Journal of Harbin Engineering University
ISSN: 1006-7043

Vol 44 No. 9
September 2023

Impact of Type of Class on Inheritance Hierarchy

Vinay Singh[1], Love Kumar[2], Sandip Kulkarni[3], Vishal Khatri[4], Sumit Singh Sonkar[5]

[1]Faculty of Computing & Information Technology, Usha Martin University, Ranchi, Jharkhand
[2]Department of Computer Engineering and Applications, Mangalayatan University, Aligarh, U.P.
[3]Assistant Professor, Department of Computer Science, Himalayan University, Itanagar, Arunachal Pradesh
[4]Associate Professor, Department of Computing & Information Technology, Sikkim Professional University, Gangtok, Sikkim
[5]Assistant Professor, Department of Computer Science, Mangalayatan University, Jabalpur, MP
Email: vinaysinghuma@gmail.com, love.mittal@mangalayatan.edu.in

Abstract

One of the major and key features of Object-Oriented programming paradigm is the inheritance. Inheritance provides reusability, which improves code maintainability, understandability, modifiability. Since hierarchy of inheritance is a collection of various types of classes their spread makes depth and breadth. The level of the depth and breadth of inheritance hierarchies, DAGs are one of the major interest areas of researchers because as the number of classes increases, complexity also increases. Recent styles of programming practices use partially implemented class and pure abstract class as the superclasses in the class design hierarchies. The evaluation of the maintainability and understandability of inheritance hierarchy should be focused on the type of classes in its use.
The prime objective of this research paper is to elaborate the effect of different types of classes for the calculation of inheritance metrics of an inheritance hierarchy.
AMS Subject Classification: 68N01, 68N19, 68N30

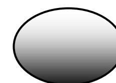
Keywords: Inheritance, DAG's, Maintainability, Understandability, Modifiability

1. INTRODUCTION

A class can be depicted as a programming abstraction which was introduced by Object –Oriented programming language structure (OOPLS). A Class definition has structure and behavior, which is used by its instances. One of the main concepts in OOPLS for creating a class is that implementation and interface must be separated. As far as design abstraction is concerned classes are of three kinds.

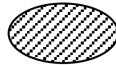
Concrete Classes: They have well-defined methods, which can be used directly by the instantiation of class. These classes can be inherited and it is possible to override them in a subclass. These classes can be super class or subclass. Concrete class directly comes from problem domain. The symbol for concrete class in this paper is denoted as –

Partially Implemented Class: These classes can only be used as superclass in which, some of the class methods may or may not have implementations. These classes are very useful in designing the framework which is required in terms of partial implementation so that the actual implementers of an application take this framework and only provide the subclasses and lot of logic is embedded in the framework itself. These classes are named as abstract class. The symbol for partially implemented class in this paper denoted as –



Pure Abstract Class: This category of classes, can only be used as a superclass due to no implementations of member functions which is inside these classes. These classes can be used when there is need to organize the classes into class hierarchies which show common behavior.

So superclass provides the common abstraction. The symbol for Pure Abstract Class in this paper is denoted as –



The traditional styles of measurement of software metrics were based on Procedure-Oriented approach. In the last two decades, Object-Oriented technique developed with great momentum and became in the mainstream of contemporary methods as a result of which object-oriented metrics have grown rapidly to estimate software which is based on object-oriented paradigm.

Among various object-oriented software matrices, the most acceptable metric is proposed by Chidamber and Kemerer [Chidamber and Kemerer, 1991, 1994] and is called the CK matrix. They proposed a set of indicators called the CK indicator set, which contains six class-based design indicators. Two of the six indicators they proposed belong to the category of inheritance indicators, namely: Depth of Inheritance Tree (DIT) and Number of Subalgebras (NOC). DIT is the longest length of the path between the root class and a given class in the inheritance hierarchy. In the Number of Children (NOC) metric, the sum of the total number of classes that immediately inherit from a given class is calculated. An extended version of the CK DIT metric can be seen in [HendersonSellers, 1996], the Average Inheritance Depth (AID) metric, where they propose the average (average) inheritance tree depth. Li [Li, 1998] proposed and evaluated CK measurement using Kitchenham's measurement evaluation framework and found all shortcomings of CK measurement. Li proposed a matrix called Number of ancestor classes (NAC) to calculate the total number of classes that can be affected by the class design. Tegraden et.al [Tegarden et.al, 1995] proposed a metric called the class-to-leaf depth (CLD) metric, which measures the maximum depth and the number of ancestors (NOA) metric in the inheritance hierarchy under the class . The total number of classes inherited directly or indirectly from a given class. The index of number of parents (NOP) proposed by Lake and Cook. Here, count the number of classes inherited from a given class anyway [Lake and Cook, 1994].

