**Coupling Complexity Metric: A Cognitive Weight of Polymorphism, Encapsulation, Method Hiding and Attribute Hiding**

IT20272104 -Dissanayaka D.D.D

IT20116606 -Fernando W.H.K

IT20136710 - Kariyapperuma K.M.K.R

IT20258030 -Gunawardhana H.P.M.N

**Abstract –** As the paradigm gains traction, analyzing object-oriented systems to assess its own quality becomes extremely important. Including class coupling, many object-oriented metrics have been introduced. The Whole research provides a latest cognitive complexity metric like Cognitive Weighted Coupling complexity between objects (CWCCBO) between objects and it is analyzing the coupling on object-oriented system and new 4 factors associated with object-oriented concepts. Global data coupling, control coupling, data coupling, internal data coupling, and lexical content coupling are the five coupling types within classes considered when computing CCWCBO. Polymorphism, method hiding, and attribute hiding are the other three couplings. Another object-oriented strategy is encapsulation. Here we introduce calculations to calculate the complexity based on four new factors in addition to the existing ones, which are given below in detail. The Cognitive Weight of Polymorphism Factor (CMOPF), Cognitive Weight of Encapsulation Complexity Metrix (CWECM), Cognitive Weight of Method Hiding Factor (CWOMHF), and Cognitive Weight Attribute Hiding Complexity Metrix (CWAHCM) are the four factors. CWOPF and CWECM are derived as object-oriented factors and CWOMHF and CWAHCM as non-object object-oriented factors.

# 1.Introduction

The code's attempt to enable functional complexity is a natural byproduct of software complexity. The complexity of software systems can sometimes grow out of control due to multiple system interfaces and complex requirements, making applications and portfolios overly expensive to maintain and risky to enhance. If left unchecked, software complexity can spiral out of control in completed projects, resulting in bloated, inefficient applications. As for the benefits, it can be difficult and time consuming to identify the architectural hotspots where risk and cost originate without the use of reliable software complexity metrics. More importantly, continuous software complexity analysis allows project teams and technology management to stay ahead of the problem and avoid the emergence of excessive complexity.

Based on the foundation of the cognitive materials that "cognitive complexity of software is based on three primary factors, named inputs, outputs, and internal processing," cognitive complexity measures endeavor to estimate the effort or degree of deprivation in containing the software. A model that represents the program's executive steps can be defined as its mathematical model. Here, depending on the change in the value of a variable, it is possible to determine the number of variables in the complexity of the information. Information complexity is defined as the number of variables it contains and the information that exists within that program. This is based on a mere assumption. That is that the variables and their executors also contain some information. However, cognition in a program can be defined as cognition included in a function, module, class, file, and clear idea of a program. This is implemented through source code, which becomes the cognition of the source code.

Object orientation is now a widely used approach in software engineering, as software developed using this technology is not difficult to maintain. Dynamic binding- cover-, heritage. Interaction-, polymorphism, and reusability are powerful features of object-oriented (OO) software development. "Chidambaram - Kemmer" metric set [1], MOOD metrics for OO design [2]. Design parameters for experiments [3], Product parameters for object-oriented design [4], Lorenz and Kidd Metrics [5]. Henderson-Seller et al. Parametric [6, (slightly) modified CK metrics [7]. And estimating the size of OO systems [8] "are some of the proposed parameters for evaluating 00 software.These parameters often focus on specific stages of software development, such as design and testing. They also cover certain OO language features and quality attributes, including inefficiency, error-freeness, integrity-flexibility, and maintainability, interoperability, and reliability- Portability, Reusability- usability and testing capability [9]. Maintenance is by far the most important thing of a software system. Complexity parameters used to predict information to determine the reliability-sustainability of software systems through automated source code analysis [11]. Here we have studied two object-oriented concepts and two methods of concealing polymorphisms and combinations and object-oriented factors and concealing properties through the concept of cognitive weight. Based on the data obtained, we presented four new equations separately. For some equations, certain constants were used based on the available data. We have shown the relevant variables in the equations separately.

The aim of this research is proposing improvements to AA CWCCBO measure (Cognitive Code-Level OO Complexity Measure). This includes both theoretical and empirical validation, as well as a comprehensive comparative analysis. The rest of this paper is structured as follows: Body consists with six sections. They are CCBO (coupling complexity between object), Cognitive Weighted Coupling Complexity Between Objects (CWCCBO), Cognitive Weight Of Polymorphism Factor (CWOPF), Cognitive weight of Encapsulation complexity Metrix (CWECM), Cognitive Weight Of Method Hiding Factor (CWOMHF), Cognitive weight Attribute hiding complexity Metrix (CWAHCM).After the body section there is an analytical description of the conclusion. Finally, under the references, we stated quotations we use to find the information.

