Advanced Object Oriented Metrics for Process Measurement

Shreya Gupta
Indian Institute of Information Technology
Deoghat, Jhalwa, Allahabad, India
gupta.shreya29@gmail.com

Ratna Sanyal
Indian Institute of Information Technology
Deoghat, Jhalwa, Allahabad, India
rsanyal@iiita.ac.in

Abstract— **Process improvement requires measurement of specific attributes of process. Measurement of a process gives us a clear insight into the system. It provides effective ways of estimation and evaluation. Then, it is essential to develop a set metrics covering the attributes. Computed measures are used as indicators for process improvement areas. These indications if incorporated into the software development, will lead to development of an effective and reliable system. Mood metrics has defined some indicators for inheritance like Attribute Inheritance Factor (AIF), Method Inheritance Factor (MIF), and for hiding are Attribute Hiding Factor (AHF), Method Hiding Factor (MHF). We are proposing extensions to these metrics. These extensions are more specific and give a better hint towards inheritance and hiding properties.**

Keywords-Mood Metrics; Attribute Inheritance Factor; Method Inheritance Factor; Attribute Hiding Factor; Method Hiding Factor.

I. INTRODUCTION

Object orientation aims to model a system [1]. They reflect a natural view and understanding of the system. Using object modeling, a system is represented as number of objects and their interaction. Objects are categorized into classes along with their respective behavioral properties [2]. Inheritance provides the facility for classes to inherit the behavioral properties of other classes. Encapsulation packages functions and data in a class. Representing essential features with exclusion of background explanations is called abstraction [3].

Object Oriented Software Paradigm gives the way for effective reuse of program components. The process of reuse expedites the software development and thereby resulting in high quality work in minimum time. They are easy to understand, adapt and scale because of modular structure, relatively low coupling and high cohesion. Merely applying object oriented programming will not reap great results. It is the combination of object oriented domain analysis, requirement analysis, object oriented design, database systems and computer aided software engineering that will lead to best results.

If software is developed without any proper measurement activities, the resulting product could be unreliable, inefficient and non-maintainable. We need to realize the ideology that software needs to be engineered. For this, standard engineering principles and guidelines are to be established. Software metrics come into play as quantify the attributes in the development. Errors undetected in a development phase are passed in the next phase. Relative cost of fixing it increases many times. Therefore, tracing errors early in lifecycle and fixing them are essential.

Second section describes the prior work in the field of software metrics particularly C.K. Metrics and Mood metrics. Since the research paper is proposing an extension to the AIF, MIF, AHF and MHF, a detailed explanation of these metrics has been provided with reference to published research papers. Third section of the paper proposes the extension to AIF, MIF, AHF and MHF computation along with the extended formulas for the same. Fourth section is Result and Analysis section for AIF, MIF, AHF and MHF, considering a system and showing the variation in values obtained by the original formulas and the extended ones. Furthermore, a case study has been taken to validate the results of these metrics.

II. PRIOR WORK

Six software metrics were proposed for object oriented design, known as C. K. Metrics [4]. These metrics are Response of a Class (RFC), coupling between the objects (CBO), Weighted Methods per Class (WMC), Number of Children (NOC), Lack of Cohesion Methods (LCOM) and Depth of Inheritance Tree (DIT). The empirical evidence specifies how object oriented metrics determine software defects is described [5].

Mood Metrics (Metrics for Object Oriented Design) were proposed by Abreu as described [6]. These metrics aim to evaluate object oriented principles in the software code. It considers inheritance factor which computes attribute inheritance factor and method inheritance factor, encapsulation factor which computes attribute hiding factor and method hiding factor. All of these metrics result in the probability value between 0 and 1.

In the following subsections, we have explained and mentioned the formulas of the existing parts of MOOD metrics.

A. Attribute Inheritance Factor (AIF)

Attribute Inheritance Factor (AIF) is the ratio of two measurements. Numerator represents the sum of number of inherited attributes of all classes in system and denominator represents sum of number of available attributes which may be local or inherited for all classes in system. It expresses the level of reuse in the system. A threshold is maintained for AIF measure that is roughly around 50%. Higher values of AIF indicate high inheritance level thereby leading to greater coupling and reducing the possibility of reuse. MOOD Metrics propose the computation of AIF [6] as given below:

$$\textbf{AIF} = \sum_{i=1}^{TC} A_i(C_i) \Big/ \sum_{i=1}^{TC} A_a(C_i) \qquad (1)$$

where $A_a(C_i) = A_d(C_i) + A_i(C_i)$
$A_i\,(C_i)$ is the count of attributes that are inherited.
$A_d(C_i)$ is the count of defined attributes. These attributes can be of any access modifier.
$A_a(C_i)$ is the count of attributes that can be referenced by class C_i
TC - total count of classes in system/ package.

