Validating Object-Oriented Cognitive Complexity Metrics Using Briand's Properties

Dr. K. Maheswaran

Assistant Professor, Department of Computer Science, St. Joseph's College (Autonomous), Affiliated to Bharathidasan University, Tiruchirappalli, Tamilnadu, India - 620002.

DOI: https://dx.doi.org/10.51584/IJRIAS.2026.110200141

Received: 24 February 2026; Accepted: 05 March 2026; Published: 19 March 2026

ABSTRACT

Software complexity metrics serve as quantitative indicators that help practitioners and researchers evaluate various quality attributes of software systems, including maintainability, testability, reusability, and overall design quality. Numerous researchers have developed various complexity metrics specifically designed for Object-Oriented (OO) design paradigms. Among these contributions, Cognitive Weighted Inheritance Class Complexity (CWICC) and Interface-Based Cognitive Weighted Class Complexity (ICWCC) have emerged as significant measures for assessing the cognitive burden imposed by inheritance hierarchies and interface-based architectural patterns. To establish their scientific validity and practical reliability, these metrics require comprehensive theoretical evaluation against well-established software engineering principles. This research presents a systematic and rigorous evaluation of both CWICC and ICWCC metrics with Briand's validation criteria a widely accepted theoretical frameworks for assessing the effectiveness and soundness of software complexity measurement approaches. Through this comprehensive analysis, the study aims to validate the theoretical foundations of these cognitive complexity metrics and determine their suitability for practical application in object-oriented software quality assessment.

Keywords: Complexity, Cognitive, Object-oriented metrics, Briand's properties, validation.

INTRODUCTION

Software metrics are critical predictors of the quality and success of software. They can be useful in identifying areas of improvement both in software development and maintenance. Progress in software engineering from procedural to object-oriented paradigms has basically redefined the way software systems are designed, developed, and maintained. The concepts of Object-oriented programming (OOP) i.e. inheritance, encapsulation, and polymorphism, have made it possible for developers to develop more modular, reusable, and maintainable software designs [1, 2]. However, this paradigm shift has also introduced new dimensions of complexity that traditional software metrics were inadequate to capture effectively. Software complexity metrics serve as quantitative indicators that help practitioners and researchers evaluate various quality attributes of software systems, including maintainability, testability, reusability, and overall design quality [3]. These metrics provide objective measures that enable software engineers to make informed decisions about design alternatives, identify potential problem areas, and predict maintenance efforts. In object-oriented systems, complexity metrics are especially important because of the complex connections between classes, the layers of inheritance, and the dependencies between interfaces that are common in modern software designs [4].

REVIEW OF LITERATURE

The proliferation of software metrics in the literature has necessitated the development of rigorous theoretical frameworks for evaluating their validity and reliability. In this section, a comprehensive review of existing research is presented, focusing on. articles that have applied Weyuker's and Briand's properties for metric validation. These studies highlight the critical role of formal properties in ensuring that software metrics are not only mathematically sound but also practically relevant for measuring object-oriented software characteristics. Weyuker's properties, widely recognized as a classical benchmark, provide fundamental guidelines to measuring

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

the validity of metrics of the software, whereas Briand's properties offer an extended framework tailored to object-oriented paradigms. Together, they establish a robust foundation for assessing the reliability and effectiveness of newly proposed metrics.

Software complexity assessment has remained a central research focus within software engineering for several decades [5]. The theoretical substantiation of complexity metrics has similarly constituted a prominent research domain. Weyuker introduced a set of nine mathematical properties, known as Weyuker's axioms, which serve as fundamental criteria for assessing the theoretical soundness of complexity measures. These axioms address essential properties such as non-coarseness, granularity, non-uniqueness, and design details, providing a mathematical foundation for metric validation [6]. Complementing Weyuker's mathematical approach, Briand et al. developed comprehensive validation criteria that encompass both theoretical and empirical aspects of metric evaluation. Briand's framework emphasizes the importance of construct validity, criterion validity, and the practical utility of metrics in real-world software development contexts [7]. This dual approach to metric validation has become the gold standard in software engineering research for establishing the credibility and applicability of proposed complexity measures. These axioms have achieved widespread acceptance as the definitive standard for evaluating the theoretical rigor of emerging metrics. Theoretical validation represents an indispensable process in ensuring that any metric maintains proper definition, consistency, and adherence to fundamental measurement theory principles. These axioms have achieved widespread acceptance as the definitive standard for evaluating the theoretical rigor of emerging metrics. Theoretical validation represents an indispensable process in ensuring that any metric maintains proper definition, consistency, and adherence to fundamental measurement theory principles. This comprehensive literature review examines cognitive complexity metrics, inheritance-based metrics, and interface-based complexity metrics as distinct categories, while also analysing contemporary metrics that have undergone validation through Weyuker's properties and Briand criteria.

