently generalized or too implementation technology dependent. In our opinion, one of the most important criticisms should be the lack of features for representing the main characteristics of OO approaches. . These metrics only calculate the complexity of operations in the method, which is similar to the complexity calculation for procedural language programs, and therefore do not capture the features of OO system. It was also the case in one of our previous works [5].

This seems to be the main reason for the failure of the conventional metrics used on the method level for complexity measure of OO code. In addition, the available OO metrics do not consider the cognitive characteristics(i.e., the cognitive complexity)[3] in calculating the code complexity. The cognitive complexity is defined as the mental burden on the user who deals with the code, for example the developer, tester and maintenance staff. The cognitive complexity can be calculated in terms of cognitive weights. Cognitive weights 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[6]. A high cognitive complexity is undesirable due to several reasons, such as increased fault-pronenessand reduced maintainability. Additionally, the cognitive complexity also provides valuable information for the design of OO systems. High cognitive complexity indicates poor design, which sometimes can be unmanageable [5]. This work proposes a new metric for evaluating the design of OO code to eliminate the drawbacks given above.

The proposed metric includes the cognitive complexity of operations in a method in terms of cognitive weights. It also considers the inheritance property to be an important feature of the OO systems. With this metrics we will compare the design of the code of object oriented program and we calculate which is best design in between number of design of same code.

II. CK METRICS

Chidamber and Kemerer invented six metrics for object oriented code which are discussed below.

A. Weighted Methods per Class (WMC)

If a Class C has n methods and c1, c2 …cn be the complexity, then WMC(C) = c1 + c2 +… + cn.

The WMC metric is the sum of the complexities of all class methods. It is an indicator of how much effort is required to develop and maintance a particular class. Classes with large WMC are likely to have more faults, limiting the possibility of re-use and making the effort expended one-shot investment. Large WMC increases the density of bugs and decreases the quality of software. A class with a low WMC usually points to greater polymorphism [3].

B. Depth of Inheritance Tree (DIT)

DIT is defined as the maximum length inheritance path from the class to the root class [11]. Classes with large DIT are likely to inherit, making more complex to predict its behavior. Greater value of DIT leads to greater the potential re-use of inherited methods. Large DIT increases density of bugs and decreases the quality of software. Small values of DIT in most of the system's classes may be an indicator that designers are forsaking re-usability for simplicity of understanding. More is the depth of the inheritance tree greater will be the reusability of class and reduces coding, testing and documentation time. A class situated too deeply in the inheritance tree will be relatively complex to develop, debug and maintain. If DIT is large then testing will be more expensive. As a positive factor, deep trees promote reuse because of method inheritance. Although, inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The depth and breadth of the inheritance hierarchy are used to measure the amount of inheritance. As depth of the inheritance tree increases, the number of faults also increases. However, it's not necessarily the classes deepest in the class hierarchy that have the most faults. Most fault-prone classes are the ones in the middle of the tree. The root and deepest classes are consulted often, and due to familiarity, they have low fault-proneness compared to classes in the middle [3].

C. Number of Children (NOC)

NOC is defined as the number of immediate subclasses subordinated to a class in the class herirarchy [9]. Small values of NOC may be an indicator of lack of communication between different class designers. A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes.

This can be a sign of poor design. A redesign may be required. Greater is the value of NOC greater will be the reusability which in turn enhances productivity. This metric gives an indication of the number of direct descendants (subclasses) for each class.

Classes with large number of children are considered to be hard to maintain and thus, difficult to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and therefore must be more f h]]lexible. A large number of child classes, can indicate that base class may require more testing and there is improper abstraction of the parent class. Not all classes should have the same number of sub classes. Classes higher up in the hierarchy should have more sub-classes then those lower down. High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable [3].

D. Coupling between Objects (CBO)

A class that is coupled to other classes is sensitive to changes in those classes and as a result it becomes more difficult to maintain and gets more error prone. As Coupling between Object classes increases, reusability decreases and it becomes harder to modify and test the software system. So there is the need to set some maximum value of coupling level for its reusability. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. Excessive coupling between object classes is detrimental to modular design and prevents reuse. So, high value of CBO is undesirable. Therefore, more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has also been found to indicate fault proneness. Excessive coupled classes prevent reuse of existing components and they are damaging for a modular, encapsulated software design. To improve the modularity of a software the inter coupling between different classes should be kept to a minimum [3].