B. *Method Inheritance Factor (MIF)*

Method Inheritance Factor (MIF) is the ratio of two measurements. Numerator represents the sum of number of inherited methods of all classes in system and denominator represents sum of number of available methods which may be local or inherited for all classes in system. Method Inheritance Acceptable range is 20% to 80%. It highly depends on the design pattern that we follow. High values of MIF indicate superfluous inheritance and low values indicate heavy use of overrides or lack of inheritance. MOOD Metrics propose the computation of MIF [6] as given below:

$$\textbf{MIF} = \sum_{i=1}^{TC} M_i(C_i) \Big/ \sum_{i=1}^{TC} M_a(C_i) \qquad (2)$$

where $M_a(C_i) = M_d(C_i) + M_i(C_i)$
$M_i\,(C_i)$ is the count of methods that are inherited. These methods should not be overridden.
$M_d\,(C_i)$ is the count of defined non-abstract methods. These methods can be of any access modifier.
$M_a\,(C_i)$ is the count of methods that can be called by class C_i.
TC - total count of classes in system/ package.

C. *Attribute Hiding Factor (AHF)*

AHF measures the extent of encapsulation of attributes in a system. Firstly, it will calculate the visibility of each attribute with respect to each class. Visibility function assigns 1 for each class, if the attribute is visible from those classes and 0 if not visible. Visibility measure for class in which attribute is present itself is considered to be 0. It sums up the visibility for a particular attribute and then divides by the (total no. of classes minus 1). Likewise, the visibility of each attribute is calculated and then values are substituted in AHF formula. Thus, AHF represents the average amount of hiding of attributes among all classes in system. Visibility of private attributes is always zero. Protected attributes act as a public attribute in the package to which the attribute belongs, and are visible only in the subclasses in other packages. Public attributes are visible to all classes in the system. If all the attributes are private, then AHF=100% and if all the attributes are public, AHF is 0%. MOOD Metrics propose the computation of AHF [6] as given below:

$$\textbf{AHF} = \sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} \textbf{(1-V(A}_{\textbf{mi}}\textbf{))} \Big/ \sum_{i=1}^{TC} A_d(C_i) \qquad (3)$$

where

$$\text{V}(A_{mi}) = \sum_{i=1}^{TC} is\_visible(A_{mi}, C_j) \,/\, (\text{TC} - 1)$$

and

$$\text{is\_visible}(A_{mi},C_j) = \begin{cases} 1 \;\; iff \;\; j \neq i \; and \; C_j \, may \; reference \; A_{mi} \\ 0 \; otherwise \end{cases}$$

$A_d(C_i)$ is the count of defined attributes. These attributes can be of any access modifier. They should not be inherited ones.

D. *Method Hiding Factor (MHF)*

It measures the extent of encapsulation of methods in a system. Firstly, it will calculate the visibility of methods with respect to each class. Visibility function assigns 1 for each class, if the method is visible from those classes and 0 if not visible. Visibility measure for the class in which method is present itself is considered to be 0. It sums up the visibility for a particular method and then divides by the (total no. of classes minus 1). Likewise the visibility of each method is calculated and then values are substituted in MHF formula. Thus, MHF represents the average amount of hiding of methods among all classes in system. Visibility of private methods is always zero. Protected methods act as a public method in the package to which the method belongs, and are visible only in the subclasses in other packages. Public methods are visible to all classes in the system. If all the methods are private, then MHF=100% and if all the methods are public MHF is 0%. MOOD Metrics propose the computation of MHF [6] as given below:

$$\textbf{MHF} = \sum_{i=1}^{TC} \sum_{m=1}^{Md(Ci)} \textbf{(1-V(M}_{\textbf{mi}}\textbf{))} \Big/ \sum_{i=1}^{TC} M_d(C_i) \qquad (4)$$

where

$$\text{V}(M_{mi}) = \sum_{i=1}^{TC} is\_visible(M_{mi}, C_j) \,/\, (\text{TC} - 1)$$

and

$$\text{is\_visible}(M_{mi},C_j) = \begin{cases} 1 \;\; iff \;\; j \neq i \; and \; C_j \, may \; reference \; M_{mi} \\ 0 \; otherwise \end{cases}$$

$M_d(C_i)$ is the count of methods and constructors. These methods can be of any access modifier. They should not be abstract or inherited.