# 2.BODY

## **Coupling complexity between object (CCBO)**

Addition of a class between objects is a calculation of how many additional classes it is associated with. The answer to CCBO can only be found in external connections to foreign classes. This study will also experiment with different alternative values. Various variant couplings have been proposed by Edward Beard [12], which are defined below:

1. Control Coupling: A phenomenon that sends control flag modules into content. One module can control the processing steps that exist in another module.
2. Global Data Coupling: Sharing two or more modules of the same global data structure, this is called global data coupling.
3. Connecting internal data: In a module where local data exists, that data is directly modified by another module.
4. Link data: The output of one module becomes the input of another module.
5. Lexical Content Coupling: The contents of 1 module are partially or completely included within the contents of another

## **Cognitive Weighted Coupling Complexity Between Objects (CWCCBO)**

"Connecting unwanted objects reduces the reuse of linked objects," and "Connecting unwanted objects increases the risk of system corruption when one or more linked objects are changed. CWCCBO can be used to calculate using the following equation:

CCT-Control Coupling of total of modules.

CGDC-Count of Global- Data Coupling

CIDC- Count of Internal -Data- Coupling

CDC-Count of Data- Coupling

CLCC-Count of Lexical- Content- Coupling

CWFOC-Control Weighting Factor Of Coupling

WCOGDC-Weighting Count of global- data coupling

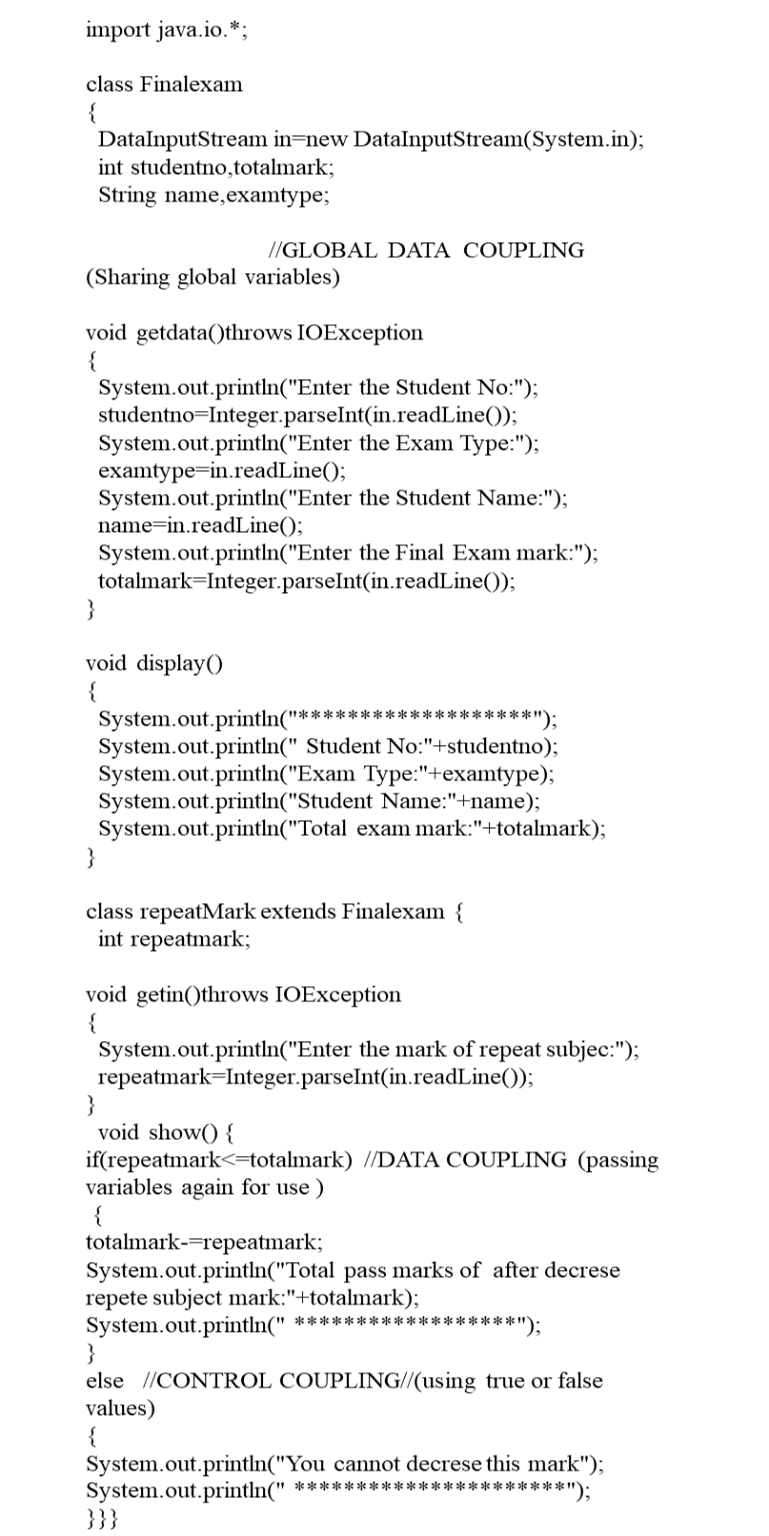
WCOIDC-Weighting Count of internal -data - coupling

