Object-Oriented Inheritance Metrics in the Context of Cognitive Complexity

Deepti Mishra
Department of Computer Engineering
Atilim University, Ankara, Turkey
deepti@atilim.edu.tr

Alok Mishra
Department of Software Engineering
Atilim University, Ankara, Turkey
alok@atilim.edu.tr

Abstract. It is important to identify modules that are fault prone or exhibit evidence of high cognitive complexity as these modules require corrective actions such as increased source code inspection, refactoring or performing more exhaustive testing. This can lead to a better quality software system. It has been found that inheritance has an impact on the cognitive complexity of a software system. In this paper, two inheritance metrics based on cognitive complexity, one at class level CCI (Class Complexity due to Inheritance) and another at program level ACI (Average Complexity of a program due to Inheritance), have been proposed for object-oriented software systems. Additionally, one more metric MC (Method Complexity) has been proposed to calculate the complexity of a method. These proposed metrics are compared with some well known object-oriented inheritance metrics by calculating their values for three random C++ programs. It has been observed that CCI and ACI are better to represent cognitive complexity due to inheritance than other well known class level and program level inheritance metrics.

Keywords: cognitive complexity, software metrics, object-oriented systems
1. Introduction

The complexities of software systems are more prominent now than ever and therefore challenges to a software engineer are to somehow understand and manage such large systems and deal with software cognitive complexities [10]. Cognitive informatics is a trans-disciplinary enquiry of cognitive and information sciences that investigates the internal information processing mechanisms and processes of the brain and natural intelligence, and their engineering applications via an interdisciplinary approach [34]. Software complexity measures serve both as an analyzer and a predictor in quantitative software engineering [32].

Thorough testing of every module is not feasible because of limited resources and time. Therefore identification of complex modules is very important as they require more efforts during inspection or review and rigorous testing which are fundamental components of software quality process [29] so as to develop a better quality software system [28]. Software defects are often the result of the incomplete or incorrect comprehension of a program segment [21]. Further, the identification of a complex module may help during maintenance. Software complexity metrics can be used to locate such modules.

Object-oriented technologies have been increasingly used in organizations these days. It is theorized that structural properties such as coupling, cohesion, functional complexity and inheritance have an impact on the cognitive complexity of the system [12]. That is, it places a mental burden on developers, inspectors, testers and maintainers to understand the system, both at the component and system level [33].

The main motivation of this paper is to propose new metrics that represent cognitive complexity of a class and program due to inheritance. These metrics will help identifying classes and programs having high complexity. This identification can lead to either channeling more testing efforts to improve quality or redesigning programs to reduce the complexity. This paper is organized as follows: In the next section, related work is described. In section 3, two inheritance complexity metrics, one at class level CCI (Class Complexity due to Inheritance) and another at program level ACI (Average Complexity of a program due to Inheritance), have been proposed for object-oriented software systems. These metrics are based on both design and cognitive aspects. Section 4 presents the practical implications of the proposed metrics. These metrics are also validated with two popular and extensively used frameworks: a theoretical and a practical framework. In section 5, some well known inheritance metrics for object-oriented systems are described and compared with proposed metrics. Finally, the paper concludes in section 6.

2. Literature Survey

Software metrics were introduced to support the most critical issues in software development and provide support for planning, predicting, monitoring, controlling and evaluating the quality of both software products and processes [6, 13, 14]. The quality of the product of a software development project is related to the efficacy of software comprehension experienced by the developers [22]. Therefore, the location of a program segment that presents a comprehension challenge to the software developer can be the basis for isolating code at a greater risk of defects [31]. Quality attributes such as maintainability are related with the ease of understanding which can be improved by good programming style [27]. Therefore, comprehension of source code plays a prominent role in ensuring quality during software maintenance and evolution [31]. The efforts required to understand a given code segment can vary with its size and complexity [22]. Complexity is determined in the context of human cognitive processes [22].
The application of cognitive complexity measures include estimating the time required to understand a program, estimating defect density, estimating debugging time, using the measure as a criterion for program slicing [31], as a style assessment metric and as a domain complexity assessment metric [20]. Ways of assessing the comprehensibility of the code can aid in improving products quality by providing targets for improvement efforts [15, 31].

Some traditional complexity metrics can be supported by the fact that they are clearly related to cognitive limitations [21]. These include LOC, fan-out or external coupling [15], and decision points such as McCabe’s Cyclomatic Complexity [26]. A well known effort to define metrics that corresponds to cognitive effort and complexity is the work of Halstead [16].