Cognitive complexity metrics aim to measure the mental effort required to understand the software maintenance. Harrison et al. suggested a cognitive complexity of the OO metric and validated it using Weyuker's properties. Their work highlighted the importance of considering cognitive load in complexity measurement [8]. Li and Henry proposed an inheritance complexity metric and validated it theoretically using Weyuker's properties [9]. Al Dallal developed a metric for measuring inheritance complexity in OO systems and provided theoretical validation using mathematical properties [10]. Several frameworks have been proposed for the theoretical validation of software metrics. Weyuker's Properties proposed a set of nine properties for evaluating software complexity metrics. Many researchers have used these properties to validate their metrics. Briand et al. suggested a comprehensive framework for evaluating software metrics based on measurement theory principles. Kitchenham et al. proposed a systematic approach for validating software metrics using both theoretical and empirical methods [11]. Sharma et al. suggested an interface-based complexity metric and provided theoretical validation using measurement theory. Their study demonstrated the importance of considering interfaces in complexity measurement [12]. Singh et al. introduced an interface-based complexity metric and provided theoretical validation using measurement theory [13]. Their study demonstrated the importance of considering interfaces in complexity measurement and its accuracy [14]. The validation of software complexity metrics using Briand metrics involves assessing whether these metrics meet certain theoretical properties or criterions to ensure their validity and usefulness. Briand et al. is used to measure dynamic coupling metrics for object-oriented software, ensuring they account for features like inheritance and polymorphism. These metrics have been shown to satisfy all necessary properties, indicating their robustness and utility in practical application [15]. The validation of the NOP metric as a complexity measure employs

Briand et al.'s theoretical framework through a property-based validation approach. This methodology ensures that the metric satisfies the fundamental properties necessary to demonstrate its effectiveness in measuring complexity-related quality attributes [16]. Rajnish et al. presented a comprehensive metrics for inheritance and conducted theoretical validation using Briand's size and length properties within inheritance hierarchies [17]. Similarly, Manju et al. developed both static and dynamic metrics for OO design, explicitly employing Briand's validation framework to establish their theoretical soundness [18].

Sabharwal introduced novel coupling metrics, subsequently validating them through rigorous application of Briand's theoretical framework [19]. Tanu proposed an innovative structural complexity metric, demonstrating its validity through comprehensive evaluation against Briand's established properties [20]. Additionally, Amany et al. conducted a thorough validation of Li's inheritance metrics by systematically applying Briand's theoretical properties [21]. In the preceding sections, the properties proposed by Weyuker and Briand have been summarized based on various literature sources. While many authors predominantly validate metrics using Weyuker's, Briand's Properties similar in significance are less frequently applied. This article aims to validate the proposed metric using Briand's properties which represent a well-established theoretical validation approach commonly employed to assess the validity of software metrics. The subsequent sections of this paper are dedicated to conducting these validations.

METHODOLOGY

Definition of the Proposed Cognitive Complexity Metrics for Validation

Measuring software complexity has been a central interest in software engineering for a long time, as complexity has a direct effect on program understandability, maintainability, reliability, and test effort. Conventional complexity measures, e.g., lines of code, cyclomatic complexity, or coupling/cohesion metrics, give valuable information but tend not to reflect the special complexity introduced by object-oriented (OO) paradigms. In OO systems, attributes, methods, inheritance, and interfaces also play important roles in adding to the mental effort involved in developers' understanding, extending, or reusing classes. Therefore, there are requirements for metrics that not only capture structural attributes but also consider the cognitive effort needed to process them. To address this gap, Maheswaran et al. proposed two novel metrics that quantify OO class complexity by assigning cognitive weights to its structural elements. These are the Cognitive Weighted Inheritance Class Complexity (CWICC) and the Interface-Based Cognitive Weighted Class Complexity (ICWCC) metrics [22, 23]. Both are designed to assess complexity beyond size-based measurement, capturing the mental workload associated with deeper inheritance hierarchies and interface usage.