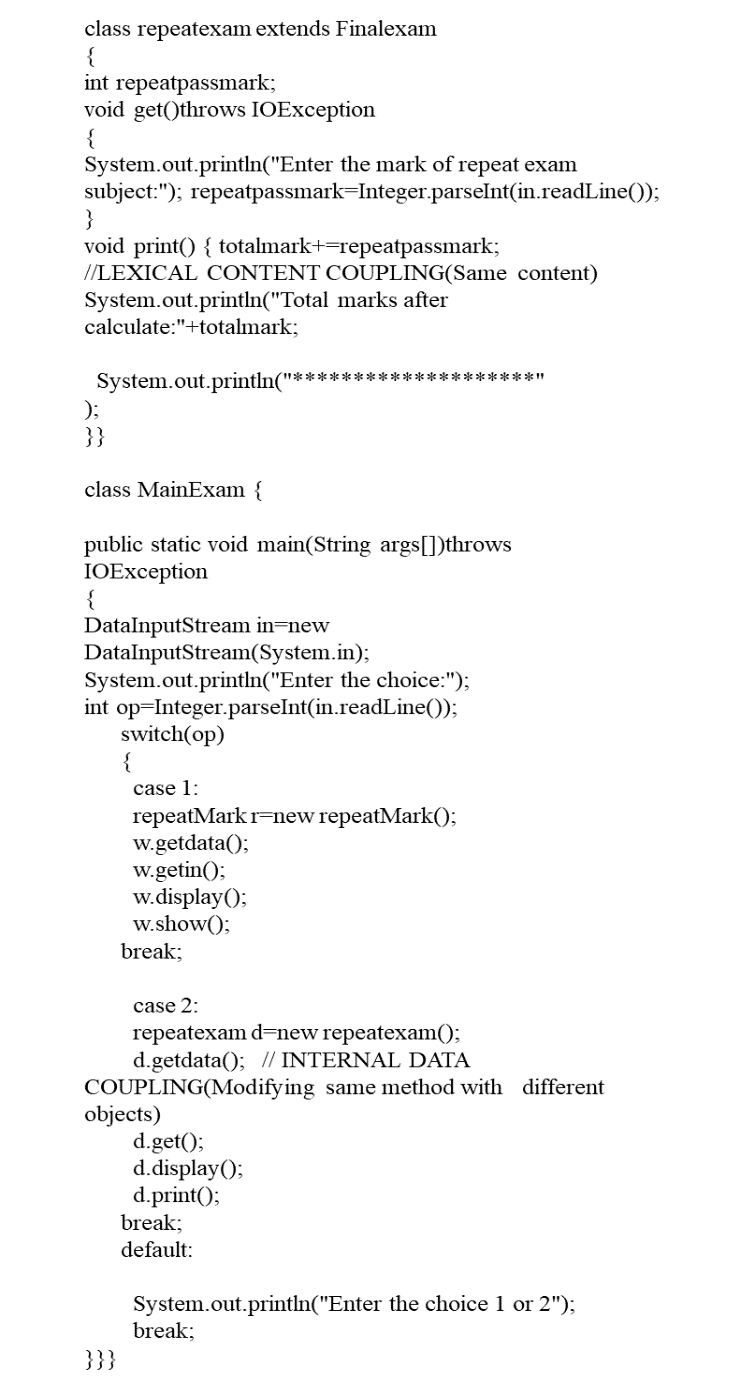
WCODC- weighting Count of data coupling

WCOLCC- weighting Count of lexical -content - coupling

Using an existing comparison test some weighting factors values can be taken as follows,

|  |  |
| --- | --- |
| WCOGDC | 1 |
| WCOIDC | 2 |
| WCODC | 3 |
| WCOLCC | 4 |





CCBO=1+1+1+1+1

=5

CWCCBO=(1\*1)+(1\*1)+(1\*2)+(1\*3)+(1\*4)

=1+1+2+3+4

=11

Let’s discuss the four new factors that we have introduced.