The object-oriented paradigm includes new concepts such as classes, methods, messages, inheritance, polymorphism, encapsulation etc. Inheritance is one of the key features of the object-oriented paradigm as it promotes reuse. Many studies [2, 8, 9, 24] have claimed that use of inheritance reduces the amount of software maintenance necessary and ease the burden of testing. The reuse of software through inheritance is claimed to produce more maintainable, understandable and reliable software [3, 4, 5]. Basili et al. [4] used Chidamber and Kemerer [9] metrics as a predictor of fault prone classes in their empirical study and found that a class located deep in the inheritance hierarchy is more fault-prone than a class higher up in the hierarchy. Daly et al. [11] also provided evidence which suggests that inheritance depth and conceptual entropy of class hierarchies can cause programmers difficulty when trying to maintain object-oriented software. This is further supported by Harrison, Counsell and Nithi [17] through their experimental assessment that systems without inheritance are easier to understand and modify than corresponding systems containing three to five levels of inheritance. Therefore, it was concluded that deriving new classes from existing library classes makes the system harder to integrate correctly. The obvious reason is that to inherit a new class, the parent’s implementation must be, at least partially, understood as well as any of the parent’s ancestors. Although inheritance within an object-oriented system is a great way to enhance the readability and internal organization of a large program, inheriting from classes designed and written by other programmers (library classes) can prove too costly in terms of time and effort required to understand the implementation of the library classes.

### 3. New Proposed Inheritance Metrics

We propose two metrics for inheritance, Class Complexity due to Inheritance (CCI) and Average Complexity of a program due to Inheritance (ACI). Also, one more metric is proposed to calculate the complexity of a method (MC) which is based on McCabe’s cyclomatic complexity [26] but it also takes into account depth of control structures. McCabe’s cyclomatic complexity of two programs, one having two sequential loops and the other having the same loops nested, is the same. This is not an ideal situation because the complexity of a program increases with nesting. This is also supported by Piwowarski [30] that the depth of nesting of loops plays a critical role in the complexity of software.

**Method Complexity (MC)** can be calculated as

\[
MC_j = P_j + D_j + 1
\]  

where: \(MC_j\) is the complexity of \(j^{th}\) method,  
\(P_j\) is the number of predicates \(j^{th}\) method has,
\( D_j \) is the maximum depth of control structures in \( j^{th} \) method; if there is no nested control structures then \( D_j = 0 \); if there is one inside another then \( D_j = 1 \) and so on ...

**Class Complexity due to Inheritance (CCI)** can be calculated as

\[
CCI_i = \sum_{\text{inherit}=1}^{k} CCI_{\text{ inherit}} + \sum_{j=1}^{l} MC_j
\]

where: \( CCI_i \) is the complexity of an \( i^{th} \) class due to inheritance, 
\( k \) is the number of classes, \( i^{th} \) class is inheriting directly, 
\( CCI_{\text{ inherit}} \) is the complexity of an inherited class, 
\( l \) is the number of methods excluding constructors, destructors, pure virtual function, \( i^{th} \) class has, 
\( MC_j \) is the complexity of \( j^{th} \) method in \( i^{th} \) class and can be calculated using a new proposed method complexity metric (MC).

**Average Complexity of a program due to Inheritance (ACI)** can be calculated as

\[
ACI = \frac{\sum_{i=1}^{n} CCI_i}{n}
\]

where: \( CCI_i \) is the complexity of an \( i^{th} \) class due to inheritance, 
\( n \) is the total number of classes in the program.

Consider program 1 and program 2 given in the appendix. Consider the base class Person and one derived class Employee in program 1. Similarly, consider the base class Shape and one derived class Triangle in program 2. Now, both these programs are similar as both have one base class and one derived class as shown in figure 1. Complexity due to inheritance is as follows:

**Program A:**

Complexity of base class Person can be calculated as

\[
CCI_i = \sum_{\text{inherit}=1}^{k} CCI_{\text{ inherit}} + \sum_{j=1}^{l} MC_j = 0 + (MC_{p1} + MC_{p2}) = 2
\]

(\( \sum_{\text{inherit}=1}^{k} CCI_{\text{ inherit}} = 0 \) as class Person is not inheriting any class)

Complexity of derived class Employee can be calculated as

\[
CCI_i = \sum_{\text{inherit}=1}^{k} CCI_{\text{ inherit}} + \sum_{j=1}^{l} MC_j = 2 + (MC_{e1} + MC_{e2}) = 4
\]
Figure 1. Two programs having one base and one derived class.

\[
\left( \sum_{\text{inherit}=1}^{k} CCI_{\text{inherit}} = 2 \text{ as class Employee is only inheriting class Person with } CCI = 2 \right)
\]

Average Complexity of full program due to inheritance (ACI)

\[
\frac{\sum_{i=1}^{n} CCI_i}{n} = \frac{(2 + 4)}{2} = \frac{6}{2} = 3
\]

**Program B:**

Complexity of base class Shape can be calculated as

\[
CCI_i = \sum_{\text{inherit}=1}^{k} CCI_{\text{inherit}} + \sum_{j=1}^{l} MC_j = 0 + (MC_{s1} + MC_{s2} + MC_{s3} + MC_{s4} + MC_{s5} + MC_{s6} + MC_{s7}) = 7
\]

\[
\left( \sum_{\text{inherit}=1}^{k} CCI_{\text{inherit}} = 0 \text{ as class Shape is not inheriting any class} \right)
\]