III. PROPOSED EXTENSION

A. *Extension in AIF and MIF*

Problem with the AIF/MIF formula is that it considers the count of members a class can reference in a system or a package. But, when we calculate AIF/MIF for each class, members outside the class (except for the members that are inherited) are not to be considered. Justification is that denominator of the formula of AIF and MIF states that "Total no. of members that a class C_i can reference", all the members that are public can be referenced by a class, no matter whether it is in its same package or outside the package. Even protected members act as public members in their own package. Thus, while calculating AIF, MIF the count of uncoupled members in the denominator should not be considered, because access to public, protected members does not reflect the measure of inheritance factor.

Thus, an extension to the empirical formula is proposed by us. For denominator, consider the members of ancestor classes of class C_i and the members defined inside class C_i only.If a class "x" is present in same package as that of class C_i and has public members, but has no interaction with the class C_i, then members of class "x" are not considered.

When a class inherits considerable number of members from the ancestor classes, it will contribute to a high measure of AIF, MIF. When a class redefines the ancestor members and adds the new members will always contribute to a low measure of AIF, MIF. The extended equation for AIF is given below:

$$\textbf{AIF extended} = \sum_{i=1}^{TC} A_i(C_i) \,/\, \sum_{i=1}^{TC} A_{ex}(C_i) \qquad (5)$$

where $A_{ex}(C_i) = A_d(C_i) + A_i(C_i)$

$A_i(C_i)$ is the count of attributes that are inherited.
$A_d(C_i)$ is the count of defined attributes. These attributes can be of any access modifier.
$A_{ex}(C_i)$ is the extended variable. It is the count of attributes that can be referenced by class C_i considering the attributes of ancestor classes of class C_i and the attributes defined inside class C_i only.

The extended equation for MIF is given below:

$$\textbf{MIF extended} = \sum_{i=1}^{TC} M_i(C_i) \,/\, \sum_{i=1}^{TC} M_{ex}(C_i) \qquad (6)$$

where $M_{ex}(C_i) = M_d(C_i) + M_i(C_i)$

$M_i(C_i)$ is the count of methods that are inherited. These methods should not be overridden.
$M_d(C_i)$ is the count of defined non-abstract methods. These methods can be of any access modifier.
$M_{ex}(C_i)$ is the extended variable. It is the count of methods that can be called b class C_i considering the methods of ancestor classes of class C_i and the methods defined inside class C_i only.

Thus, AIF extended and MIF extended give an accurate idea about the actual level of inheritance that exists in the code. If the level of inheritance is high, then it is a hindrance to the reusability, maintainability and understandability of system. It will be difficult to reuse the modules of code into some other system because of its dependency on other modules.

B. *Extension in AHF and MHF*

Original AHF equation consists of visibility function that checks that if class may reference the attribute in consideration. But, in the extension that I have proposed, it checks whether actually the class has referenced the attribute or not. This extension in AHF is more specific in nature and gives a clear hint of the hiding factor. It also checks for a good design characteristic that attributes of a class should accessed by methods of the class only. If they are directly accessed by the objects of some other class, then design is not stable. The extended equation for AHF is given below:

$$\textbf{AHF extended} = \sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} \textbf{(1-V(A}_{\textbf{ex}}\textbf{))} \,/\, \sum_{i=1}^{TC} A_d(C_i) \qquad (7)$$

where

$$V(A_{ex}) = \sum_{i=1}^{TC} is\_visible(A_{ex}, C_j) \,/\, (TC - 1)$$

and

$$is\_visible(A_{ex},C_j) = \begin{cases} 1 \ iff \ \ j \neq i \ and \ C_j \, did \ reference \ A_{mi} \\ 0 \ otherwise \end{cases}$$

$A_d(C_i)$ is the count of defined attributes. These attributes can be of any access modifier. They should not be inherited attributes.