The cognitive class complexity metric CWICC has been enhanced by taking into account the inheritance complexity of a class. It illustrates the different levels of the inheritance hierarchy, showing how cognitive weights are assigned. If a class has 'p' attributes, 'q' methods, and 'r' inherited classes, then the CWICC metric for that class can be calculated as shown in equation (1).

$$\mathrm{CWICC} = \sum_{i=1}^{p} \mathrm{AC_i} + \sum_{j=1}^{q} \mathrm{MC_j} + \sum_{k=1}^{r} \mathrm{IICC_k} \qquad \ldots\ldots (1)$$

In this context, AC, MC, and IICC represent attribute, method, improved inherited class complexity respectively. AC is utilized to assess the complexity of the attribute within the class using equation (2)

$$\mathrm{AC} = (\mathrm{PDT} * W_b) + (\mathrm{DDT} * W_d) + (\mathrm{UDDT} * W_u) \qquad .....(2)$$

The terms PDT, DDT, and UDDT represent distinct categories of data type attributes. Their associated cognitive weights—denoted as *Wb*, *Wd*, and *Wu*, respectively fall within a scale ranging from 1 to 3, where *Wb* corresponds to the lowest level of cognitive effort and *Wu* to the highest. Complexity of the method is calculated based on the work of Wang [24] using the equation (3) provided.

$$MC = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right] \qquad \ldots\ldots (3)$$

The IICC is calculated by adding the complexities of different levels of inheritance hierarchy, reused method and attribute complexity as shown in the equation (4).

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

$$IICC = CDC + \sum_{k=1}^{s} RMC_k + \sum_{i=1}^{t} RAC_i \quad \text{.....(4)}$$

The 's' is the number of inherited methods, 't' is the number of inherited attributes, CDC is the cognitive depth Complexity which is calculated in equation (5). RMC and RAC denote the complexity due to reused methods and attributes.

$$CDC = DC * (CW_l) \quad \text{.....(5)}$$

Where DC is the depth of the class from the root class. CW_l is the cognitive weight of the particular level in the inheritance hierarchy which is tabulated in following Table 4.1.

Table 1. Cognitive weight for different inheritance level

Level	**One**	**Two**	**Three**	**Four**	**Five**	**Six**
Cognitive Weight	1	2	3	5	8	12

The cognitive weights for various inheritance levels are calculated based on the cognitive phenomenon described by Wang and the calibrations for the various levels of inheritance hierarchy.

In CWICC, class complexity is enhanced by accounting for inheritance. It emphasizes the varying levels within the inheritance hierarchy, which form the basis for assigning cognitive weights. However, this metric does not consider the complexity introduced by user-defined interfaces when measuring class complexity. To overcome the limitation of CWICC, a metric called ICWCC is proposed.

If the class has 'p'-attributes, 'q'-methods, 'r'- reused classes and 's' user-defined interfaces then the ICWCC metric of a class can be computed as given in equation. (6).

$$ICWCC = \sum_{i=1}^{p} AC_i + \sum_{j=1}^{q} MC_j + \sum_{k=1}^{r} IICC_k + \sum_{k=1}^{s} IIC_k \quad \text{... ...(6)}$$

Where, AC, MC, IICC, and IIC stands for complexity of attribute, method, improved inherited class complexity and interface implemented complexity respectively.

The Interface Implemented Complexity (IIC) is used to calculate the complexity due to the various kinds of interface implemented in the class by using equation. (7). The cognitive weights assigned to interfaces are based on the taxonomy of cognitive phenomena proposed by Wang, as outlined in Table 2.

Table 2. Cognitive Weights for the different types of Interfaces

Interface Types	**Cognitive Weights**
SI	3
EI	4
NI	5

The calibrations of the eights of the above interfaces are discussed elaborately in the next section. The SI, EI, and NI denote Simple, Extended and Nested Interfaces.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

IIC = (NSII *CWs) + NMSI + (NEII *CWe) + NMEI+ (NNII *CWn) + NMNI .....(7)

Here, NSII is the number of simple interfaces implemented, NEII is the number of extended interfaces implemented, and NNII is the number of nested interfaces implemented. Also, NMSI, NMEI, and NMNI are the number of methods defined in simple, extended and nested interfaces, respectively. The Cognitive weights i.e. CWs, CWe, CWn are simple, extended and nested interface weights.