Complexity of derived class Triangle can be calculated as

\[
CCI_i = \sum_{\text{inherit}=1}^{k} CCI_{\text{inherit}} + \sum_{j=1}^{l} MC_j = 7 + (MC_{t1} + MC_{t2}) = 9
\]

\[
\left( \sum_{\text{inherit}=1}^{k} CCI_{\text{inherit}} = 7 \text{ as class Triangle is only inheriting class Shape with } CCI = 7 \right)
\]
Average Complexity of full program due to inheritance (ACI) =  
\[ \sum_{i=1}^{n} CCI_i \]
\[ = \frac{(7 + 9)}{2} = 16/2 = 8 \]

Complexity (due to inheritance) metrics should not only consider the number of classes a particular class is inheriting but also the complexities of the inherited classes. Also, it should consider not only the number of methods inherited but also the complexity of the methods. This is also supported by Abreu and Carapuca [1] that the greater the inheritance relation is, the greater the number of methods a class is likely to inherit, making it more complex and therefore requiring more testing. A method with a complex decision structure will be harder to test and maintain and is more error-prone [1]. They suggested that complexity metrics based on the above criteria allow pin-pointing potentially troublesome classes or methods, thus helping in the planning of review and test efforts [1]. Constructors and destructors are not considered in these calculations as they are not inherited. If inheritance metric only considers number of classes inherited then it gives value 1 for both examples. But if the number of classes inherited, complexities of inherited classes as well as complexity of the derived class are all taken into consideration then the value of inheritance metric for program A and B as shown in figure 1 are 3 and 8 respectively. These values are more reasonable as program A is simpler than program B in terms of inheritance. The derived class only inherits 2 methods from the base class in program A whereas the derived class is inheriting 7 methods in program B.

4. Evaluation of the Metrics

4.1. Evaluation through a Practical Framework

Proposed complexity metrics are calculated from source code. The complexity of the code can be a good predictor of understandability of the code, reusability, testing efforts and future maintenance efforts [31]. Understandability, reusability, testability and maintainability are different quality attributes of the program code so proposed metrics predict the quality of a program.

- Class Complexity due to Inheritance (CCI) metric is directly linked to a programmer’s comprehension effort when a class is reused in OO design. Class Complexity due to Inheritance (CCI) metric takes into account the complexity of methods within the class and the complexity of inherited classes. Therefore, classes with higher CCI values either have complex methods of their own or inheriting complex classes or both. In both cases, classes with higher CCI values will take more effort in understanding. Further, it may require understanding their parents as well. Therefore, these classes are more difficult to reuse than classes with lower CCI values.

- Class Complexity due to Inheritance (CCI) metric can be used to predict the testing effort needed to test a class thoroughly. Classes with higher CCI values are complex and therefore can be considered more error prone. So, testing efforts should be directed more towards these classes than classes with lower CCI values.

- Class Complexity due to Inheritance (CCI) metric is directly linked to maintenance effort. Classes with higher CCI values are complex so there is more probability of errors. To correct these errors, these classes need to be first understood by the developers performing maintenance and then
changes can be made. If these classes inherit some other classes then their parent classes also need to be understood. Classes with higher CCI values will need more effort to comprehend and modify.

- Average Complexity of a program due to Inheritance (ACI) is directly linked to design complexity. Programs with higher ACI values have a more complex design than programs with lower ACI values.

- Average Complexity of a program due to Inheritance (ACI) can be used to predict testing efforts. As programs with higher ACI values are complex in design so these programs require more efforts during testing; specifically integration testing.

- Average Complexity of a program due to Inheritance (ACI) can also be used to predict maintenance efforts. Maintenance efforts will be more because of the difficulty in comprehension and modification due to complex design.

Kaner and Bond [19] proposed a framework for the evaluation of a metric as follows:

- **The purpose of the metric**: The purpose of Class Complexity due to Inheritance (CCI) metric is to predict the quality of a class in terms of understandability, reusability, testability and maintainability. The purpose of Average Complexity of a program due to Inheritance (ACI) metric is to predict the quality of design of a program comprising of multiple classes.

- **Scope of the metric**: The scope of Class Complexity due to Inheritance (CCI) metric is a single class in an object oriented program/package. The scope of Average Complexity of a program due to Inheritance (ACI) metric is an object-oriented program or full package consisting of many classes.

- **Attributes measured by the metric**: Class Complexity due to Inheritance (CCI) metric measures the quality of a class in terms of understandability, reusability, testability and maintainability whereas Average Complexity of a program due to Inheritance (ACI) metric is an object-oriented design complexity metric that measures the quality of design of a program comprising of multiple classes.

- **Natural scale of the attribute**: Quality attributes like understandability, reusability, testability, maintainability can be measured on an ordinal scale.