Original MHF equation consists of visibility function that checks that if class may reference the method in consideration. Same extension goes with MHF. We check whether actually the class has referenced the method or not. The extended equation for MHF is given below:

$$\textbf{MHF extended} = \sum_{i=1}^{TC} \sum_{m=1}^{Md(Ci)} \textbf{(1-V(M}_{\textbf{ex}}\textbf{))} \,/\, \sum_{i=1}^{TC} M_d(C_i) \qquad (8)$$

where

$$V(M_{ex}) = \sum_{i=1}^{TC} is\_visible(M_{ex}, C_j) \,/\, (TC - 1)$$

and

$$\text{is\_visible}(M_{ex}, C_j) = \begin{cases} 1 \ \ iff \ \ j \neq i \ and \ C_j \ did \ reference \ M_{mi} \\ 0 \ otherwise \end{cases}$$

$M_d(C_i)$ is the count of methods and constructors. These methods can be of any access modifier. They should not be abstract or inherited.
TC - total count of classes in system/ package.

Thus, AHF extended and MHF extended propose a change in the visibility function of their respective calculations. This visibility function ensures that whether the members of a class have been actually referenced by outside members or not. This helps us in understanding the amount of abstraction in the system thereby giving clarity in estimation of actual hiding factors.

IV. RESULTS ANALYSIS

To demonstrate the variation in AIF and AIF extended values, MIF and MIF extended values, we have used a small example considering a design for university database.

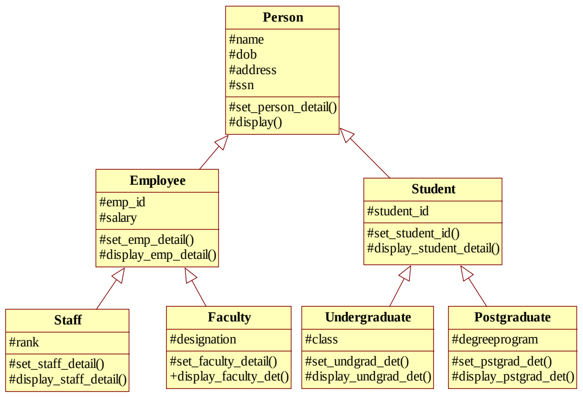


Figure 1. University Database

We have calculated the AIF values [6] and proposed extended AIF for each class as well as MIF values [6] and proposed extended MIF. The class diagram of the example system along with tabulated results and graph of AIF and AIF extended, MIF and MIF extended is given below.

A. *AIF Analysis*

A threshold value of 0.5 is maintained in order to determine whether level of inheritance is acceptable or not. For AIF values greater than 0.5, extent of inheritance is high Classes employee, student, undergraduate, postgraduate have "AIF extended" values greater than 0.5 and "AIF" values less than 0.5. Class diagram in Fig. 1 shows us effectively that these classes inherit large number of attributes from ancestor classes than the attributes they actually contain, thereby depicting unacceptable level of inheritance.

TABLE I. ANALYSIS OF AIF

Classes	AIF for each class (Farooq,2005)	AIF Extended for each class (Proposed)
Person	0.00	0.00
Employee	0.36	0.67
Staff	0.54	0.86
Faculty	0.54	0.86
Student	0.36	0.67
Undergraduate	0.45	0.83
Postgraduate	0.45	0.83
Main	0.00	0.00

Total AIF of the system can be calculated using (1):

$$\begin{aligned} AIF &= (0+4+6+6+4+5+5) / (11+11+11+11+11+11+11) \\ &= 0.39 \end{aligned}$$

Here, the numerator is the number of attributes inherited from ancestor classes for each class. As "person" is the base class, it does not inherit any attribute, "employee" class inherits four attributes, "staff" class inherits six attributes in total from person class and employee class, "faculty" class six, and so on for rest of the classes. Sequence of classes used in the formula is same as the sequence given in the table I. Denominator is number of attributes that can be referenced by each class. Attributes in the class diagram are protected in nature, but we know that protected members are public in their own package. Therefore, each class can reference all the public and protected members in the system. Denominator is eleven for each class.

We compute AIF extended using (5). The numerator remains the same as that of AIF but denominator changes as we consider the attributes of ancestor classes of class C_i and the attributes defined inside class C_i only. For example, "person" class is base class; we consider only the four attributes defined inside it. "Employee" class is inheriting from person class, so the attributes in consideration are six, out of which four are from person class and two from employee class. Similarly, denominators are determined for rest of the classes.

$$\begin{aligned} AIF \ extended &= (0+4+6+6+4+5+5) / (4+6+7+7+5+6+6) \\ &= 0.73 \end{aligned}$$

Therefore, AIF extended is giving a clear idea that level of inheritance in the system is not acceptable as it is greater than 0.5.