Reusability of classes will be better in case of a inheritance hierarchy which is deeper, consequently inherited classes which have higher coupling creates difficulties to maintain the system. While calculating the inheritance metrics one should consider the type of classes it uses. This paper proposed the weighted value of different type of classes in inheritance hierarchy tree for the management of object-oriented based software code.

2. EXISTING INHERITANCE METRICS

Various researchers have proposed various metrics. This paper is highly inspired and based upon inheritance metrics of researchers Chidamber and Kemerer [Chidamber and Kemerer, 1991, 1994] and Li [Li, 1998].

They evaluated the DIT of the class hierarchy. In the case of multiple inheritance, the depth is the maximum length from the desired class to the root class of the inheritance hierarchy tree. The calculation of the index can affect the number of ancestral classes in this class. However, they did not consider the class type when calculating the DIT metric, be it a concrete class, a partial implementation class, or a pure abstract class.

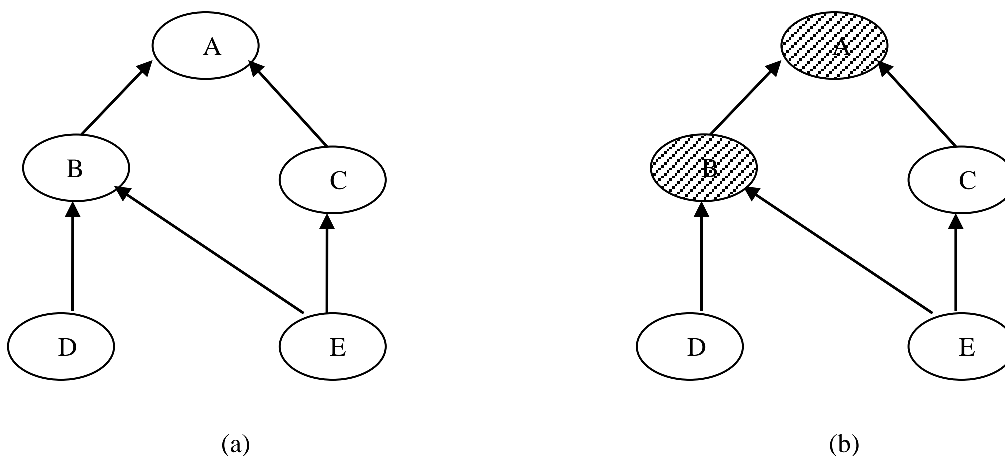


Fig. 1: Two Class Inheritance DAG's

In figure 1 there are two class inheritances DAG's, they both have the same structure. The difference between them is, in figure 1.b, class A and B are pure abstract class. Since CK does not consider in which types of classes in inheritance structure a class falls so the values of DIT of both the Fig.1a and Fig.1b will be equal. Class A is a root class there for DIT (A) = 0. The values of DIT for B and C will be DIT (B)=DIT(C) =1, due to the fact that the length to the root class of nodes B and C is one. Similarly, DIT value for classes D and E will be 2, since maximum length between classes D and E from root class A is 2. So, DIT (D)= DIT (E) = 2.

CK also proposed a metric called NOC (number of subcategories) of a class, which is the direct total count of class subcategories in the class DAG. NOC calculates the total number of classes or subclasses from direct parents inherited by the class. The idea behind NOC is to evaluate the potentially impacted classes in the total . In the above example of the class inheritance diagram in Figure 1a and Figure 1b, NOC (A) = NOC (B) = 2, because the number total children in grade 1 A and B are two. NOC (C) = 1 because it only has a first level. NOC (D) = NOC (E) = 0 because they are leaves. DIT shows the classes affected by the attributes of its parent class, and NOC shows a useful influence on the descendant classes. Li showed that when there are multiple classes of roots, the CK DIT index has some ambiguities.