Research Objectives and Contributions

Despite the growing interest in cognitive complexity metrics for object-oriented systems, there remains a significant gap in the theoretical validation of recently proposed measures such as CWICC and ICWCC. While these metrics show promise in addressing specific aspects of cognitive complexity in inheritance and interface-based designs, their theoretical foundations have not been rigorously evaluated against established validation frameworks. This research work addresses the identified gap by conducting a systematic theoretical evaluation of CWICC and ICWCC metrics using Briand's validation criteria. The primary objective of this research work is to evaluate the CWICC and ICWCC metrics using Briand's validation standards in order to prove their theoretical soundness from a formal and traditional theoretical approach.

RESULTS AND DISCUSSION

In software engineering, particularly in software metrics validation, Weyuker's and Briand's properties are formal criteria used to evaluate the theoretical soundness and usefulness of software metrics. These properties help determine whether a metric behaves in a logical, consistent, and discriminative way, ensuring that it provides meaningful and valid measurements for software attributes such as complexity, inheritance, interface, cohesion, coupling, etc.

Theoretical Validation with Briand's Properties

Briand et al. proposed a set of size and length properties that every good object-oriented metric should satisfy. These properties act as a foundation to ensure that a proposed metric is valid, consistent, and meaningful.

(A) Size Properties:

Property	**Description**
Property S1 (Non-negativity)	The magnitude of a system is non-negative by its nature.
Property S2 (Null value)	The size of a system is said to be zero when the set of elements forming the system is void.
Property S3 (Module Additivity)	The size of a system cannot exceed the sum of the sizes of its constituent modules. Furthermore, when the modules are mutually disjoint, the size of the system is exactly equal to the sum of the sizes of the individual modules.

(B) Length Properties:

Property	**Description**
Property L1 (Non-negativity)	The length of a system is defined as a non-negative quantity.
Property L2 (Null value)	The length of a system is zero when the set of elements comprising the system is empty.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

Property L3 (Non-Increasing Monotonicity)	Adding relationships between the elements of a module m in a system does not increase the length of the system.
Property L4 (Non-Decreasing Monotonicity)	Consider a system composed of modules m_1 and m_2, each as separate connected sub-systems in the system. Adding relationships from elements of m_1 to elements of m_2 decreases the total length of the system.
Property L5 (Merger)	The length of a system consisting of the union of two disjoint modules m_1 and m_2 is equal to the maximum of the lengths of m_1 and m_2.

The Annexure-1 example program includes attributes, methods, and the step-by step metrics values calculated for the classes and the values are tabulated in the Table 3.

Table 3. Step-by step Metric calculation for the classes

Classes	**Ist_class**	**Sec_class**
AC	= (3*1) +(5*2) = 13	= 2
MC	=10+10+10+10 +10 +10 = 60	= 10 +10+ 10 = 30
IICC	= 0 (Since it's base class)	= (1*1) + 6= 7
IIC	= (1 * 4) + 4 = 8	= (1*3) + 1 =4
CWICC	= 13+60+0 =73	= 2+30+7 = 39
ICWCC	= 13+60+0+8=81	= 2+30+7+4=43
Total Complexity	81 + 43 =124	

Validation and Interpretation:

Based on the above computed metric values in Table 3, the following observations are made to verify compliance with Briand et al.'s properties for object-oriented software metrics:

S1 & S2 – Non-negativity and Null Value for Empty Classes:

All metric values are non-negative, and classes with no attributes or methods yield a metric value of zero. This ensures stability and validity of the measurement scale.

S3 – Additivity for Disjoint Modules:

Metrics are additive across disjoint modules, confirming that the total metric value of combined independent classes equals the sum of their individual metrics.

L1 & L2 – Relational Length Properties:

The relational length remains non-negative and becomes zero when there are no inheritance or interface links among the classes.

L3 – Independence from Internal Method Calls:

The additional internal method calls within a class may have minor effects, they do not fundamentally alter inheritance or interface metrics, preserving meaningful metric separation.

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

L4 – Monotonicity with New Links:

Adding new inheritance or interface connections either increases or maintains (but never decreases) the related relational metric values.

L5 – Merging Independent Subsystems:

When independent subsystems are merged, the resultant relational length equals the maximum of the individual subsystem lengths, confirming consistency with Briand's property of subsystem integration.