B. *MIF Analysis*

Same extension is followed in MIF extended but with respect to the methods. Classes staff, faculty, undergraduate, postgraduate have "MIF extended" values greater than 0.5 and "MIF" values less than 0.5.

TABLE II. ANALYSIS OF MIF

Classes	MIF for each class (Farooq,2005)	MIF Extended for each class (Proposed)
Person	0.00	0.00
Employee	0.14	0.5
Staff	0.29	0.67
Faculty	0.29	0.67
Student	0.14	0.5
Undergraduate	0.29	0.67
Postgraduate	0.29	0.67
Main	0.00	0.00

Total MIF of system is calculated using (2):

$$\text{MIF} = (0+2+4+4+2+4+4+0)/\ (14+14+14+14+14+14+14+14) = 0.18$$

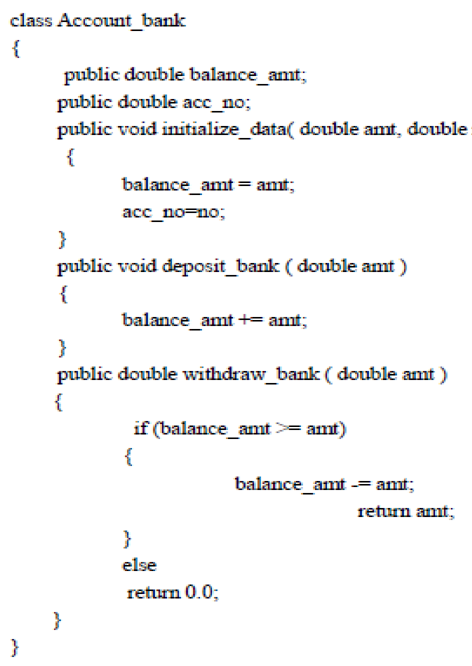
Sequence of the classes in the formula remains same as given in table II. Numerator is number of methods inherited by each class from ancestor classes. Denominator is number of methods that can be referenced by each class. As mentioned earlier that protected members are public in their own package, each class can reference all public and protected methods in system. Denominator is fourteen for each class. Now, we calculate MIF extended of system using (6).

$$\text{MIF extended} = (0+2+4+4+2+4+4+0)/\ (2+4+6+6+4+6+6+1) = 0.57$$

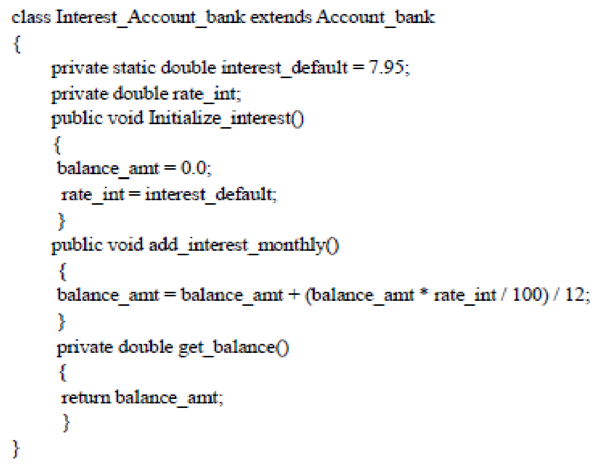
Numerator remains the same as that of MIF. Denominator changes according to the proposed work. We need to consider methods of ancestor classes of a class C_i and the methods defined inside class C_i only. "Person" is a base class and has two methods of its own. "Employee" class is inheriting two methods from person class and has two methods of its own, therefore a count of four. Likewise, we do the calculation. Therefore, MIF extended is giving a clear idea that level of method inheritance in system is not acceptable as it is greater than 0.5.