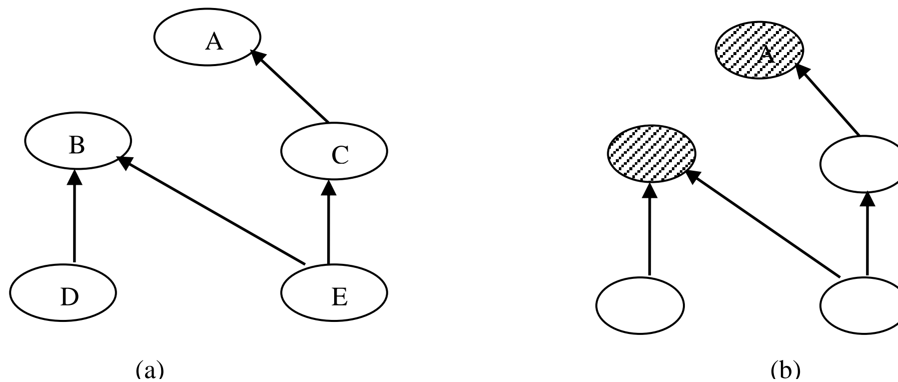


Fig. 2: Another Class Inheritance DAG

Given another example of DAG class inheritance, it has multiple roots (classes), as shown in Figure 2, it has two root classes A and B. So the maximum degree from class E to root class A is 2 (DIT (E) = 2) for root class B is 1 (DIT (E) = 1). This is an ambiguous situation.

Li proposed a metric to remove this ambiguity, it is called NAC (number of ancestor classes), which is a modified form of DIT. The NAC definition counts the total number of previous classes for which a class gets properties from the inheritance hierarchy.

Li proposed another inheritance metric, the NDC metric [Li, 1998], which is an alternative form of the CK NOC metric. Brito and Carapuca proposed three new genetic indicators, namely, total count of descendants (TPC), total count of parents (TPAC) and total ascending count (TAC) [Brito and Carapuca, 1994]. The TPC indicator calculates the number of classes that inherit directly or indirectly from a class. Taking into account that the class in Figure 1 inherits DAG, TPC (A) = 4, TPC (B) = 2, TPC (C) = 1, TPC (D) = TPC (E) = 0. From Figure 2 , TPC (A) = 3, TPC (B) = 1, TPC (C) = 0, and TPC (D) = 2 The Total Parent Count (TPAC) metric is the total number of superclasses inherited directly by a given class . It can be seen from Figure 1 that TPAC (A) = 0, TPAC (B) = TPAC (C) = TPAC (D) = 1 and TPAC (E) = 2.

The total number of superclasses inherited directly or indirectly by a A given class is called the total ascending count (TAC) of that class. From Figure 1, TAC (A) = 0, TAC (B) = TAC (C) = 1, TAC (D) = 2 and TAC (E) = 4. From Figure 2 TAC (A) = TAC (B) = 0, TAC (C) = TAC (D) = 1, TAC (E) = 3.

1.1. Metric for Understandability in Class Inheritance Hierarchies [Sheldon, Jerath and Chung, 2002]

Let us consider an inheritance DAG given in figure 3

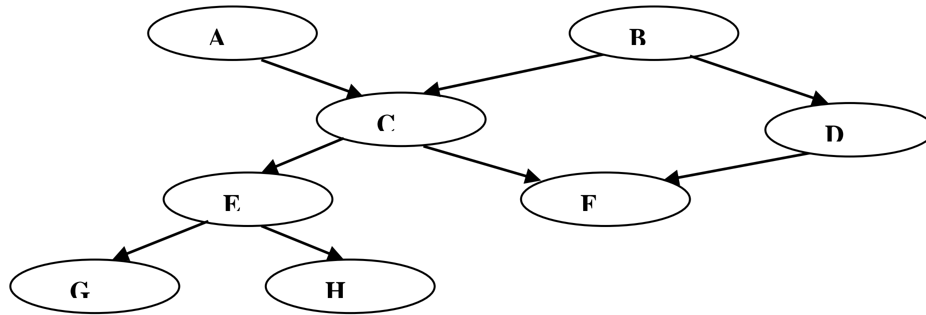


Fig. 3: Class Inheritance DAG with only Concrete Class

To read F, we must not only read F heuristically, but also C and D. In Figure , to understand C, we must first read A and B, because they are superclasses of C. Therefore, we must understand five categories, F, C, D, A and B. This value is PRED = (F) + 1.

Therefore, we define the intelligibility (U) of a class as follows:

U (Ci) = PRED (Ci) + 1

where Ci is the i-th class.

$$\sum_{i=i}^{t}(\text{PRED }(C_i) + 1)/t \qquad (2)$$

By using the above equations, the understandability of classes in figure 3 can been calculated as:

U (A) = PRED (A) + 1 = 0+1 = 1

U (B) = PRED (B) + 1= 0+1 = 1

U(C) = PRED (C) + 1= 2+1 = 3

U (D) = PRED (D) + 1= 1+1 = 2

U (E) = PRED (E) + 1= 3+1 = 4

The total intelligibility (TU) inherited by the DAG class is defined as follows:

$$\sum_{i=1}^{t}(PRED(Ci) + 1$$

where t is the number of classes

Average degree of understandability (AU) of class in hierarchical DAG is as follows:

U (F) = PRED (F) + 1= 4+1= 5

U (G) = PRED (G) + 1= 4+1 = 5

So, we have the total understandability (TU) = 1 + 1 + 3 + 2 + 4 + 5 + 5 + 5=26 and average understandability (AU) = 26/8=3.25.