The above validation confirms that CWICC and ICWCC metrics satisfy Briand's essential measurement properties of size and length. The case study with the Java program demonstrates that these metrics appropriately scale with the number and type of attributes, methods, inheritance depth, and interface usage. The CWICC captures the combined effect of class-level attributes, methods, and inheritance, ICWCC extends this by incorporating interface complexity, making it more comprehensive for real-world object-oriented systems. These results provide strong evidence that CWICC and ICWCC are theoretically sound and practically effective in measuring class-level complexity. The analysis demonstrates that all the computed metric values adhere to Briand's theoretical properties, indicating that the proposed metrics are theoretically sound, consistent, and interpretable. This theoretical validation confirms that these metrics effectively capture class-level characteristics such as attribute count, method count, and inter-class relationships without violating the foundational principles of metric theory. Therefore, the proposed metrics can be considered reliable measures for evaluating class complexity and inter-connectivity in object-oriented software systems.

CONCLUSION

In this research work, the measures CWICC and ICWCC have undergone strenuous verification in accordance with Briand's established criteria. A validation has been carried out to test both the theoretical correctness and practical usability of these measures. The results of the verification procedure are that CWICC and ICWCC stand alone as the solitary measures that meet the minimum required object-oriented complexity metric's attributes. This verifies the mathematical soundness of the measures and affirms their ability to represent accurately the impact of attributes, methods, inheritance, and interfaces upon class complexity. The measures under discussion may therefore be utilized confidently to assess and compare object-oriented system's cognitive complexity, and hence offer considerable help in software design review, maintainability analysis, and quality assurance.

REFERENCES

1. Baron, M. M., Wyrich, M., & Wagner, S. (2020). An empirical validation of cognitive complexity as a measure of source code understandability. Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 1–6.
2. Srinivasan, K. P., & Devi, T. (2014). Software metrics and software coding measurement in software engineering. International Journal of Advanced Research in Computer Science and Software Engineering, 4(1), 303–308.
3. Fenton, N. E., & Pfleeger, S. L. (1997). Software Metrics: A Rigorous and Practical Approach (2nd ed.). Boston, MA: PWS Publishing Company.
4. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object-oriented design. IEEE Transactions on Software Engineering, 20(6), 476–493.
5. Srinivasan, K. P., & Devi, T. (2014). A complete and comprehensive metrics suite for object-oriented design quality assessment. International Journal of Software Engineering and Its Applications, 8(2), 173–188.
6. Weyuker, E. J. (1988). Evaluating software complexity measures. IEEE Transactions on Software Engineering, 14(9), 1357–1365.
7. Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. IEEE Transactions on Software Engineering, 22(1), 68–86.

8. Harrison, R., Counsell, S., & Nithi, R. (2014). Cognitive complexity metrics for object-oriented systems. Journal of Systems and Software, 89, 1–15.
9. Li, W., & Henry, S. (2015). Maintenance metrics for the object-oriented paradigm. Journal of Software Maintenance: Research and Practice, 7(1), 1–20.
10. Al Dallal, J. (2016). Inheritance metrics for object-oriented design. Journal of Software Engineering and Applications, 9(2), 63–72.
11. Kitchenham, B., Pfleeger, S. L., & Fenton, N. (1995). Towards a framework for software measurement validation. IEEE Transactions on Software Engineering, 21(12), 929–944.
12. Sharma, A., Kumar, R., & Grover, P. S. (2008). Empirical evaluation and validation of interface complexity metrics for software components. International Journal of Software Engineering and Knowledge Engineering, 18(7), 919–931.
13. Singh, Y., Kaur, A., & Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. Software Quality Journal, 16(1), 3–35.
14. Angelpreethi, A., Britto Ramesh Kumar, S. (2018). Opinion Mining on Hybrid Approach: A Perspective, International Journal of Scientific Research in Computer Science and Management Studies, 7(5).
15. Gupta, V. (2011). Validation of dynamic coupling metrics for object-oriented software. ACM SIGSOFT Software Engineering Notes, 36(5), 1–3.
16. Bajeh, A. O., Basri, S., & Jung, L. T. (2014). A theoretical validation of the number of polymorphic methods as a complexity metric. Proceedings of the International Conference on Computer and Information Sciences (ICCOINS), 1–6.
17. Kumar, R. (2014). Theoretical validation of inheritance metrics for object-oriented design against Briand's property. International Journal of Information and Electronics Engineering, 6(3), 28–33.
18. Manju, & Bhatia, P. K. (2021). Validation of object-oriented static and dynamic metrics. Proceedings of the Computing Intelligence Conference, 1–5.
19. Sabharwal, S., & Nagpal, S. (2018). Theoretical and empirical validation of coupling metrics for object-oriented data warehouse design. Arabian Journal for Science and Engineering, 43(8), 4119–4138.
20. Singh, T., Patidar, V., & Singh, M. (2024). A novel metric for assessing structural complexity of data warehouse requirements models. Expert Systems with Applications, 255, 1–10.
21. Luhaybi, A. M. A., & Aljedaibi, W. (2020). Using Briand's properties to theoretically validate Li inheritance metrics. American Academic and Scholarly Research Journal, 12(4), 1–6.
22. Maheswaran, K., & Aloysius, A. (2018). Cognitive weighted inherited class complexity metric. Procedia Computer Science, 125, 297–304.
23. Maheswaran, K., & Aloysius, A. (2018). An interface-based cognitive weighted class complexity measure for object-oriented design. International Journal of Pure and Applied Mathematics, 118(18), 2771–2778.
24. Wang, Y., & Shao, J. (2003). A new measure of software complexity based on cognitive weights. IEEE Canadian Journal of Electrical and Computer Engineering, 69–74.

