Improvements for Object-Oriented Cyclomatic Complexity Metrics Based on Cognitive Weights

D.I. De Silva, K. Rajendran, Rahman S.H., Thiranya M.A.R, Jayasinghe H.M.C.P, Rajapaksha R.M.
Department of Software Engineering

Abstract - ***The role of Object Oriented Programming (OOP) in today's software development and architectures makes Code Complexity and Object Oriented Programming (OOP) an important issue. Implementation, Polymorphisms, Abstraction etc. are essential design ideas in OOP that affect coding architecture, organization, and design. Complexity metrics are used to forecast vital aspects regarding software systems' operation and maintenance. Software metrics must be approached differently in object-oriented software development. In this paper, an attempt has been made to suggest improvements to Misra S.'s study of object-oriented complexity metrics based on cognitive weights, and how examples and calculations can be used to define the software complexity of a system, evaluate the proposed improvements with its parameters to establish well-proposed metrics.***

Keywords—*Cyclomatic Complexity, Cognitive weights, Object-oriented, encapsulation, polymorphism, method chaining, class complexity*

I. INTRODUCTION

The complexity of software systems can occasionally grow out of control due to various system interfaces and complex requirements, making applications and portfolios too expensive to maintain and too dangerous to improve. If left uncontrolled, software complexity can spiral out of control in completed projects, resulting in bloated, inefficient programs. "The process of maintaining software necessarily destroys it," says Alain April, an expert in the field of IT maintenance.

Cyclomatic Complexity evaluates the amount of control structure in a program, such as in RPG operation codes. Because conditional logic makes programs more difficult to understand, evaluating cyclomatic complexity reveals how much has to be managed. However, using cyclomatic complexity alone can lead to incorrect answers. A module can be complicated, but it rarely interacts with other modules. A module can also be reasonably simple but extensively connected with numerous other modules, significantly increasing the codebase's total complexity. Complexity metrics will look awful in the first situation. The complexity metrics will appear to be good in the second, but the outcome will be misleading. To achieve a genuine system-level, software complexity assessment, it's also necessary to assess the coupling and cohesion of the modules in the codebase.

In this research, a variety of cognitive code-level CCL, object-oriented (OO) complexity measures have been proposed in the literature [1], [2], [3]. In addition, a survey on existing CCL OO complexity measures can be found in [4].

According to Misra S.'s study of 'An Object Oriented Complexity Metric Based on Cognitive Weights' [8], Complexity Measures have been described to calculate the complexity of an object oriented code with cognitive weight. Cognitive weights are defined as the level of difficulty or the amount of time and effort necessary to interpret a piece of software that meets the complexity criterion. However, the proposed complexity metrics for object oriented systems by using cognitive weights has room for improvements with the measure of complexity measures. In this paper, improvements to calculate the cyclomatic complexity for object-oriented metrics based on cognitive weights is being proposed.

In the following section 2, the proposal for cognitive code level object-oriented metrics is provided, along with four types of complexities: Method and attribute complexity in section 2A, Polymorphism metric in section 2B, encapsulation metric in section 2C and method chaining metric in section 2D. The conclusion is provided in section 3.

II. PROPOSED COGNITIVE CODE LEVEL OBJECT ORIENTED METRICS FOR COGNITIVE WEIGHTS

A. Method and Attribute Complexity

The Method and Attribute Complexity factor examines Object-Oriented Programming's more general properties. Attributes and methods are found in a general concrete class. It calculates the complexity of the class codebase by looking at the type of properties and method implementation. The Method and Attribute Complexity is made up of two parts. [3]

1. Weight due to attribute types of the class (Wat)

1

2. Weight due to method types of the class (Wmt)

When the weight due to the attribute types is equal to Wat and weight due to the method types is equal to Wmt, The Method and Attribute Complexity (Wcc) is given by the following equation:

$$W_{cc} = W_{at} + W_{mt}$$

W_{cc} = Class Complexity
W_{at} = As a result of the attribute kinds, there is a certain amount of weight.
W_{mt} = Due to the various method types, there is a certain amount of weight.

a) Weight due to attribute types (Wat)

A class can have both primitive and non-primitive types of properties in object-oriented programming languages. Attributes are used to store the class's properties. Depending on the type of attribute, the code complexity varies. When compared to primitive types, utilizing data structures adds greater architectural and psychological complexity. As a result, class attributes are divided into three categories based on their complexity level.

1. Primitive Types
2. Objects
3. Data Structures

The components that fall under these categories may differ depending on the programming language used. As a result, the Java programming language will be explored for a more detailed explanation of the metric. In Java, there are eight primitive types as byte, short, int, long, float, double, char, boolean. Other defined class objects could be classified, as objects including Strings. Arrays, Lists, Sets, Ques, etc. fall under the data structures. The following is the given weights for each category.