- **Natural variability of the attribute**: Quality attributes are subjective in nature. Understandability, reusability, testability and maintainability of the same class/program may change according to the perspective of different developers.

- **Metrics definition**: The definition of both CCI metric and ACI metric are given in section 3.

- **Measuring instrument to perform the measurement**: The measuring instrument for both CCI and ACI is counting by a human or a machine. Either developers or quality team has to measure the metrics value manually or a tool can be used to calculate CCI and ACI metrics values.
• **Natural scale of the metric:** The natural scale of Class Complexity due to Inheritance (CCI) metric is ratio scale. Similarly, Average Complexity of a program due to Inheritance (ACI) metric’s natural scale is also ratio scale.

• **The natural variability of the readings from the measuring instrument:** Since the reading from the measuring instrument (manually or using a tool) is based on a mathematical formula and not subjective in nature so no variability of readings (measurement error) from the instrument is expected.

• **Relationship of the attribute to the metric value:** Both CCI and ACI metrics are related to the quality of class/program. If the value of CCI metric for a class increases, the quality (understandability, reusability, testability and maintainability) of that class will decrease. Similarly an increase in ACI value of an object-oriented program decreases the quality of that program in terms of understandability, testability and maintainability.

• **The natural and foreseeable side effects of using the measuring instrument:** Manual calculation of ACI and CCI metric for large program codes may provide an incorrect result and therefore using a tool to calculate these metrics values will be a more reliable option. It will also result in less efforts in the measurement.

4.2. Theoretical Evaluation

Briand, Morasca and Basili [7] proposed a framework which is generic and based on precise mathematical concepts. This framework provides a set of properties that characterize and formalize intuitively relevant measurement concepts: complexity, size, length, cohesion and coupling. This framework is extensively used by researchers for the theoretical validation of metrics and basic definitions of this framework are as follows [7]:

**Definition Representation of Systems and Modules:** A system S will be represented as a pair of \(< E, R >\), where E represents the set of elements of S, and R is a binary relationship on E (\(R \subseteq E \times E\)) representing the relationships between S’s elements. Given a system \(S = < E, R >\), a system \(m = < E_m, R_m >\) is a module of S if and only if \(E_m \subseteq E, R_m \subseteq E_m \times E_m\) and \(R_m \subseteq R\). As an example, E can be defined as the set of code statements and R as the set of control flows from one statement to another. A module m may be a code segment or a subprogram. For the proposed metrics, E represents the set of classes (elements) in a program/package S (system) and R is the relationship on E between classes and between subelements within a class. The subelements of a class are its data and methods and the subelements of a method are the statements within a method.

**Definition Complexity:** The complexity of a system S is a function Complexity(S) that is characterized by the following properties:

- Property **COMPLEXITY.1: Nonnegativity.** The complexity of a system \(S = < E, R >\) is non-negative i.e. \(\text{Complexity}(S) \geq 0\).

  **Proof:** Both ACI and CCI metrics values are calculated using non-negative numbers and therefore metrics values will always be greater than or equal to zero.
• Property COMPLEXITY.2: Null Value. The complexity of a system $S = <E, R>$ is null if $R$ is empty i.e. $R = \emptyset \Rightarrow \text{Complexity}(S) = 0$.

Proof: If there is no relationship between classes and between subelements within classes or methods, then both ACI and CCI metrics will be zero.

• Property COMPLEXITY.3: Symmetry. The complexity of a system $S = <E, R>$ does not depend on the convention chosen to represent the relationships between its elements.

Proof: ACI and CCI metrics are not dependent on the convention. The same system represented using different conventions will give the same ACI and CCI value.

• Property COMPLEXITY.4: Module Monotonicity. The complexity of a system $S = <E, R>$ is no less than the sum of the complexities of any two of its modules with no relationships in common i.e. $(S = <E, R>$ and $m_1 = <E_{m1}, R_{m1} >$ and $m_2 = <E_{m2}, R_{m2} >$ and $m_1 \cup m_2 \subseteq S$ and $R_{m1} \cap R_{m2} = \emptyset$)

$\Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2)$

Proof: Consider a system containing two modules with no relationship in common (see figure 5). Assume the system $S$ contains two modules $m_1$ (class MovingVeh) and $m_2$ (class Driver).

$\text{CCI}(S) = \text{CCI(MovingVeh + Driver)} = 6$

$\text{CCI(MovingVeh)} + \text{CCI(Driver)} = 3 + 3 = 6$

Therefore, $\text{CCI} (S) \geq \text{CCI} (m_1) + \text{CCI} (m_2)$.

Since this property is applied at module level (class level) and ACI is a system level metric therefore this property is not applicable to ACI metric.