If the definition of class C changes, you must first determine what the U(C) value of U is, and then you can change the definition of class C. Similarly, if the superclass E has any effect on the subclass G and/or the subclass H, these subclasses should be understood as modifying the E class. Define the modifiable degree (M) of a class as follows:

$$M \text{ of class } C_i = U(C_i) + SUCC(C_i)/2 \qquad (3)$$

Where Ci is the ith class.

The total degree of modifiability (TM) of a class hierarchy DAG is as follows:

$$TM\ of\ the\ class\ inheritance\ DAG = TU + \sum_{i=1}^{t} (SUCC\ C_i)/2) \quad (4)$$

Where t is the total number of classes in the class inheritance DAG.

To calculate Average degree of Modifiability (AM) of a class hierarchy DAG is given as follows:

$$AM\ of\ the\ class\ inheritance\ DAG = AU + (\sum_{i=1}^{t} (SUCC(C_i)/2))/t \quad (5)$$

The maintainability of classes given in figure 3 is calculated by expression 3,

Therefore, by expression (4),

TM = 3.5+4+5+2.5+5+5+5+5 = 35

and, by expression (5), AM = 35/8 = 4.38
The number which comes from expression (5) is the average degree of modifiability of the class in class hierarchy of given figure 3.

The average inheritance depth (AID) of a class has an expression given as below:

$$(\sum depth\ of\ each\ class) / number\ of\ classes$$

Considering two-class inheritance DAG's in Figure 7:

1.2. Henderson-Seller's AID

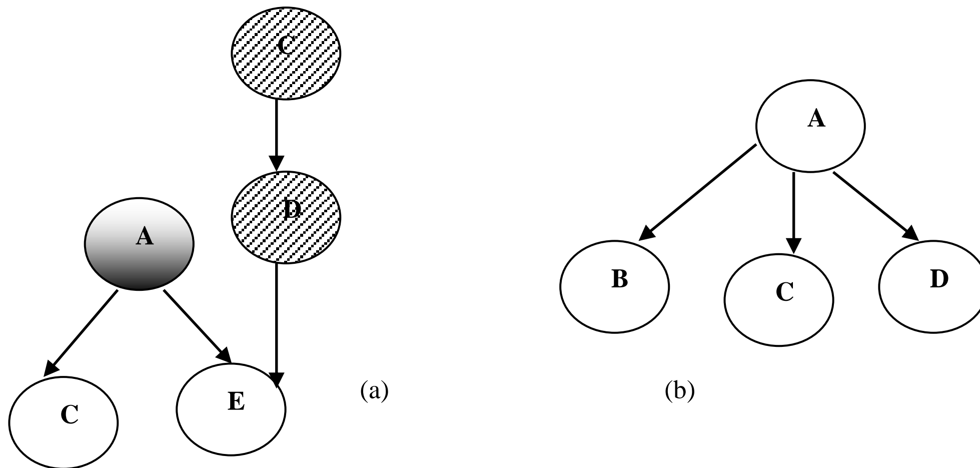


Fig. 4: Another Class Inheritance DAG's

AID of Fig. 7a: = (0+0+1+1+ (1+2)/2)/5=0.7

AID of Fig 7b: = (0+1+1+1)/4=0.75

The average inheritance depth value indicates that DAG in Figure 7a is easier than the DAG in Figure 7b. The calculation of AU and AM are as follows:

AU of Fig. 7a = (1+2+2+1+1)/5=1.75

AM of Fig.7a = 1.75+ (2/2+1/2+1/2)/5=2.15

AU of Fig. 7a = (1+2+2+2)/4=1.75

AM of Fig. 7b =1.75+ (3/2)/5=2.05

It has been observed that the maintainability of Figure7a is better than Fig. 7b, which is also confirmed by the AID value as above. If all the classes in Figure 7a are considered as concrete

classes (then the AU of figure 7a is (AU=2) and figure 7b (AU=1.75)). Figure 7b is more understandable than the figure. 7a. The above calculation also strengthens our belief that we should take the weighted complexity value of the type of class while evaluating the maintainability (understandability and modifiability) of inheritance DAG.

2. PROPOSED WORK

In this research metric of inheritance hierarchy for understandability and maintainability, we have proposed in the context of types of classes viz. concrete class, and non-concrete class.

According to the concept of object-oriented paradigm classes are of two types: concrete and concrete class.