Attribute Type	Weight	
Primitive Type	P_w	1
Objects	O_w	2
Data Structures	D_w	3

The following is the equation to calculate the attribute type complexity metric:

$$W_{at} = (P_{ac}P_w) + (O_{ac}O_w) + (D_{ac}D_w)$$

In the above,
Wat = The weight due to the attribute type
Pac = Primitive type attribute count
Oac = Object type attribute count
Dac = Data structures type attribute count

b) Weight due to method types (Wmt)

To specify the functionality of the classes, methods are introduced. A class can have as many methods as it wants. As the number of methods in a class grows, so does the code complexity. Methods differ from one another in terms of the number of parameters, parameter types, and return kinds. Depending on how the methods are implemented, they add varying amounts of complexity. As a result, each method's weight is assigned using this complexity metric.

When the weight due to the parameter type equal to Wpt, weight due to the control structures equal to Wcs and number of methods of the class equal to n, The weight due to the method types of the entire class (Wmt) is given by following formula.

$$W_{mt} = \sum_{i=1}^{n} (W_{pt})_i + (W_{cs})_i$$

In the above equation:
Wmt = The weight due to the method types of the entire class
Wpt = Weight due to the parameter types
Wcs =Weight due to the control structures
n = Number of methods of the class

B. Polymorphism Metric

Polymorphism refers to an object's ability to take on multiple forms. When a parent class reference is used to refer to a child class object, polymorphism is most commonly utilized in OOP. It is a basic notion in object-oriented programming (OOP) that describes instances when something appears in several forms.

A proposed new metric for polymorphism will capture the object-oriented complexity using the polymorphic complexity metric defined by Abreu et al. He defines the PF complexity metric as a quotient ranging from 0% to 100% in order to compare and derive conclusions among systems with different sizes, implementation types, system complexities and domain types [1].

The polymorphism metric can be calculated using the following:

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) * DC(C_i)]}$$

In the above,
M0(Ci) represents the number of overriding methods, in the class Ci,
Mn(Ci) represents the number of newly created methods that are made in class Ci
DC(Ci) represents the number of children in the class Ci
TC represents the total amount of classes in the system application,
The proposed Cognitive Weighted Polymorphism Factor assigns distinct cognitive weights based on the three types of polymorphism outlined above. [2]
Case Study Program:

```
class A {
    float v1 = 5;
    void m1(int i) {}
    void m2(string s) {}
    void m3 (float i){}
}
class B extends A {
    @override
    void m1 (float f) {}
    @override
    void m2 (string s) {}
    @override
     void m3 (float i) {}
}
```

Below, we are applying Abreu's method as follows:

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[Mn(C_i)*DC(C_i)]}$$

$$PF = \frac{M_o(C1) + M_o(C2)}{Mn(C1)*DC(C1) + Mn(C2)*DC(C2)}$$

$$PF = \frac{0 + 3}{(2 * 1) + (0 * 0)}$$

$$PF = 1.5$$

According to the calculation, the polymorphism factor of this program is 1.5.

C. Encapsulation Metric

In object oriented programming Encapsulation is another key factor that uses data hiding by wrapping up the methods and attributes under a single unit.
Encapsulation is a feature of object-oriented design. Only when the amount of design encapsulation is maximized does object-oriented design benefit. Data is encapsulated to protect it from illegal access. By providing rigorous external interfaces, encapsulation is an approach for limiting interdependencies among separately developed modules. Encapsulation of class design can be used to control the complexity of a class. [3]. The encapsulated class complexity metric measures two aspects: one is encapsulation, and the other is complexity. EC2M is calculated at the method and attribute levels in this case. EC2M is calculated using the following metric:

$$EC_2M = \frac{\sum_{i=1}^{n} E(m \,\&\, a) + \sum_{i=1}^{n} DA(m \,\&\, a)}{\sum_{i=1}^{n} T(m \,\&\, a)}$$

EC_2M = Complexity according to encapsulation

E(m & a) = It only contains methods and attributes that aren't utilized by any other class (encapsulated methods and attributes).

DA(m& a) = (Directly accessible methods and attributes): It only includes methods and attributes that are utilized by other classes in the same design.