• Property COMPLEXITY.5: Disjoint Module Additivity. The complexity of a system $S = <E, R>$ composed of two disjoint modules $m_1, m_2$ is equal to the sum of the complexities of the two modules. i.e. $(S = <E, R>$ and $S = m_1 \cup m_2$ and $m_1 \cap m_2 = \emptyset$)

$\Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2)$

Proof: Consider a system consisting of two independent modules 1 and 2 as shown in figure 2.

$\text{CCI}(S) = \text{CCI(module 1 + module 2)} = 5+7+3 = 15$

$\text{CCI(module 1) + CCI(module 2)} = (5+7)+3 = 15$

Therefore, $\text{CCI} (S) = \text{CCI} (m_1) + \text{CCI} (m_2)$

Since this property is applied at module level (class level) and ACI is a system level metric therefore this property is not applicable to ACI metric.

5. Comparison With Other Inheritance Metrics

Some well known inheritance metrics are summarized in table 1. These metrics values are calculated for three programs written in C++ which are given in appendix. The inheritance structure of these three programs are shown in figure 3, 4 and 5.

Metrics TLI, MIF, AID and our proposed metric ACI are program level metrics whereas other metrics are class level metrics. Class level metrics are used to determine the complexity of a class whereas program level metrics can be used to determine the complexity of a program or module (consisting of many classes) as a whole. Values of program level metrics (including proposed metric ACI) and class
Table 1. Inheritance metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inheritance Metrics by Chidamber and Kemerer [9]</strong></td>
<td></td>
</tr>
<tr>
<td>Depth of Inheritance Tree (DIT)</td>
<td>Depth of inheritance of the class is the DIT metric for the class.</td>
</tr>
<tr>
<td>Number of Children (NOC)</td>
<td>Number of immediate subclasses subordinated to a class is the NOC for that class.</td>
</tr>
<tr>
<td>Number of Methods Extended (NME)</td>
<td>Number of methods extended i.e. redefined in subclass by invoking a method with the same name on a superclass.</td>
</tr>
<tr>
<td><strong>Inheritance Metrics by Abreu and Carapuca [1]</strong></td>
<td></td>
</tr>
<tr>
<td>Total Children Count (TCC)</td>
<td>Number of classes that inherit directly from a class is the TCC of that class.</td>
</tr>
<tr>
<td>Total Progeny Count (TPC)</td>
<td>Number of classes that inherit directly or indirectly from a class is the TPC of that class.</td>
</tr>
<tr>
<td>Total Parent Count (TPAC)</td>
<td>The number of super classes from which a class inherits directly is the TPAC of that class.</td>
</tr>
<tr>
<td>Total Ascendancy Count (TAC)</td>
<td>The number of super classes from which a class inherits directly or indirectly is the TAC of that class.</td>
</tr>
<tr>
<td>Total length of inheritance chain (TLI)</td>
<td>Total number of edges in the inheritance hierarchy graph.</td>
</tr>
<tr>
<td>Method Inheritance Factor (MIF)</td>
<td>Ratio of the sum of the inherited methods (not overridden) in all classes of the system under consideration to the total number of available methods (defined plus inherited) for all classes. Methods defined in a class can be new or overridden.</td>
</tr>
<tr>
<td><strong>Inheritance Metrics by Henderson-Sellers [18]</strong></td>
<td></td>
</tr>
<tr>
<td>Average Inheritance Depth (AID)</td>
<td>$AID = \frac{\text{Sum of depth of each class}}{\text{Number of classes}}$.</td>
</tr>
<tr>
<td><strong>Inheritance Metrics by Li [23]</strong></td>
<td></td>
</tr>
<tr>
<td>Number of Ancestor Classes (NAC)</td>
<td>Total number of ancestor classes from which a class inherits is the NAC of that class.</td>
</tr>
<tr>
<td>Number of Descendent Classes (NDC)</td>
<td>Total number of descendent classes (subclasses) of a class is the NDC of that class.</td>
</tr>
<tr>
<td><strong>Inheritance Metrics by Lorenz and Kid [25]</strong></td>
<td></td>
</tr>
<tr>
<td>Number of Methods Inherited (NMI)</td>
<td>Number of methods inherited i.e. defined in superclass and inherited unmodified.</td>
</tr>
<tr>
<td>Number of Methods Overridden (NMO)</td>
<td>Number of methods overridden i.e. redefined compared to superclass.</td>
</tr>
<tr>
<td>Number of New Methods (NNA)</td>
<td>Number of added methods in a subclass i.e. defined in subclass and not in superclass.</td>
</tr>
</tbody>
</table>
level metrics (including proposed metric CCI) for program 1, 2 and 3 are presented in tables 2 and 3 respectively.

According to table 2, TLI, AID and our proposed metric ACI values are lowest for program 3 but TLI and AID values for program 1 and 2 are contradictory to our proposed metric ACI. TLI only considers the number of classes inherited whereas AID just considers the average number of classes inherited in an inheritance tree without considering how complex the inherited classes are or how many methods are inherited or how complex the inherited methods are. As the number of classes inherited are more in program 1 than in program 2 so TLI and AID values for program 1 are higher. Although the number of classes inherited are less in program 2, the complexity of the inherited class is more in program 2. The
inherited class in program 2 has more methods than the inherited class in program 1. So, the cognitive complexity (due to inheritance) of program 2 is higher than program 1.