## **Cognitive Weight Of Polymorphism Factor (CWOPF)**

There are several object-oriented concepts.  They are encapsulation, polymorphism, aggregation, abstraction so on.  Among them, polymorphism is one of the important object-oriented concept. polymorphism means the possibility of a programming language to manipulate objects in a variety of ways based on its data types and classes.

In other words, polymorphism can be defined as the possibility to re-define or re-enable classes and interfaces that are already derived. It has been shown to enhance the possibility to extend and reuse [13]. Polymorphism's importance arises from the several advantages it provides in the software development life cycle, allowing for the creation of qualitatively efficient and successful software. Polymorphism has many benefits. The primary benefit of this is that the process inside the interface is hidden from the client, resulting in greater freedom as well as disassociating

**Pure-Polymorphism:** It is accomplished by calling the similar person functionality with various signatures within the sole class scope. Number, type, and order of the arguments make the signature. It's the same as overloading methods within a class. Parametric overloading is another name for it.

**2) Static-Polymorphism:** In a class hierarchical tree, it is composed level polymorphism. It's the same as overriding a method. It is accomplished by defining person functionalities with the similar name but distinct signatures in separate types. Inheritance relations may or may not connect these classes. Method binding occurs at binding period for both pure and static polymorphisms.

**3) Dynamic-Polymorphism:** It's also known as composition level polymorphism or method overriding. In contrast to static polymorphism, it is accomplished by defining person functionalities in child classes with the same signature and name. it's the possibility to use the similar name and signature previously used in an overridden method [14].

The following formula can be used to calculate the complexity metrics of the polymorphism factor,

Number of overriding-methods in a class ,

= Number of new-methods defined in a class ,

= Number of child classes in a class ,

= Total number of Classes in a program.

Using the above formula, we propose the following formula to calculate the complexity using the cognitive weight of the polymorphism factor. In addition to the architectural complexity, using the CWOPF proposed here allows the software to identify the various types of polymorphism that exist in the software and thereby calculate its cognitive weight complexity.

= Number of overriding-methods in class(

= Number of child classes in a class

TCN = Total number of Classes in a program.

A.C.W = () / 3

= Quantity of pure-polymorphism

= Quantity of static-polymorphism

= Quantity of dynamic-polymorphism

= Cognitive weight of pure-polymorphism

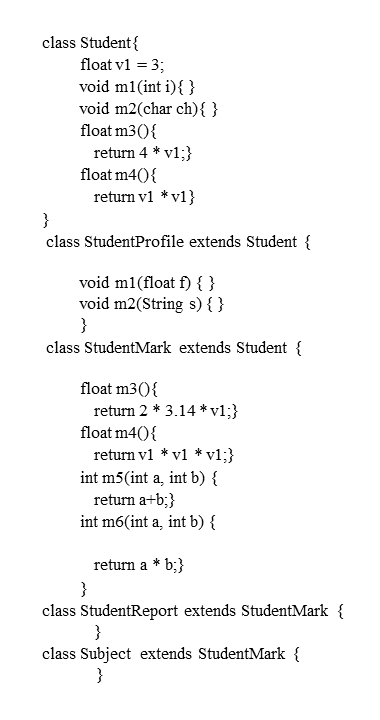
= Cognitive weight of static-polymorphism

= Cognitive weight of dynamic-polymorphism

Using an existing comparison test some cognitive values can be taken as follows,

|  |  |
| --- | --- |
| Pure-polymorphism (Pp) | 3 |
| Static-polymorphism (Sp) | 5 |
| Dynamic-polymorphism (Dp) | 7 |

Considering the following code snippet let’s calculate the value of CWOPF,



## **Cognitive weight Of Encapsulation complexity Metrix(CWECM)**

The object-oriented language was used as a very important languages of the 20th century. Facilitates o-o language heritage, abstraction, closure, linking, polymorphism, and integration, enhancing the ability to reuse, test, maintain and extend the software. encapsulation will become very popular among other 00 developers. Heritage and Contact Software Development Lifecycle reduces the impact of coverage on software. Based on a literature survey of software engineering [15-16] it can be suggested that prompt debugging is essential for the success of a reliable software system. Table 1 of this paper illustrates the design effect on complexity and reliability. The arrow with the highest score shows more impact and the lower arrow shows minimum effects. Focusing on o-o design structures like proposed access, legacy, connection, integration, and integration contributes minimizing system complexity. [17].