E. Response for a Class (RFC)

RFC is the no. of methods in the response set i.e. the number of methods of the class plus the number of methods called by any of those methods. RFC = |RS| where RS is the response set for the class. Response set of an object ≡ {set of all methods that can be invoked in response to a message to the object}. As RFC increases, the effort required for testing also increases because the test sequence grows. It also follows that RFC increases, the overall design complexity of the class increases. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated as there is more number of test sequences. The Response for a class is high thus increasing the testing effort, test sequence and the overall design complexity of the class. Therefore, reduce the number of operations that maybe execute in response to a message received [3]. Testing and debugging is complicated as there is more number of test sequences.

F. Lack of Cohesion of Methods (LCOM)

LCOM is defined as the measurement of the dissimilarity of methods in a class via instanced variables. Each method within a class, C accesses one or more attributes. LCOM is the number of methods that access one or more of the same attributes. If no methods access the same attributes, then LCOM=0. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. As coupling increases, reusability decreases and testing and debugging are also complicated and expensive. Although there are cases in which high value for LCOM is justifiable, it is desirable to keep cohesion high; i.e. keep LCOM low. Higher value of Lack of Cohesion in Methods increases the complexity of class design. Therefore, reduce the lack of cohesion in methods by breaking down the class into two or more separate classes. High cohesion indicates good class subdivision. Lack of Cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. It does not promote encapsulation and implies classes should probably be split into two or more subclasses. High LCOM indicates the low quality design of the software [3].

Consider a Class C1 with methods M1, M2. . . Mn.
Let {Ii} = set of instance variables used by the method Mi. There are n such sets I1 . . . In. LCOM = |the number of disjoint sets formed by the intersections of the n sets.‖
N = number of different possible pairs of methods ($N = n(n-1)/2$).
$P = |\{(m_i,m_j) : i < j \text{ and } I_i \cap I_j = \text{null}|$
$Q = |\{(m_i,m_j) : i < j \text{ and } I_i \cap I_j = \text{not null}|$.
$N = P+Q$ and $LCOM = P$

III. CCC METRICS

A. Cognitive Class Complexity

The Cognitive Complexity Metrics is used to calculate the Complexity value. After finding the Complexity value we can find the maintained effort which is required for design. When the complexity is high the testing requires more hence more effort requires.

Following are steps to calculate the cognitive class complexity of object oriented design:
1) First to calculate the cognitive weight.

2) Calculate the weighted method complexity

a) Sum of weight of each function:

$$MC_i = \sum_{i=1}^{n} w_{c_i}$$

b) Product of weight of function:

$$MC_j = \prod_{j=1}^{m} MC_i$$

c) Weight of method complexity:

$$MC = \sum_{j=1}^{q} MC_j$$

3) Calculate class complexity

4) Calculate the cognitive class complexity.

a) Cognitive class complexity of class:

$$CCC_k = \sum_{k=1}^{n} CC_k$$

b)Cognitive class complexity of object oriented program:

$$CCC = \prod_{j=1}^{m} CC_j$$

IV. EMPIRICAL STUDY

Following are the different hypothetical examples with their CK metrics value.

A. Example 1: Object-Oriented design for multilevel inheritance

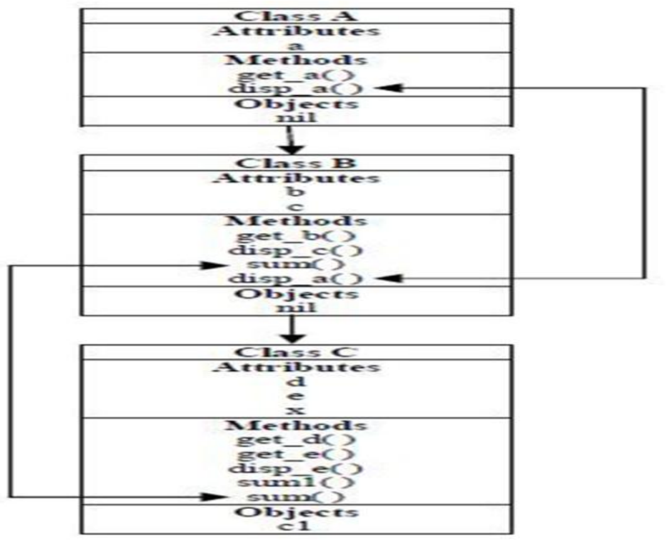


Figure I Object-Oriented Design for Multilevel Inheritance

Now Fig 1 has three classes

Step1:To calculate the cognitive weight.