MIF and ACI values agree on the most complex program. Program 2 is the most complex among all three programs according to MIF and ACI values. According to MIF metric, program 1 (value 0) is less complex than program 3 (value 0.4). These values are contradictory to our proposed metric ACI.
Table 2. Inheritance metrics (program level) values for program 1, 2 and 3

<table>
<thead>
<tr>
<th>Program</th>
<th>TLI</th>
<th>MIF</th>
<th>AID</th>
<th>ACI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1.33</td>
<td>6.33</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.625</td>
<td>1.00</td>
<td>8.75</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.4</td>
<td>0.33</td>
<td>5.00</td>
</tr>
</tbody>
</table>

Table 3. Inheritance metrics (class level) values for program 1, 2 and 3

<table>
<thead>
<tr>
<th>Class</th>
<th>DIT</th>
<th>NME</th>
<th>TMC</th>
<th>TPM</th>
<th>TPAC</th>
<th>NAC</th>
<th>NDC</th>
<th>NMI</th>
<th>NMO</th>
<th>NNA</th>
<th>NCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Employee</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Student</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>SalaryEmp</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>CommissionEmp</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>HourlyEmp</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Shape</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Triangle</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Rectangle</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NameTriangle</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>MovingVeh</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Driver</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>DrivenVehicle</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

MIF metric considers the ratio of the number of methods inherited by a class (without redefinition) by total number of available methods. Again it doesn’t consider how complex inherited methods are. Also, it seems in MIF metric calculation that overridden methods decrease inheritance complexity as they are used in the denominator. Inheritance complexity should increase if the number and complexity of the inherited methods increase irrespective of whether the inherited methods are inherited as they are or redefined in the subclass. Therefore, ACI better represents cognitive complexity (due to inheritance) of a program.

It is obvious that classes at a lower level in the hierarchy in an inheritance tree are more complex because understanding these classes requires understanding, at least partially, parent’s implementation as well as any of the parent’s ancestors. In this case, DIT, NME, TPAC, TAC, NAC, NMO and our proposed class level metric CCI are more suitable to determine the complexity of a class since these metrics values are higher for lower level classes as seen in Table 3. NME, TPAC and NMO values are the same for classes at different levels of the inheritance tree; for example, their values are the same.
for class Employee, SalaryEmp, CommissionEmp and HourlyEmp. This is not an ideal situation as SalaryEmp, CommissionEmp and HourlyEmp are subclasses of Employee so their complexity due to inheritance should be more. What makes our proposed metric different from DIT, TAC and NAC is that values of DIT, TAC and NAC for class Employee (program 1) and class Triangle (program 2) are the same as they are both inheriting one class. But their parent classes are different in complexity and they inherit different methods in terms of number and complexity so their values should be different. CCI values are different (Employee = 4 and Triangle = 9) therefore CCI better represents cognitive complexity (due to inheritance) of a class.

6. Conclusion

Cognitive complexity measures can be used to predict the quality of a software system. It has been found that identifying complex modules can help during software testing and maintenance phases. Two object-oriented inheritance cognitive complexity metrics are proposed and compared with other well known object-oriented inheritance metrics by using three random C++ programs. Reasons are given to prove that CCI (class level inheritance complexity metric) and ACI (program level inheritance complexity metric) are better to represent cognitive complexity due to inheritance than other well known class level and program level inheritance metrics.

Proposed metrics are calculated from source code so these metrics can be a good predictor of understandability, reusability, testing efforts and future maintenance efforts. The programmer’s comprehension effort will be more for classes having high CCI values. Therefore, these classes will be difficult to reuse and require more testing as well as maintenance efforts. Average Complexity of a program due to Inheritance (ACI) can be linked to design complexity so programs having high ACI values require more efforts during testing; specifically integration testing. Maintenance efforts will also be more because of the difficulty in comprehension and modification due to complex design. As a future work, these metrics can be used for fault prediction quantitatively. We are working to develop a tool to calculate the proposed metrics.