C. AHF Analysis

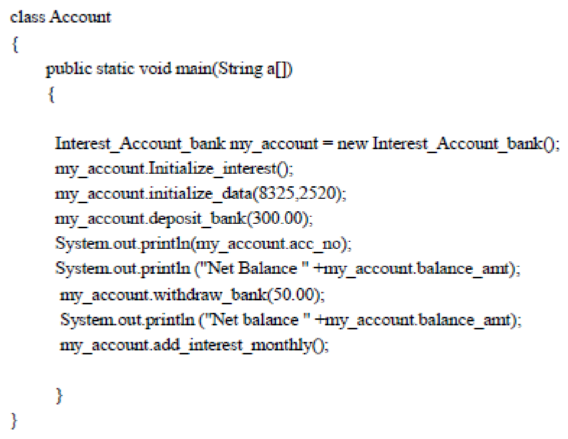
We have considered a code to demonstrate the hiding factor. The sample code is as follows:



```
class Account_bank
{
     public double balance_amt;
    public double acc_no;
    public void initialize_data( double amt, double no )
     {
          balance_amt = amt;
          acc_no=no;
    }
    public void deposit_bank ( double amt )
    {
          balance_amt += amt;
    }
    public double withdraw_bank ( double amt )
    {
           if (balance_amt >= amt)
          {
                         balance_amt -= amt;
                                    return amt;
          }
          else
           return 0.0;
     }
}
```



```
class Interest_Account_bank extends Account_bank
{
    private static double interest_default = 7.95;
    private double rate_int;
    public void Initialize_interest()
    {
     balance_amt = 0.0;
      rate_int = interest_default;
     }
    public void add_interest_monthly()
     {
     balance_amt = balance_amt + (balance_amt * rate_int / 100) / 12;
     }
     private double get_balance()
     {
      return balance_amt;
      }
}
```



```
class Account
{
     public static void main(String a[])
     {

      Interest_Account_bank my_account = new Interest_Account_bank();
      my_account.Initialize_interest();
      my_account.initialize_data(8325,2520);
      my_account.deposit_bank(300.00);
      System.out.println(my_account.acc_no);
      System.out.println ("Net Balance " +my_account.balance_amt);
       my_account.withdraw_bank(50.00);
       System.out.println ("Net balance " +my_account.balance_amt);
       my_account.add_interest_monthly();

      }
}
```

On the basis of above code, we check the current references made to attributes and methods. First we calculate AHF. Consider the attributes of Account_bank class, they are balance_amt, acc_no. Attribute balance_amt may be referenced by rest of the two classes, i.e. Interest_Account_bank and Account as it is a public variable. Therefore, using (3), visibility (bank_amt) is 2/(3-1), that is 1. Thus, value of (1-V(bank_amt)) is 0. Similarly, for the attribute acc_no, (1-V(acc_no)) is 0. Now, we consider the attributes of class Interest_Account_bank, they are interest_default, rate-int.

TABLE III. ANALYSIS OF AHF

Classes	Attributes	(1-V(A_{mi})) in AHF	(1-V(A_{ex})) in AHF ext.
Account_bank	balance_amt	0	0.5
Account_bank	acc_no	0	0.5
Interest_Account_bank	interest_default	1	1
Interest_Account_bank	rate-int	1	1

Both the attributes are private; therefore none of the classes can access them. Visibility is 0 for both the attributes, (1-V(A_{mi})) is 1. Now, we apply the formula for AHF from (3).

AHF = (0+0+1+1) / (4) = 0.5

Now, we calculate AHF extended. For the attributes balance_amt and acc_no of Account_bank class, they are actually referenced by the object of Interest_Account_bank. Thus, only one class has made an access to these attributes. Visibility for these attributes is 1 / (3-1), i.e. 0.5. Therefore,

(1-V(A_{ex})) is (1-0.5) i.e. 0.5. Similarly, for interest_default and rate-int attributes, none of the classes has accessed them, therefore visibility is 0 and (1-V(A_{ex})) is 1. Now, we apply the formula for AHF extended from (7):

AHF extended = (0.5+0.5+1+1) / (4) = 0.75

Higher value of AHF extended indicates that attributes are not actually referenced, thereby imparting a private attribute behavior to them. Visibility of attributes is not properly used by the design of the system.

D. MHF Analysis

First, we calculate MHF. Consider the methods of Account_bank class. This class has three public methods, namely initialize_data, deposit_bank and withdraw_bank. All three methods are public, and can be accessed by rest of the two classes. Therefore, using (4) visibility of all four methods is 2/(3-1), that is 1. Thus, value of (1-V(M_{mi})) is 0 for all three methods. Class Interest_Account_bank has three methods, namely initialize_interest, add_interest_monthly, and get_balance. Getbalance method is a private method that cannot be referenced by outside classes. Therefore, its visibility is 0 and (1-V(M_{mi})) is 1.