Complexity increases as design construction decreases. The complexity is therefore inversely proportional to the combination. Reliability increases when the design capsule is maximized. Reliability is a qualitative quality that is closely related to object-oriented design construction. Accordingly, an assumption can be suggested based on the following discussion: As the capsule increases, the complexity decreases. The depth of a software is inversely proportional to its acceptance. If the legacy and relevance of the class structure is high, the complexity will undoubtedly be a high and low reliability system. In practice, as acceptance for something increases, the system becomes more erroneous, unreliable and difficult to understand. The existing complexity of the system lowers the reliability of the system. Encapsulation can be defined as a process that minimizes interdependence between separate written modules by defining rigid non-intrusive interfaces.

In a class plan, its complexity seems to depend on qualities and methods. They are more complex than a class with a large number. It is difficult to understand. The following equation is designed to try to control the complexity of a design by increasing the capsule of a class design. . CWECM enables it to be calculated

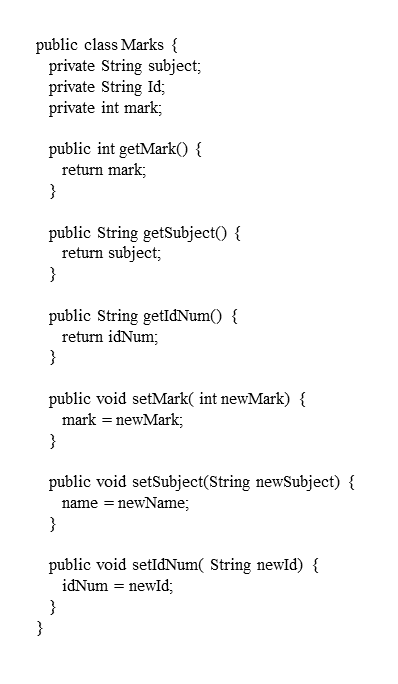
M – methods

A- Attributes

En (EM + EA)- methods and attributes of encapsuled

DA (M +A)- methods and attribute for direct accessible

S (M+A)-count of all attribute and methods



## **Cognitive Weight Of Method Hiding Factor (CWOMHF)**

The method hiding factor is a non-object-oriented component that is included in the information hiding. Because information hiding and encapsulation are not the same thing [19]. Although these instance variables and methods are already encapsulated, they may sometimes be visible to other classes. This visibility can be fully or partially. This information hiding property is a combination of encapsulation and the scope of the attributes and methods within the class. Method hiding must be done with caution, since less method hiding can result in an implementation that isn't properly abstracted, while excessive method hiding can result in very little functionality. As a result, the method hiding factor must be chosen very precisely to maintain and keep a good balance between the hidden and already visible methods in the system. Let's take this as Cognitive Weight Of Method Hiding Factor (CWOMHF) .

The following formula can be used to calculate the complexity metrics of the method hiding factor

Md(Ci) = Mv(Ci) + MH(Ci)

Md(Ci) = Total number of methods in class Ci

Mv(Ci) = Number of visible methods in class Ci

Mh(Ci) = Number of hidden methods in class Ci

TCN = Total number of Classes in the whole system.

Using the above formula, we propose the following formula to calculate the complexity based on the cognitive weight of the method hiding factor. The CWOMHF complexity metrics consist of cognitive complexity by taking the various types of method visibility. Visibility might be completely hidden, slightly visible, or totally visible. In Java, there are different types of visibility methods. Such as 'Private,' 'protected,' 'public,' and the default,' for example. The method only visible within a relevant class can name as private, and a method visible to the entire system can be is defined as public. The function is visible both within the class and its inheriting classes, as well as within the package and other packages, which can be named as a protected method.