Non-Concrete class: In this type of classes they may have partial implementation or no implementation of the class. There are two types of classes in this category

1) Pure Abstract Class – In this category of classes, they have no implementation. Definition of the class is empty.
2) Abstract Class- In this category there is a partial implementation of the class.

2.1. Effect Of Type Of Classes In Calculation Of Inheritance Metrics

Apart from previous research done on inheritance metrics in this research, it is found that there is a significant effect of types of classes (concrete and non-concrete class). We developed a metric for understandability and maintainability separately on basis of their types.

It can be observed that complete implementation of a class in the context of a concrete class has more complexity than non-concrete class (they have no implementation or partial implementation). Similarly, in the context of non-concrete class, since pure abstract class has no implementation, on the other hand abstract class has a partial implementation; therefore, the abstract class contains more complexity than Pure Abstract Class. On the basis of this analysis, this paper proposed weight for each type of classes for evaluating metrics.

The weighted value of class complexity:

Type of class	Weight
Pure Abstract Class	0
Abstract class	0.5
Concrete class	1

Proposed Metrics for Understandability:

U(Ci) = (PRED (Ci) * CV(Ci)) +1 (6)

Where CV (Ci) is the complexity of ith class.

The expression for total understandability of (TU) of an inheritance DAG is defined as follows:

$$\sum_{i=1}^{t}(PRED(C_i) * CV(C_i)) \quad (7)$$

The expression for the average degree of understandability (AU) of an inheritance DAG is as follows:

$$(\sum_{i=1}^{t}((PRED(C_i) * CV(C_i)))/t \quad (8)$$

Proposed Metrics for Modifiability:

The degree of modifiability (M) of a class can be measured as follows:

$$\text{M of a class Ci} = U(C_i) + (SUCC(C_i) - Number\ of\ NonConcrete\ Class)/2 \quad (9)$$

Here, Ci is the ith class. The expression for total degree of modifiability (TM) of a class inheritance DAG is

$$TU + \sum_{i=1}^{t}((SUCC(C_i) \text{-} Number\ of\ NonConcrete\ Class)/2) \quad (10)$$

Where t is the sum of the total number of classes in the inheritance Direct Acyclic Graph.

The expression for the degree of average modifiability (AM) in inheritance DAG is as follows:

AM for inheritance DAG =

$$AU + (\sum_{i=1}^{t}((SUCC(C_i) \text{-} Number\ of\ NonConcrete\ Class)/2)/t \quad (11)$$

Let, consider the inheritance DAG given in figure 4. Figure 4 is the same DAG Figure 3 with the change in the type of classes.

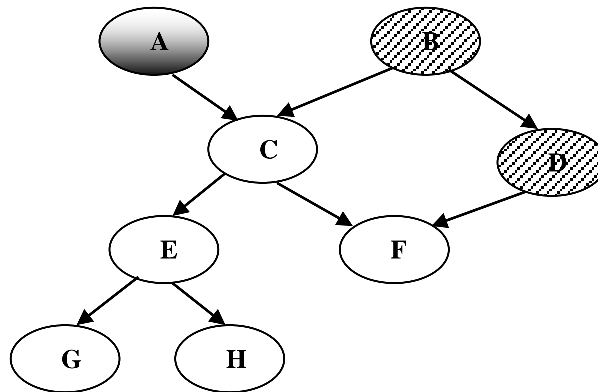


Fig. 5: Change of type of class in Figure 3

Table 1 for classes with weights

Concrete class	Pure Abstract Class	Abstract Class
C, E, F, G, H	B, D	A
1	0	0.5

The understandability of figure 4 is calculated as from expression 6:

U(Ci) = (PRED (Ci) * CV(Ci)) +1

U (A) = (0*0.5) +1 =1

U (B) = (0*0) +1=1

U(C) = (1*0.5) + (1*0) +1=2.5

U (D) = (1*0) +1=1

U (E) = (1*1) + (1*0) + (1*1) + (1*0) +1=2

U (F) = (1*0) + (1*0) + (1*1) + (1*0.5) + 1=2.5

U (G) = (1*1) + (1*1) + (1*0) + (1*0.5) +1=3.5

U (H) = (1*1) + (1*1) + (1*0) + (1*0.5) =3.5

Total Understandability (TU) from expression (7)

$$\sum_{i=1}^{t}(PRED(C_i) * CV(C_i))$$

So, TU= 1+1+2.5+1+2+2.5+3.5+3.5 =17

AU of a class inheritance from expression (8)

$$(\sum_{i=1}^{t}((PRED(C_i) * CV(C_i)))/t$$

So, AU =17/8=2.125

The modifiability of the inheritance DAG in Figure 4 is calculated as from expression (9)

$$\text{M of a class Ci } = U(C_i) + (SUCC(C_i) - \textit{Number of NonConcrete Class})/2$$