Step2: Calculate the weighted method complexity

Sum of weight of each function:

$$MC_i = \sum_{i=1}^{n} w_{c_i}$$

Product of weight of function:

$$MC_j = \prod_{j=1}^{m} MC_i$$

Weight of method complexity:

$$MC = \sum_{j=1}^{q} MC_j$$

Step3: Calculate class complexity

$$CC = \sum_{P=1}^{s} MC_P$$

Step4: Calculate the cognitive class complexity.
Cognitive class complexity of class:

$$CCC_k = \sum_{k=1}^{n} CC_k$$

Cognitive class complexity of object oriented program:

$$CCC = \prod_{j=1}^{m} CC_j$$

Hence with the help of different steps we can calculate the cognitive class complexity of example I:

First to calculate the weighted method complexity

Class A has two method, CC_A=1+2=3
Class B has three method, CC_A=2+(2*2)=6
Class C has six method, CC_C=1+1+1+1+2=6

Hence we have calculate the total cognitive class

Complexity:
Cognitive class complexity:
CCC_B =3*6*6=108

A. Example 2: Object-Oriented design for multiple Inheritance

Figure II shows the Object-Oriented design for multiple inheritance and the CK metrics values for each class.

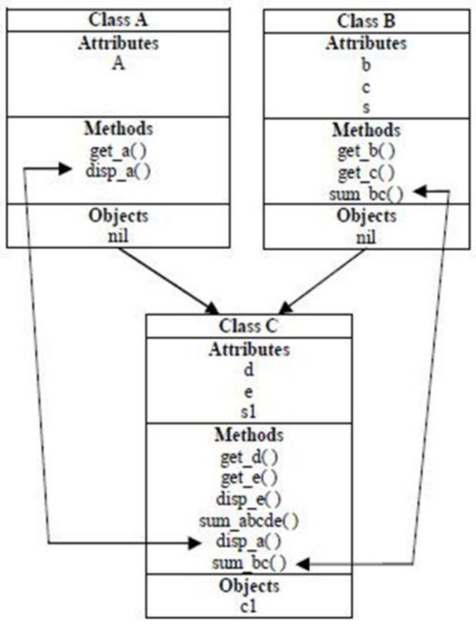


Figure II Object-Oriented Design for Multiple Inheritance

Now fig II has three classes:

Class A has two method, CC_A=1+2=3

Class B has three method, CC_B=1+1+2=6

Class C has six method, CC_C=1+1+1+1+2*2+ 2*3=14

Cognitive class complexity:

CCC_B=(3*14)+(4*14)=98

B. Example 3: Object-Oriented design for vehicle classification program

Figure III shows the Object-Oriented design for multiple inheritance and the CK metrics values for each class.

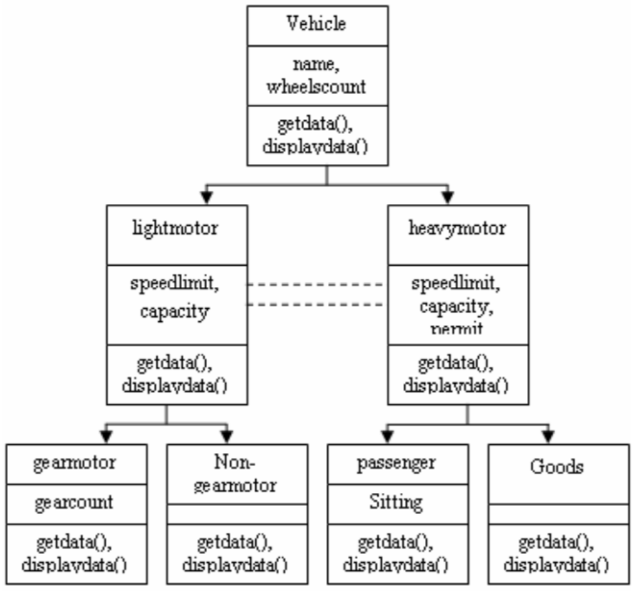


Figure III Object-Oriented design for vehicle classification program

Class vehicle has two method, CC_V=1+1=2

Class lightmotor has two method, CC_L=2+1=3

Class heavymotor has two method, CC_M=2+1=3

Class gearmotor has two method, CC_G=2+2=4

Class nongearmotor has two method, CC_{NG}=2+2=4

Class passanger has two method, CC_P=2+2=4

Class goods has two method, CC_{GO}=2+2=4

Hence cognitive class complexity

CCC_{VEH}=2*{[3*(4+4)]+[3*(4+4)]}=96

C. Example 4: Object-Oriented design for computer class

Figure IV shows the Object-Oriented design for multiple inheritance and the CK metrics values for each class.