The derived formula for the CWOMHF can be mathematically defined as follows,

Mp(Ci) = Number of private methods in class Ci

Md(Ci) = Number of default methods in class Ci

Mt(Ci) = Number of protected methods in class Ci

Mu(Ci) = Number of public methods in class Ci

CWpm = Cognitive Weight of private methods

CWdm = Cognitive Weight of default methods

CWtm = Cognitive Weight of protected methods

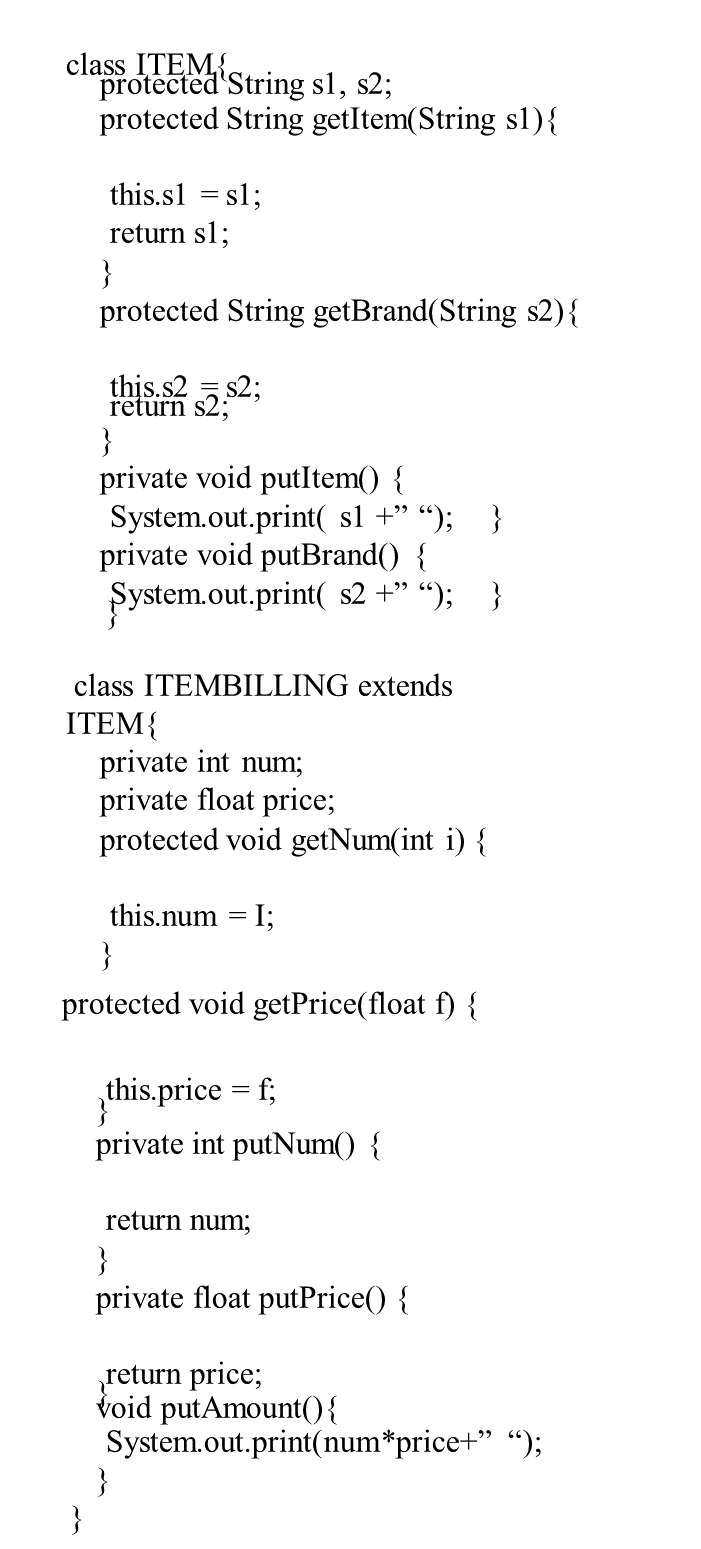
CWum = Cognitive Weight of public methods

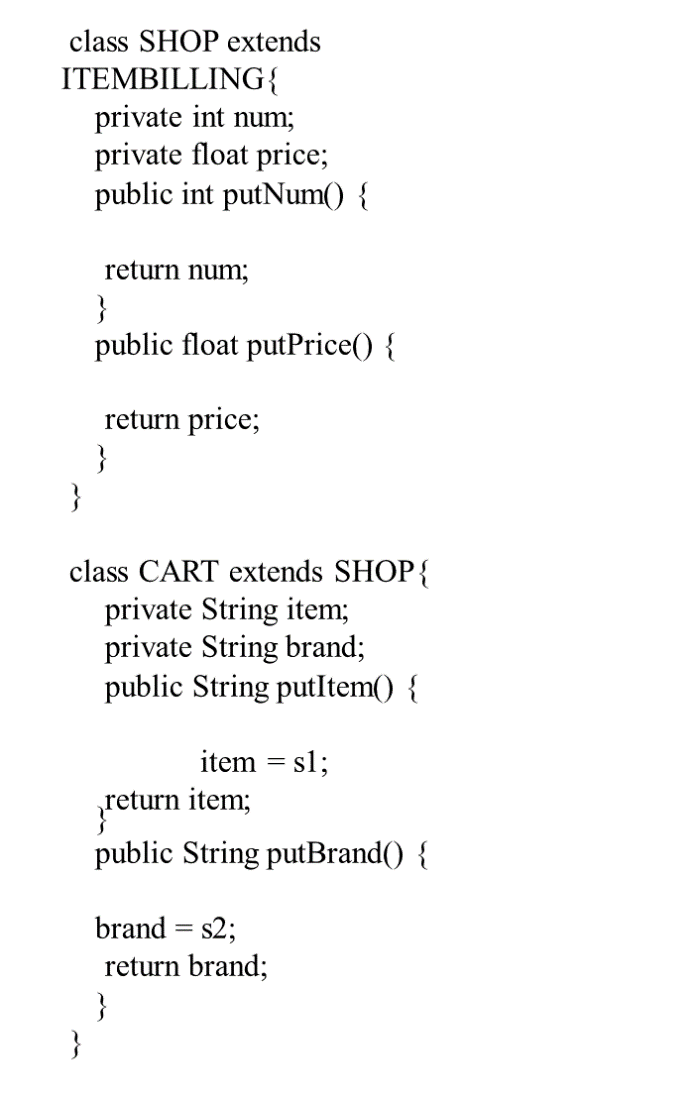
= Total number of Classes in the whole system