TABLE IV. ANALYSIS OF MHF

Classes	Methods	(1-V(M_{mi})) in MHF	(1-V(M_{ex})) in MHF ext.
Account_bank	Initialize_data	0	0.5
Account_bank	withdraw_bank	0	0.5
Account_bank	deposit_bank	0	0.5
Interest_Account_bank	Initialize_interest	0	1
Interest_Account_bank	add_interest_monthly	0	**1**
Interest_Account_bank	get_balance	1	1

Now, we apply the formula for MHF from (4):
MHF= (0+0+0+0+0+1) / (6) = 0.17

Now, we calculate MHF extended. For methods initialize_data, deposit_bank and withdraw_bank of Account_bank class have been actually referenced by the object of Interest_Account_bank class. By using (8), visibility of these methods is 1/(3-1) i.e. 0.5. Therefore, (1-V(M_{ex})) is (1-0.5) i.e. 0.5. Methods of Interest_Account_bank have not been referenced by any other class, therefore, their visibility is 0 and (1-V(M_{ex})) is 1. Now, we apply the formula for MHF extended from (8):

MHF Extended= (0.5+0.5+0.5+1+1+1) / (6) = 0.75

Such a high value of MHF extended indicates that most of methods are not being actually referenced by the outside classes.

E. Case Study Analysis

Library Management system for a college is used as a case study. It has separate java files for books, catalogue, members, librarian etc. Books may be reference book or issuable book. Members may be student or a faculty member. All the four metric i.e. AIF, MIF, AHF and MHF were applied on the case study. Also, the proposed extensions to these metrics were applied.

TABLE V. CASE STUDY RESULT

Metric	(Farooq,2005)	Extended Versions
AIF	0.25	0.52

MIF	0.18	0.59
AHF	0.52	0.98
MHF	0.94	0.11

There was variation in results, confirming the extensions were specific and gave a hint about design of the system. Metric values are capable to comment on stability of design and actual hiding factors. In all the cases, extended metrics resulted in values higher than the original metrics. Extended AIF and extended MIF gave values higher than threshold indicating the system has higher inheritance. Classes are highly coupled in system. Extended AHF and extended MHF also result in higher values than original metric showing greater hiding factor.

CONCLUSION AND FUTURE WORKS

The extensions in AIF and MIF are more accurate than previous definitions as they give a better idea about usage of inheritance property in the code. Results are accompanied with analysis part showing the variation in the values. Clearly, classes that have AIF, MIF values greater than threshold value needs some modification in their design. Extensions in AHF and MHF check whether a member (data or method) has been actually referenced or not. This gives clarity in estimation of actual hiding factors. Therefore, proposed extensions give accurate estimation of inheritance and hiding factor. Regarding future works, developed tool must have a provision for suggesting corrections to user, based on result of metrics. Developed tool analyses java source files and class files. Thus, tool can give results only after coding phase. An approach may be developed to apply metrics in earlier phases of development.

ACKNOWLEDGMENT

The authors highly acknowledge the immense support of Indian Institute of Information Technology, Allahabad, for providing the adequate resources to carry out this research work.

REFERENCES

[1] Jacobson I., Christerson M., Jonsson P., and Overgaard G. Object-Oriented Software Engineering, Pearson Education, Singapore, Ninth Indian Reprint, 2004.
[2] Pressman R. S., Software Engineering: A Practitioner's Approach, McGraw Hill Publication, Singapore, Sixth edition, 2005.
[3] Balagurusamy E., Programming with Java: A Primer, McGraw Hill Publication, New Delhi, Thirtieth reprint, 2006.
[4] Shyam R. C. and Chris F. K. “A Metrics Suite for Object Oriented Design “IEEE Transactions on Software Engineering, Vol. 20, No. 6, June 1994.
[5] Subramanyam R. and Krishnan M.S. “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects” IEEE Transactions on Software Engineering, Vol. 29, No. 4, April 2003.
[6] Farooq A., Braungarten R., and Dumke R.R. “An Empirical Analysis of Object-Oriented Metrics for Java Technologies” 9th International Multitopic Conference, pp. 1-6, IEEE INMIC 2005.