T(m & a) = (Total amount of characteristics and methods): - It takes into account all methods and attributes of a class, whether they are encapsulated or accessible methods and attributes.

```
//class complexity calculation of class D
Class D{

  Private int value1;
  private double value2;

  public D(){}
  public void setvalue1(int value1){}
  public void setValue2(double value2){}
  public int getValue1(){}
  public double getValue2(){}
  public double calculate(){}

}
Class Main(){

public static void main(String args[]){
   D ob1 = new D();
   ob1.setValue1(20);
   ob1.setValue2(30000.00);
   System.out.print(ob1.calculate());
  }
}
```

The calculation for the above implementation is as follows:

$$EC_2M = \frac{\sum_{i=1}^{n} E(m \,\&\, a) + \sum_{i=1}^{n} DA(m \,\&\, a)}{\sum_{i=1}^{n} T(m \,\&\, a)}$$

$$EC_2M = \frac{2 + 4}{8}$$

$$EC_2M = \frac{6}{8}$$

$$EC_2M = 0.75$$

D. Method Chaining Metric

Method chaining, also known as named parameter idiom, is a syntactic sugar that allows you to invoke several methods in an object-oriented programming language. Each object's calling method returns the same object, which allows the calls to be chained together in a single line. The use of moderating factors is no longer necessary because of this method of invoking numerous procedures. Multiple method calls are obviously a source of increased code complexity. [5].

Example:

```
java.lang.StringBuffer obj = new StringBuffer();
obj.append("Hello World!").append("Welcome to the system").reverse().length();
```

The number of different methods invoked by one object at a time (degree of methods) is regarded as the major characteristic in this metric to reflect the complexity caused by Method Chaining. The following are the weights for each level of technique. To indicate the complexity created by Method Chaining, the number of various methods executed by one object at a time (degree of methods) is regarded as the most important attribute in this measure. The weights for each level of technique are listed below.

Degree Of Methods	**Weight**
Non – method statements	0
One Method	0
Two Methods	1 (n – 1) $\rightarrow$ (2 – 1 = 1)
N number of Methods	n - 1

$$W_{mc} = \sum_{i=1}^{n} d_i$$

In the above equation, Wmc represents the weight due to method chaining.
n represents the number of method invocation statements.
d represents the assigned weight for the relevant degree of methods.

Example:

```
public class MethodChainingClass{
        public static void main(String a[]){
            ObjectProvider obj = new ObjectProvider ();
            obj.methodA().methodD().
  methodE().methodB();
    obj.methodC().methodA();
    obj.methodA();
    obj.methodA().methodD().methodE();
        }
    }
```

n = number of method invocation statements = 4

$$W_{mc} = \sum_{i=1}^{4} d_i$$

$$W_{mc} = (3 + 1 + 0 + 2)$$
$$W_{mc} = 6//$$

III. CONCLUSION

In this paper, we find that the proposed complexity metrics and its approaches does not only fulfill the task of simplifying the system, but also uses it as a means to calculate the complexity in the most simplest manner, while bringing out good object-oriented metrics. It also helps us to understand the importance of using object-oriented metrics to calculate the complexity metrics under any circumstances, and therefore aid developers and programmers to define the complexity of software systems.

REFERENCES

[1] E. J. Weyuker, "Evaluating software complexity measures," in IEEE Transactions on Software Engineering, vol. 14, no. 9, pp. 1357-1365, Sept. 1988, doi: 10.1109/32.6178.
[2]
[3] T. Honglei, S. Wei and Z. Yanan, "The Research on Software Metrics and Software Complexity Metrics," 2009 International Forum on Computer Science-Technology and Applications, 2009, pp. 131-136, doi: 10.1109/IFCSTA.2009.39.
[4]
[5] Thamburaj, T.F. and Aloysius, A. (2015). Cognitive Weighted Polymorphism Factor: A Comprehension Augmented Complexity Metric. International Journal of Computer and Information Engineering, [online] 9(11), pp.2335–2340. Available at: https://publications.waset.org/10002880/cognitive-weighted-polymorphism-factor-a-comprehension-augmented-complexity-metric [Accessed 11 Apr. 2022].

[6] 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.

[7] Yadav, A. and Khan, R.A. (2012). Development of Encapsulated Class Complexity Metric. Procedia Technology, [online] 4, pp.754–760. Available at: https://www.sciencedirect.com/science/article/pii/S2212017312004021 [Accessed 11 Apr. 2022].

[8] T.J.McCabe, "A Complexity Measure ", IEEE Transaction on Software Engineering, Vol. 2, 1976, pp. 308-320.

[9] M.A. Soltani. Method chaining in Java. on linkedin.com https://www.linkedin.com/pulse/methodchaining-java-mehdi-ali-soltani [Accessed: 24 April 2020]

[10] S. Misra, "An Object Oriented Complexity Metric Based on Cognitive Weights," 6th IEEE International Conference on Cognitive Informatics, 2007, pp. 134-139, doi: 10.1109/COGINF.2007.4341883.

t