Using an existing comparison test some cognitive values can be taken as follows,

|  |  |
| --- | --- |
| Private Method (PM) | 4 |
| Private Method (PM) | 5 |
| Protected Method (TM) | 6 |

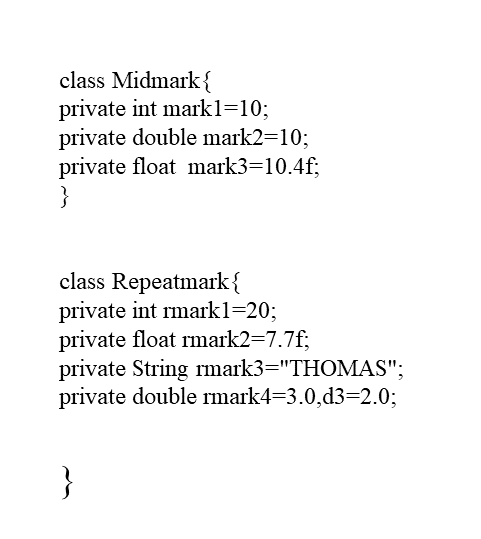
Considering the following code snippet let’s calculate the value of CWOMHF,





## **Cognitive weight Attribute hiding complexity Metrix(CWAHCM)**

The success of a software product depends largely on the concealment of data protection properties against any random or unwanted changes. The ability to hide attributes is achieved by encapsulating data value-holding variables. Typically, the 'class' block is used to encapsulate. But capsulation and concealment of information are not the same things. information can be hidden without engraving, and vice versa [20]. Therefore, attribute concealment can actually be activated by a combination of class coverage and scope of attributes. AHCM: - The ratio of the sum of the invisibility of all the properties defined in all classes to the total number of properties defined in the software system. Validation Complex Metrics The CWAHCM is derived from the AHCM Complex Metrics, which considers only the structural mechanism of the OO paradigm and does not include the cognitive component. By combining both operational and architectural complexities, interpretive complexity can thus add cognitive complexity to the CWAHCM AHCM metric. According to an experiment conducted by Francis Thamburaj and Aloysius, the cognitive load (CW) [21] has already been calibrated. Accordingly, the cognitive burden for a private attribute is 2; For the default attribute 3; 4 for the security attribute, and 1 for the general attribute. The complexity is calculated by substituting these CW loads for this derived formula.



# 3.Conclusion

A CWCCBO metric has been developed to assess class level complexity. The term "class complexity" refers to the complexity of a class. The cognitive complexity of various types of connections is included in CWCCBO. CWCCBO shown, the cognitive weight of various types of relationships are affected by class complexity. A Cognitive Test was used to validate the cognitive loads given by the four coupling types, and discovered that a cognitive loads required to comprehend a CLCC were greater than those CGDC, CCT, and CGDC. The measure was tested using comparative studies and case studies, and that was the strong indicator for CLC which is Class-Level Complexity. Calculated using the technology developed by CWCCBO. To measure class-level complexity, the Cognitive Weight of Polymorphism Factor (CWOPF) has been introduced. CWOPF catches both structural and cognitive complexity. Collection of metric comprehension tests were used to calculate the new polymorphism complexities. CWECM suggested as a mechanism to evaluate the complexity of the class plan. CWECM introduced that high coverage reduces a complexity of class design. As a result, it concludes, the complexity of object-oriented software can be retained by expanding encapsulation. This paper proposes and mathematically defines a new complexity metric that is the concealing factor of the Cognitive Weight System for measuring class-level complexity. The cognitive system's hidden factor is related to structural complexity, and moreover the cognitive complexity deriving from the effort and time required for comprehend the software. Collection of cognitive tests were used to measure cognitive load, and that measure was discovered that cognitive load varies sequentially from the default, default and private domains to the different domain scope used to hide system visibility in other classes. The new CWOMHF complexities are meticulously broader in nature and more realistic than realistic. The CWAHCM Complexity Metric has shown a better and broader index. The calculations related to the equations obtained using case studies are performed and final conclusions are reached.