References


APPENDIX

Program 1:

```cpp
#include<conio.h>
#include<stdio.h>
#include<iostream.h>

//base class Person

class Person
{
    char name[50],gen[6];
    int age;
    public:
        Person()
        void get_info();
        void put_info();
    }
    void Person::get_info()
    {
        cout<<"Enter the name of person?";
        gets(name);
        cout<<"Enter the gender of person?";  //MCp1 = 1
        cin>>gen;
        cout<<"Enter the age of person?";
        cin>>age;
    }
    void Person::put_info()
    {
        cout<<name<<"\t";
        cout<<gen<<"\t";  //MCp2 = 1
        cout<<age<<"\t";
    }

    //*****************************************************************************
    // employee class derived from person

    class Employee:public Person
    {
        int empid;
        char des[20];
        public:
            Employee()
    }
```
void get_info();
void put_info();

void Employee::get_info()
{
    Person::get_info();
    cout<<"Enter emp id?";   //MCe1 = 1
    cin>>empid;
    cout<<"Enter designation?";
    gets(des);
}

void Employee::put_info()
{
    Person::put_info();
    cout<<empid<<"\t";      //MCe2 = 1
    cout<<des<<"\n";
}

//****************************
//SalaryEmp class derived from Employee

class SalaryEmp: public Employee
{
    double salary;
    public:
        SalaryEmp(){}  
        void get_info();
        void put_info();
};

void SalaryEmp::get_info()
{
    Employee::get_info();
    cout<<"Enter salary";  //MCs1 = 2
    cin>>salary;
    if (salary < 0.0)
        salary = 0.0;
}

void SalaryEmp::put_info()
{
    Employee::put_info();   //MCs2 = 1
    cout<<salary<<"\n";
}
class HourlyEmp: public Employee
{
  int hoursWorked;
  double wage;
  double earning;
public:
  HourlyEmp(){
  }
  void get_info();
  void put_info();
};

void HourlyEmp::get_info()
{
  Employee::get_info();
  cout<<"Enter number of hours worked";
  cin>>hoursWorked;
  cout<<"Enter hourly wage";
  cin>>wage;
  if (hoursWorked >= 0.0)
  {
    if (hoursWorked <= 168.0)
      hoursWorked = hoursWorked;
    else
      hoursWorked = 0.0;
  }
  else
    hoursWorked = 0.0;
  if (wage < 0.0)  //MCh1 = P+D+1
    wage = 0.0;  // = 4+1+1= 6
  if (hoursWorked <= 40.0)
    earning = hoursWorked*wage;
  else
    earning = (40*wage)+((hoursWorked-40)*wage*1.5);
}

void HourlyEmp::put_info()
{
  Employee::put_info();
  cout<<hoursWorked<<"\t";  //MCh2 = 1
cout<<wage<<"\t";
cout<<earning<<"\n";
}

//*******************************************************
//CommissionEmp class derived from Employee

class CommissionEmp: public Employee
{
int sale;
double commRate;
double earning;
public:
    CommissionEmp(){
void get_info();
void put_info();
};

void CommissionEmp::get_info()
{
    Employee::get_info();
cout<<"Enter gross sale amount";
cin>>sale;
cout<<"Enter commission rate";
cin>>commRate;

    if (sale < 0.0)
sale = 0.0;

    if (commRate > 0.0) //MCc1 = P+D+1
{
        if (commRate < 1.0)
            commRate = commRate;
        else
            commRate = 0.0;
    }
    else
        commRate = 0.0;

    earning = sale*commRate;
}
void CommissionEmp::put_info()
{
    Employee::put_info(); //MCc2 = 1
    cout<<sale<<"\t";
    cout<<CommRate<<"\t";
    cout<<earning<<"\n";
}

//******************************************
//Student class derived from Person

class Student:public Person
{
    int studid;
    char class_name[10];
    public:
    Student() {}
    void get_info();
    void put_info();
};
void Student::get_info()
{
    Person::get_info();
    cout<<"Enter stud id?"; cin>>studid;
    cout<<"enter student class name?"; //MCs1 = 1
    gets(class_name);
}
void Student::put_info()
{
    Person::put_info();
    cout<<studid<<"\t"; //MCs2 = 1
    cout<<class_name<<"\n";
}

void main()
{
    clrscr();
    Employee e;
    cout<<"\n\nENTER EMPLOYEE INFORMATION\n\n";
    e.get_info();
    Student p;
    cout<<"\n\nENTER STUDENT INFORMATION \n\n";
    p.get_info();
cout<<"\n NAME\tGENDER\tAGE\tEMPID\tSALARY\tDESIGNATION\n";
e.put_info();
cout<<"\n NAME\tGENDER\tAGE\tSTUDID\tCLASS NAME\n";
p.put_info();
}

Program 2:
#include <iostream>
#include <cstring>
using namespace std;

// base class Shape

class Shape {
    double width;
    double height;
    char name[20];
public:
    Shape() {
        width = height = 0.0;
        strcpy(name, "unknown");
    }
    Shape(double w, double h, char *n) {
        width = w;
        height = h;
        strcpy(name, n);
    }
    Shape(double x, char *n) {
        width = height = x;
        strcpy(name, n);
    }
    void display() { //MCs1 = 1
        cout << "Width and height are " << width << " and " << height << "\n";
    }
    double getWidth() { return width; }  //MCs2 = 1
    double getHeight() { return height; }  //MCs3 = 1
    void setWidth(double w) { width = w; }  //MCs4 = 1
    void setHeight(double h) { height = h; }  //MCs5 = 1
    char *getName() { return name; }  //MCs6 = 1
    virtual double area() { //MCs7 = 1

cout << "Error: area() must be overridden.\n";
    return 0.0;
};