M(A)= (1+(5/2) – 0) =3.5

M(B)= (1+(6/2) – 1) =3

M(C) = (2.5 + (4/2) -0) =4.5

M(D)= (1+(1/2) -0) =1.5

M(E) = (2+(2/2) – 0) =3

M(F)= (2.5 + (0/2) -0) =2.5

M(G)= (3.5 + (0/2) -0) =3.5

M(H)= (3.5 +(0/2) – 0) =3.5

Total Modifiability (TM) calculated from expression (10) is given below

$$TU + \sum_{i=1}^{t}((SUCC(C_i) - \textit{Number of NonConcrete Class})/2)$$

TM= 3.5+3+4.5+1.5+3+2.5+3.5+3.5 = 25

Average Modifiability (AM) from expression (11)

$$AU + (\sum_{i=1}^{t}((SUCC(C_i) - \textit{Number of NonConcrete Class})/2)/t$$

AM=25/8 =3.12

It has been observed that the AM value of inheritance DAG in figure. 3 is 4.33. whereas, the same inheritance DAG in figure 4 with the change in the type of superclass to non-concrete reduces the AM value to 3.12.

Table.2. Comparison (Concrete Class and Interface)

Using DAG	AM
Fig.3(Concrete Class)	4.33
Fig.4(Pure Abstract Class)	**3.12**

It seems that the change of type of class to non-concrete class reduces the maintainability. It is also be noticed that in Figure 4 when changes in the type of class A from pure abstract class to abstract class, the AM value becomes 3.37.

Table.3. Comparison (Concrete Class and Abstract Class)

Using DAG	AM
Fig.3(Concrete Class)	4.33
Figure 4(Abstract Class) Node A	**3.37**

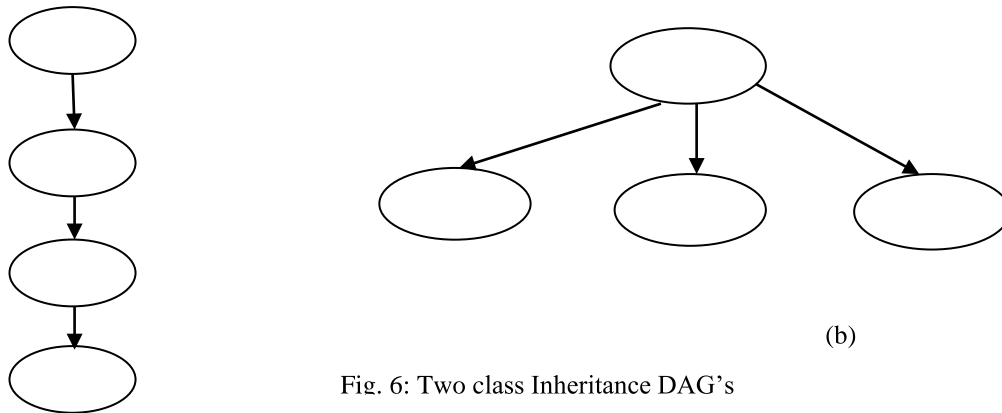


Fig. 6: Two class Inheritance DAG's

AU of Figure 5 (a): ((1+2+3+4)/4 =10/4=2.5

AM of Figure 5 (a): (2.5 +1.5+1+0.5)/4=**3.25**

AU of Figure 5 (b): (1+2+2+2)/4=7/4=1.75

AM of Figure 5 (b): (2.5+2+2+2)/4=8.5/4=**2.13**

Since 2.13 <3.25, the intelligibility and modifiability of the DAG in Figure 5 (b) has better results than in Figure 5 (a). This supports CK's observation that "designers may tend to keep the hierarchy of inheritance at a superficial level and forego reuse through inheritance to simplify understanding" [Chidamber and Kemerer, 1994].

Taking another example of inheritance DAG in figure 6 in which the hierarchy is same but types of classes are changed with A, B, C are pure abstract classes in figure 6.a while in figure 6.b only A is pure abstract class.