# 4.Reference

[1] S. R. Chidamber and C. F. Kemerer, ‘‘A metrics suite for object-oriented design,’’ IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 476–493, Jun. 1994.

[2] R. Harrison, S. J. Counsell, and R. V. Nithi, ‘‘An evaluation of the MOOD set of object-oriented software metrics,’’ IEEE Trans. Softw. Eng., vol. 24, no. 6, pp. 491–496, Jun. 1998.

[3] R. V. Binder, ‘‘Design for testability in object-oriented systems,’’ Comms. ACM, vol. 37, no. 9, pp. 87–101, Sep. 1994.

[4] S. Purao and V. Vaishnavi, ‘‘Product metrics for object-oriented systems,’’ ACM Comput. Surveys, vol. 35, no. 2, pp. 191–221, Jun. 2003.

[5] M. Lorenz and J. Kidd, Object-Oriented Software Metrics: A Practical Guide. Upper Saddle River, NJ, USA: Prentice-Hall, 1994.

[6] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, ‘‘Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design),’’ Object Oriented Syst., vol. 3, no. 3, pp. 143–158, Sep. 1996.

[7] V. R. Basili, L. C. Briand, and W. L. Melo, ‘‘A validation of object-oriented design metrics as quality indicators,’’ IEEE Trans. Softw. Eng., vol. 22, no. 10, pp. 751–761, Oct. 1996.

[8] G. Costagliola, F. Ferrucci, G. Tortora, and G. Vitiello, ‘‘Class point: An approach for the size estimation of object-oriented systems,’’ IEEE Trans. Softw. Eng., vol. 31, no. 1, pp. 52–74, Jan. 2005.

[9] S. L. Pfleeger and J. M. Atlee, Software Engineering: Theory and Practice. Chennai, India: Pearson Ed., 1998.

[10] I. Sommerville, Software Engineering. Reading, MA, USA: Addison-Wesley, 2004.

[11] T. J. McCabe and A. H. Watson, ‘‘Software complexity,’’ Crosstalk, vol. 7, no. 12, pp. 5–9, 1994.

[12] Edward Berard. V ―Essays on object-orientedsoftware engineering (vol. 1)‖ Berard Software  
Engineering, Prentice-Hall, 1993, ISBN:0-13-288895-5, 1993

[13] T. G. Mayer, T. Hall, “Measuring OO systems: a critical analysis of the MOOD metrics,” Tools 29, (Procs. Technology of OO Languages & Systems, Europe’ 99), R. Mitchell, A. C. Wills, J. Bosch, B. Meyer (Eds.): Los Alamitos, Ca., USA, IEEE Computer Society, pp. 108–117, 1999.

[14] S. Benlarbi, and W. L. Melo, “Polymorphism measures for early risk prediction,” IEEE Software Engineering, 1999. Proceedings of the 1999 International Conference, pp. 334-344, 1999.

[15]. S.Singh, K.S. Kahlon, P.S. Sandhu, “Re-engineering to analyze and measure object oriented paradigms ”, The 2nd IEEE  
International Conference on Information Management and Engineering , **ISBN:** 978-1-4244-5263-7,16-18 April 2010,p: 472 –  
478,DOI: 10.1109/ICIME.2010.5478013 .  
[16]. N.Sharygina, J. C. Browne, R. P. Kurshan, “A Formal Object-Oriented Analysis for Software Reliability: Design for  
Verification”, Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering ISBN:3-  
540-41863-6, 2011, p: 1-15.

[17] Yong Cao Qingxin Zhu, “Improved Metrics for Encapsulation Based on Information Hiding”, DOI: 10.1109/ICYCS.2008.76,  
The 9th International Conference for Young Computer Scientists, IEEE computer society, 2008, p: 742-724.

[18] Nathanael Sch aril, Andrew P. Black,and St´ephane Ducasse, “ Object oriented Encapsulation for Dynamically Typed  
Languages”, OOPSLA’04 Vancouver, BC, Canada, p. 130–139, Oct. 2428, 2004, ACM 1581138318/04/0010.

[19]Tahvildari, Ladan, and Ashutosh Singh, “Categorization of object-oriented software metrics,” Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, Halifax, Nova Scotia, pp. 235–239, May 2000.

[20] Harrison, R., Counsel, S.J. and Nithi, R.V., “An Evaluation of the MOOD Set of Object-Oriented Software Metrics,”IEEE Tr.on Software Eng., 6, 491-496, 1998

[21] Francis Thamburaj and A. Aloysisus, “Cognitive Perspective of Attribute Hiding Factor Complexity Metric,”InternationalJournal of Engineering and Computer Science, ISSN: 2319-7242,11, 14973-14979, November 2015