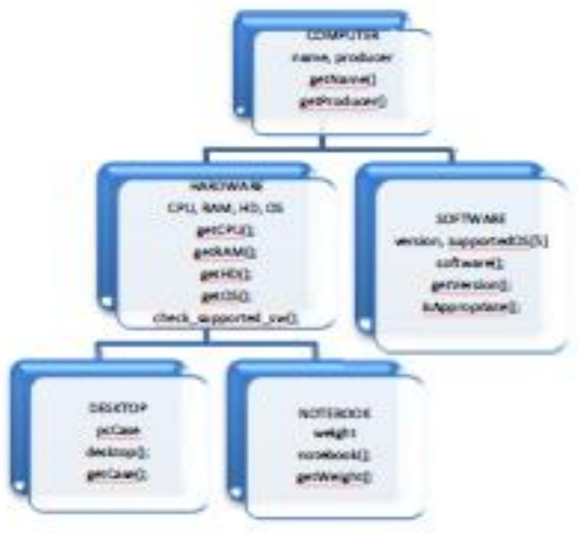


Figure IV. Object-Oriented design for computer classes model

Computer class has two method: $CC_C = W_{C1} + W_{C2} = 1+1=2$
Software class has three method: $CC_s = W_{s1} + W_{s2} + W_{S3} = 4+1+7=12$
Hardware class has five method: $CC_k = W_{m1} + W_{m2} + W_{m3} + W_{m4} + W_{m5} = 1+1+1+1+12=16$
Desktop class has two method: : $CC_d = W_{d1} + W_{d2} = 1+1=2$
Notebook class has two method: : $CC_C = W_{C1} + W_{C2} = 1+1=2$

Hence Cognitive class Complexity is:
$CCC_{COM} = 2*\{[16*(2+2)]+12\} = 152$

D. Example 5: Object-Oriented design for vehicle classification program

Figure V shows the Object-Oriented design for multiple inheritance and the CK metrics values for each class.

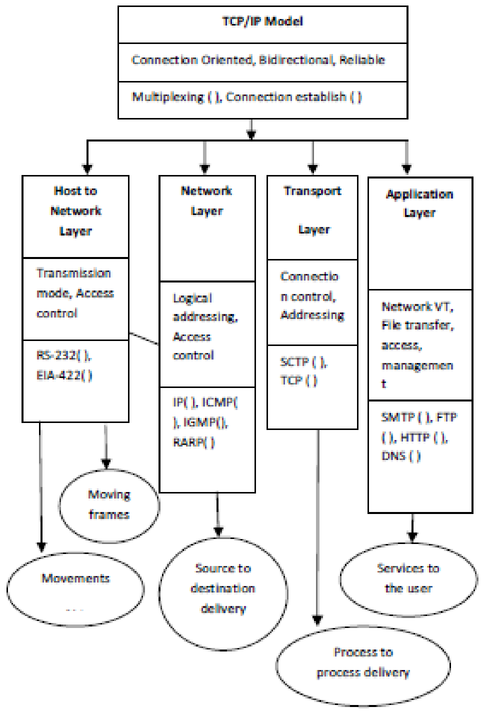
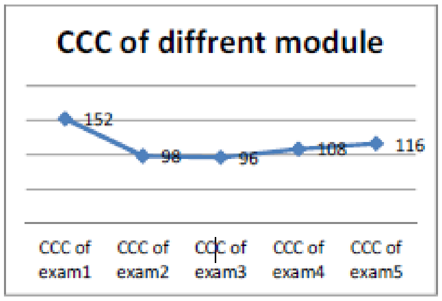