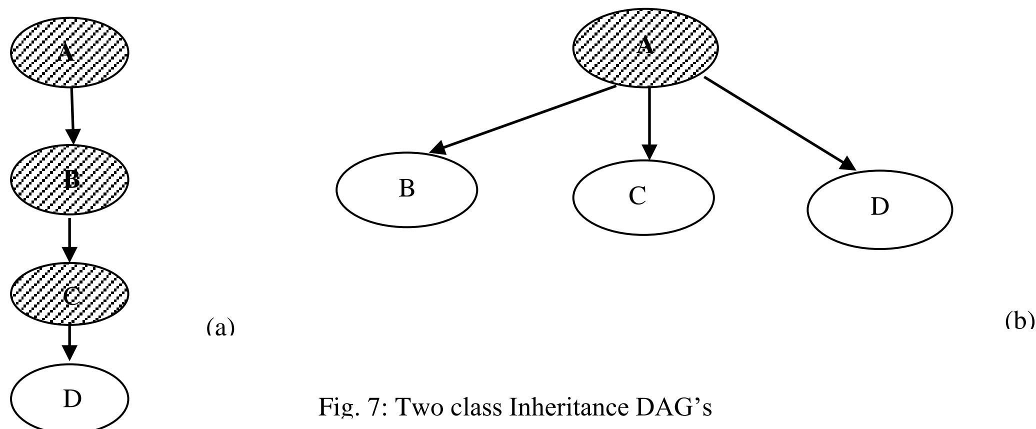


Fig. 7: Two class Inheritance DAG's

AU of Figure 6(a): (1+1+1+1)/4 =1

AM of Figure 6(a): (1.5+1.5+1.5+1)/4=1.37

AU of Figure 6(b): (1+1+1+1+1)/4=1.25

AM of Figure 6(b): (2.5+1+1+1)/4=1.37

It can be seen in Figure 5 and Figure 6 that changing the class type changes the intelligibility and modifiability values. The compressibility of Fig. 6 (a) is better than that of Fig. 6 (b), and the modifiability of Fig. 6 (a) and Fig. 6 (b) are the same.

2.2. Verification Based on Some Projects

We verified the correctness of our work on the basis of some projects and compared our work with that of the existing works for calculating the object-oriented programming metrics. We found that by giving different weights to different classes we have a significant improvement in class metrics. The comparison is shown below in table 4

Table 4: Comparison of the Proposed Work with Existing Models

Project	No. Of Classes	TU (Proposed)	AU (Proposed)	TM (Proposed)	AM (Proposed)	TU (Existing)	AU (Existing)	TM (Existing)	AM (Existing)
Project 1	10	21	2.1	30	3	29	2.9	38	3.8
Project 2	7	19	2.7	27	3.8	28	4	33	4.7
Project 3	5	9	1.8	16	3.2	21	4.2	22	4.4

The results show that our work considerably reduces the Average Understandability (AU) and Average Modifiability (AM). The results as shown by in the above table are graphically shown below:

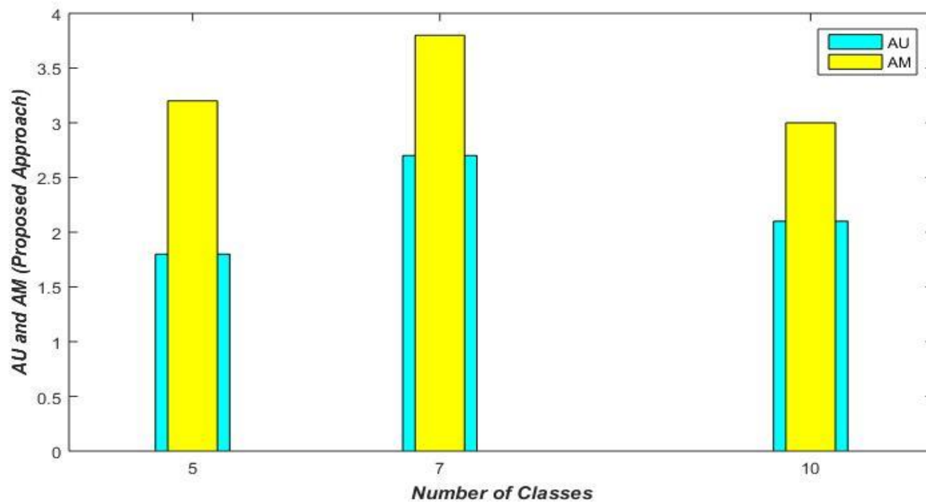


Fig. 8: AU and AM for Project 1, Project 2 and Project 3 (Proposed Approach)