//******************************************
//class Triangle derived from Shape

class Triangle : public Shape {
    char style[20];
public:
    Triangle() {
        strcpy(style, "unknown");
    }
    Triangle(char *str, double w, double h) : Shape(w, h, "triangle") {
        strcpy(style, str);
    }
    Triangle(double x) : Shape(x, "triangle") {
        strcpy(style, "isosceles");
    }
    double area() {
        return getWidth() * getHeight() / 2;  //MCt1 = 1
    }
    void showStyle() {
        cout << "Triangle is " << style << "\n";  //MCt2 = 1
    }
};

//******************************************
// class NameTriangle derived from Triangle

class NameTriangle : public Triangle {
    char name[20];
public:
    NameTriangle(char *clr, char *style, double w, double h) :
        Triangle(style, w, h) {
        strcpy(name, clr);
    }
void displayName() {
    cout << "Name is " << name << "\n"; //MCn1 = 1
};
//******************************************
//class Rectangle derived from Shape

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : Shape(w, h, "rectangle") { }
    Rectangle(double x) : Shape(x, "rectangle") { }
    void displayName() { //MCr1 = 1
        cout << "Name is " << getName() << "\n";
    }

double area() {
    return getWidth() * getHeight(); //MCr2 = 1
}
};

int main() {
    Shape *shapes[5];

    shapes[0] = &Triangle("right", 8.0, 12.0);
    shapes[1] = &Rectangle(10);
    shapes[2] = &Rectangle(10, 4);
    shapes[3] = &Triangle(7.0);
    shapes[4] = &Shape(10, 20, "generic");

    for(int i=0; i < 5; i++) {
        cout << "object is " << shapes[i]->getName() << "\n";
        cout << "Area is " << shapes[i]->area() << "\n\n";
    }

    NameTriangle t1("A", "right", 8.0, 12.0);
    NameTriangle t2("B", "isosceles", 2.0, 2.0);

    t1.showStyle();
    t1.display();
    t1.displayName();
    cout << "Area is " << t1.area() << "\n";
Program 3:

```cpp
#include <iostream>
using namespace std;

// class declaration and implementation part
// base class #1
class movingVeh
{
   protected:
      double payload;
      double weight; //note this variable
      double mpg;
   public:
      void initialize(double pl, double gw, double input_mpg)
      {
         payload = pl;
         weight = gw; //MCm1 = 1
         mpg = input_mpg;
      }
      double efficiency(void) //MCm2 = 1
      {
         return(payload / (payload + weight));
      }
      double cost_per_ton(double fuel_cost) //MCm3 = 1
      {
         return (fuel_cost / (payload / 2000.0));
      }
};

// base class #2
class driver
{
   protected:
      double hourly_pay;
      double weight; // another weight variable
      // variable with same name as in class number one
   public:
      void initialize(double pay, double input_weight)
      // same method name but different number of parameter
      {
```
hourly_pay = pay; //MCd1 = 1
weight = input_weight;
};
double cost_per_mile(void) {return (hourly_pay / 55.0); }; //MCd2 = 1
double drivers_weight(void) {return (weight); }; //MCd3 = 1
};

// derived class with multi inheritance
// declaration and implementation
class drivenVehicle : public movingVeh, public driver
{
public:
    void initialize(double pl, double gw, double input_mpg, double pay)
    // another same method name but different number of parameter
    {
        payload = pl;
        movingVeh::weight = gw; //MCdr1 = 1
        mpg = input_mpg;
        hourly_pay = pay;
    }

double cost_per_full_day(double cost_of_gas) //MCdr2 = 1
{ return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg); };

double total_weight(void) //MCdr3 = 1
// see, how to call different variables with same name
{
    cout<<"\nCalling appropriate member variable\n";
    cout<<"---->(movingVeh::weight)+(driver::weight)\n";
    cout<<"------------------------------------------\n";
    return ((movingVeh::weight) + (driver::weight));
};

// the main program
int main()
{
    drivenVehicle john_merc;

    john_merc.initialize(20000.0, 12000.0, 5.2, 12.50);
    // accessing the derived class method
    john_merc.driver::initialize(15.50, 250.0);
    // accessing the base class number two

cout << "The efficiency of the Merc is " << john_merc.efficiency() << " %\n";
cout << "The cost per mile for John to drive
is $" << john_merc.cost_per_mile() << "\n";
cout << "The cost per mile for John to drive
is "$" << john_merc.cost_per_mile() << "\n";
cout << "The cost per day for John to drive Merc
is $" << john_merc.cost_per_full_day(1.129) << "\n";
cout << "The total weight is " << john_merc.total_weight() << " ton\n";
return 0;
}