TCP/IP Model
Connection Oriented, Bidirectional, Reliable
Multiplexing ( ), Connection establish ( )
Host to Network Layer
Transmission mode, Access control
RS-232( ), EIA-422( )
Network Layer
Logical addressing, Access control
IP( ), ICMP( ), IGMP(), RARP( )
Transport Layer
Connection control, Addressing
SCTP ( ), TCP ( )
Application Layer
Network VT, File transfer, access, management
SMTP ( ), FTP ( ), HTTP ( ), DNS ( )
Moving frames
Movements
Source to destination delivery
Process to process delivery
Services to the user

Figure V. Object-Oriented design for TCP/IP classification model

Class TCP/IP has two method, CC_T=2+2=4
Class host to n/w has two method,CC_{KN} =2
Class network has four method, CC_N=1+1+1+2=5
Class transport has two method, CC_T=4+2=6
Class application has four method, CC_A=2+8+4+2=16

Hence total cognitive class complexity:
CCCTCP*(2+5+6+16)=116

Chart compares the CCC metric value in different object-oriented designs. We employed hypothetical examples to access the applicability of well-known object-oriented CCC metrics to predict the complexity in class. We can easily visualize the CCC of classes. Once we have calculated the CC with the help of object-oriented method class metric, then we can easily find out which object-oriented design that will effect on the quality and reliability of the object-oriented design and also find out which design requires less maintances effort.The design whose CCC metrics value is low requires low maintance effort and design whose complexity is high requires more maintance effort.



Line graph CCC of different module

In this chart we can easily find the CCC and with this we can compare the design.

CONCLUSION & FUTURE WORK

In this work, we have considered various hypothetical examples to access the applicability of well-known object-oriented CK metric to predict the bugs in class. Object-oriented metrics have become an essential part of object technology as well as software engineering tools exploratory analysis of empirical data is provided to relate the metrics to productivity, rework effort, and design effort on three commercial object-oriented systems. The empirical results suggest that the metrics provide significant explanatory power for variations in these economic variables, over and above what provided by traditional measures. We have visualized that DIT metric is the best metric to predict the fault-proneness of classes and it is the most useful to improve the quality and reliability of the design. The same kind of work can also be performed on large software systems i.e. industrial systems or open source systems for fault-proneness publically for the software developer community.

REFERENCES

[1]. Kaner C, “Software Engineering Metrics: What do they measure and how do we know”, Proceeded 10th International, 2004.
[2]. Kim J et al., “Cognitive processes in logical design: comparing object-oriented and traditional”, 1991.
[3]. Harrison R et al., “An evaluation of the MOOD set of Object Oriented Software”, 1998.
[4]. Henderson-Sellers B, “Object-oriented metrics, measure of complexity”, (Englewood Cliffs, N.J.: PTR, Prentice-Hall), 1996.
[5]. SANJAY MISRA et al., **“**An inheritance complexity metric for object-oriented code:A cognitive approach” ,MS received 25 April 2010.
[6]. SANJAY MISRA et al., “Weighted Class Complexity: A Measure of Complexity for Object Oriented System”, JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 24, 1689-1708 (2008).
[7]. A. Aloysius et al., “Maintenance Effort Prediction Model Using Cognitive Complexity Metrics ” St. Joseph’s College, Tiruchirappalli – 620002, India.
[8]. Linda H. Rosenberg, “Applying and Interpreting Object Oriented Metrics”, Track 7 - Measures/Metrics.
[9]. Muktamyee Sarker, “An overview of Object Oriented Design Metrics” June 23, 2005.

[10]. Basci D et al., “Measuring and Evaluating a Design Complexity Metric for XML Schema”, 2009.
[11]. Basci D et al., “Data Complexity Metrics for Web-Services. Advances in Electrical and Computer”, 2009.
[12]. Briand L C et al., “Modelling development effort in object oriented system using design properties”,2001.
[13]. Chidamber S R et al., “A metrics suite for object oriented design. IEEE Transactions on”, 1994.
[14]. Sanjay Misra, et al “Software Eng. 6: 476–493336”.
[15]. Gupta V et al., “Package coupling measurement in object-oriented software. J. of Computer”, 2009.
[16]. Yuming Zhou et al, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults ”, 2006.
[17]. Chidamber Shyam et al, ” Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis ”,2003.
[18]. Rakesh kumar et al, “A Heuristics Based Review on CK Metrics”.
[19]. Yuming Zhou et al, “Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults”, March 5,2008.
[20]. R. Subramanyam et al, “Empirical Analysis of CK Metric for Object-Oriented DesignComplexity Implications for Software Defects ”.