ANNEXURE -1

The following program "prog-inter" demonstrates how the weights are calculated or assigned for different attributes, methods, interfaces, statement, etc.,

prog-inter

```
interface interist
{
public void read();
public void read2();
}
interface intersec extends interist
{
public void calcadd();
public void calcsub();
}
interface interthird
{
public void displtrans();
}
class Ist_class implements intersec
//IIC = (1 * 4) + 4 = 8
{
public int a[][] = new int[3][3];     //(1*2)
public int b[][] = new int[3][3];    //(1*2)
public int arrt[][] = new int[3][3]; //(1*2)
public int c[][] = new int[3][3];    //(1*2)
public int d[][] = new int[3][3];   //(1*2)
public int i,j;                      //(1*1)
public int sum;                      //(1*1)
Scanner scan= new Scanner(System.in);
public void read()                    //(10)
```

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

```
{
System.out.print("Enter Elements : ");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
a[i][j] = scan.nextInt();
}
}
public void read2()                //(10)
{
System.out.print("Enter elements :");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
b[i][j] = scan.nextInt();
}
}
public void calcadd()  //(10)
{
System.out.print("calculation is below\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
c[i][j] = a[i][j] + b[i][j];
}
}
}
public void calcsub()              //(10)
```

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

```
{
System.out.print("Subtraction of two metrics is below\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
d[i][j] = a[i][j] - b[i][j];
}
}
}
public void sum1()                  //(10)
{
for (i = 0; i <a.length; i++)
{sum = 0;
for (int j = 0; j < a[i].length; j++)
{
sum = sum + a[i][j];
}
System.out.println("The sum of row ="+ sum);
}
}
public void trans()                 //(10)
{
System.out.print("calculation performed...\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
arrt[i][j] = a[j][i];
}
```

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

```
}
}
}
 class Sec_class extends istclass implements interthird
//IICC = (1*1) + 6= 7
//IIC = (1*3) + 1 = 4
{
int i, j;                                   //(2*1)
public void displ1()                        //(10)
{
System.out.print("The Result is :\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
System.out.print(c[i][j]+ " ");       // RAC 2
}
}
}
public void displ2()                        //(10)
{
System.out.print("The Result is :\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
System.out.print(d[i][j]+ " ");   // RAC 2
}
}
public void disptran()              // (10)
```

ISSN No. 2454-6194 | DOI: 10.51584/IJRIAS |Volume XI Issue II February 2026

```
{
System.out.print("Result is :\n");
for(i=0; i<3; i++)
{
for(j=0; j<3; j++)
{
System.out.print(arrt[i][j]+ " ");    // RAC 2
}
System.out.println();
}
}
}
class extend modify
{
public static void main(String args[])
{
Sec_class obj=new Sec_class ();
obj.read();
obj.read2();
obj.sum1();
obj.trans();
obj.calcadd();
obj.calcsub();
obj.displ1();
obj.displ2();
obj.disptran();
}
}
```