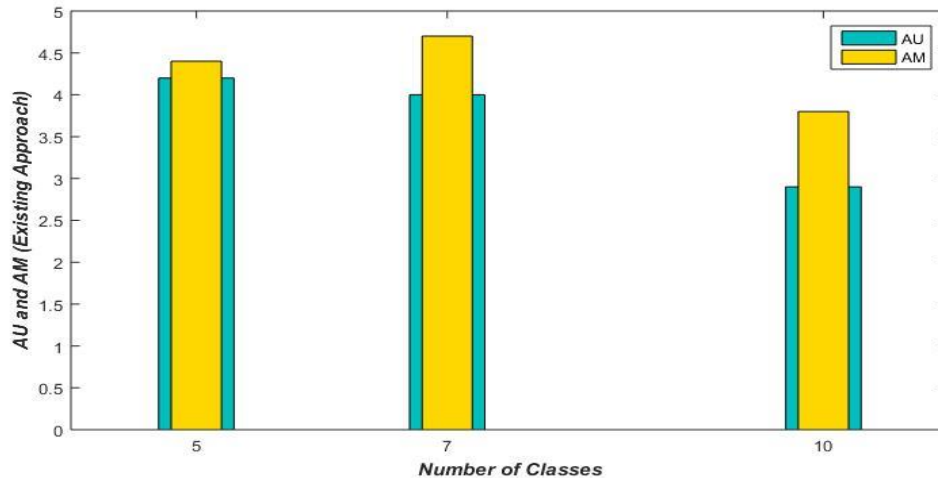


Fig. 9: AU and AM for Project 1, Project 2 and Project 3 (Existing Approach)

CONCLUSION

The types of classes in the inheritance hierarchy play a key role in the determination of understandability, maintainability, and modifiability of software code. Based on this fact, in this paper two simple and heuristic validated matrices have been proposed with the different weighted values for concrete class and non-concrete classes in inheritance hierarchy.

The understandability and modifiability of inheritance DAG in figure 5(b) is better than figure 5(a), which supports the CK's observation in Chidamber and Kemerer, 1994. Whereas change of type of classes in figure 6 contradicts CK's observation, that inheritance hierarchies should be kept shallow. In this research it is observed that the depth and breadth is not always the indicator of maintainability in inheritance hierarchy. It can be observed from figure 6 that deeper inheritance is not always bad for understandability, figure 6a is more maintainable than Figure 6b regardless of depth and breadth in inheritance hierarchy. On the basis of figure 3 and figure 4 it can also be figured out that in same inheritance hierarchy with the change in the type of classes can increase understandability.

The software project managers, developers and designers can be benefited from the use of this approach of measuring the maintainability of inheritance hierarchy in their programming. It is concluded that measurement of inheritance hierarchy should consider the type of classes it uses. It is also recommended that the use of non-concrete classes as superclass increases understandability of inheritance hierarchy.

References

[1] [Brito and Carapuca, 1994] Brito, A.F., and Carapuca, R (1994). "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework", Journal of System Software, Vol. 26, 1994, pp. 87-96

[2] [Chidamber and Kemerer, 1991] Chidamber, S. R., and Kemerer, C. F. Towards (1991) A Metric Suite for Object-Oriented Design, in *Proceeding. Sixth OOPSLA Conf*, 197-211.

[3] [Chidamber and Kemerer, 1994]Chidamber, S. R., and Kemerer, C. F. (1994) A Metric Suite for Object-Oriented Design, *IEEE Trans. on Software Engineering*, 20, 6(1994), 476-493.

[4] [Lake and Cook, 1994] Lake, A., Cook c., (1994) Use of Factor Analysis to Develop OOP Software complexity Metrics. In proc. of the Annual Oregon Workshop on Software Metrics, April 10-12, 1994, Silver Falls Oregon.

[5] [Li, 1998] Li, W. (1998) "Another metric suite for object-oriented programming,"

The Journal of Systems and Software 44(2): 155-162.

[6] Sherje, N.P., Agrawal, S.A., Umbarkar, A.M., Dharme, A.M., Dhabliya, D. Experimental evaluation of Mechatronics based cushioning performance in hydraulic cylinder (2021) Materials Today: Proceedings, .

[7] [Henderson-Sellers, 1996] Henderson-Sellers, B.,(1996) "Object-Oriented metrics : measures of complexity," *Prentice-Hall*, pp.142-147

[8] [Sheldon, Jerath, and Chung, 2002] Sheldon, F.T, Jerath, K and Chung, H,(2002) "Metrics for maintainability of class inheritance hierarchies", Journal of Software Maintenance and Evolution: Research and Practice, 14, pp. 1-14.

[9] [Tegarden et.al, 1995] Tegarden, D.P., Sheetz, S.D., Monarchi, D.E., "A software complexity model of object-oriented system". Decision support systems 1995; 13 (3-4):241-262.

[10] Agrawal, S.A., Umbarkar, A.M., Sherie, N.P., Dharme, A.M., Dhabliya, D. Statistical study of mechanical properties for corn fiber with reinforced of polypropylene fiber matrix composite (2021) Materials Today: Proceedings, .