Static Analysis To Model & Measure OO Paradigms

Satwinder Singh
Lecturer
Dept. Computer Science Engineering, Baba Banda Singh Bahadur Engg. College, Fatehgarh Sahib-140407
Email: satwindercse@gmail.com

K.S. Kahlon
Professor
Dept. Computer Science Engineering, Guru Nanak dev University Amritsar-143001
Email: karanvkahlon@yahoo.com

ABSTRACT

Object oriented development has proved its worth in today's system because its design and development is better, reliable and easier to access than the traditional methodologies. Due to updated requirements and lack of documentation in old systems has provided a motivation to revamp the systems. Rebuilding or redesigning the same system is highly expensive. To overcome this problem reverse engineering of the system is used as most suitable alternative. Field of reverse engineering is expanding its horizon day by day; it requires reusability not only at code level but also at higher level which can measure the analysis results and original system. Reverse engineering, strategy has been developed to analyse and modeling the OO files by designing the translator. It models and measures the OO by using traditional metrics and new encapsulation metrics (Public Factor (PuF) & Private Factor (PrF)) essential for developing the good software. In this work we tried to refine metrics especially for object-oriented programming and set of these metrics has been defined.

Keywords: Static analysis, Public Factor, Private Factor, AHF, MHF.

1. INTRODUCTION

Donald Firesmith in his book Dictionary of Object Technology (SIGS Books, 1995), defined analysis as "the development activity consisting of the discovery, modeling, specification and evaluation of requirements," while OO analysis is "the discovery, analysis and specification of requirements in terms of objects with identity that encapsulate properties and operations, message passing, classes, inheritance, polymorphism and dynamic binding". Generally, OO methodologists seem to agree that OOA objects are the objects in a problem space. More specifically, an OOA object has been defined as an abstraction of something in the problem domain

Analysis of a system can be done statically and dynamically. Both the static and dynamic analyses are important to understand the system. Static analysis is the process of analyzing software without executing it where as dynamic analysis is the method of analyzing the runtime behaviour of a software system.

Modeling is not a simple process and one should know beforehand about the difference between the programming and modeling. Initial model of the program describes the essential properties which are required to analyze the correctness of the program. This correctness cannot be analyzed only from the text or specifications based on modeling but it requires some specific metrics.

This paper describes static analysis model of OO program file for beginner programmers. It is designed for both tutorial and assessment purposes. The key features of the analysis are its configurability and extensibility. Analyses can be configured to suit different types of exercises. In addition, the complexity of analysis is measured by different Object Oriented metrics. Among them some are predefined and two new metrics are defined to measure encapsulation.

This paper is divided in following sections; Section II gives the review of literature. Section III gives the Translator Design which does the static analysis of the OO file. Section IV discusses the OO System Metrics. In next section results of analysis and measurements are discussed and paper ends with conclusion and future work.

SAC'10, March 22-26, 2010, Sierre, Switzerland.

2 REVIEW OF LITERATURE

According to Nghi Truong *et al*[2] the static analysis framework consists of two analyses: Software engineering metrics and structural similarity. The first evaluates the quality and the second examines the similarity in structure of student programs compared with model solution. The analyses were performed on XML marked-up AST representations of programs. Feedback to students includes comments about the quality and structure of their programs. Overall, the framework had four limitations. First, the chosen technique only works with small or "fill in the gap" type programming exercises to minimize the implementation variation in structural similarity analysis. Second, the framework was able to analyse only well formed gaps. Third, the framework did not implement semantic analysis; however, with its extensible architecture, additional analyses could be plugged in easily. Last, the framework only analyses syntactically correct programs. All gaps need to be completed in order to carry out the analysis with multiple dependent gaps exercises.

Martin [17] presents the case that simply using objects to model an application was insufficient to gain robust, maintainable and reusable designs. That there was other attributes of a design that were required and these were based upon a pattern of interdependencies between the subsystems of the design that support communications within the design, isolate reusable elements from non-reusable elements, and block the propagation of change due to maintenance. Moreover, this paper presents a set of metrics that could be easily applied to a design, and that measures the conformance of that design to the desired pattern of dependencies. Metrics provide information to the designers regarding the ability of their design to survive change, or to be reused.

Saini et al [23] has developed the Encapsulation Factor metrics to measure only the Privacy and unity but did not touch the protected

and public behavior of the class. So to overcome this problem we tried to describe a new set of metrics to measure the public and protected behavior of members of a class.

Magnus Andersson *et al* [8] reported software metrics those can be used to determine the object-oriented design quality of a software system. An experimental study was conducted as an attempt to validate each metric and to understand them. They formulated the strategies on how analysis of source code with metrics could be integrated in an ongoing software development. Moreover they have explained how metrics could be used as a practical aid in code and architecture investigations on already developed systems. Metrics do have a practical application and up to some extent represents software systems design quality, such as: complexity of methods/classes, package structure design and the level of abstraction in a system. Andersson *et al* did not measure vital design issues such as polymorphism and encapsulation. Some of the metrics represent few aspects of the design quality of the system.

Software metrics is the only mode to measure the quality of the program. Mengel and Yerramilli, [4] Leach and Mengel [3] have analysed Halstead metrics (Halstead, 1977), McCabe cyclomatic complexity [5] and number of coupling instances. Berry and Meekings [6] style guide line were common and useful static metrics for measuring the programs of computer science applications. However, they were often used for marking and detection purposes rather than for teaching design and writing good quality programs. Good software design concept is so subjective that empirical studies are necessary to clarify which measure and describe good design [18-22].

3. TRANSLATOR DESIGN

3.1 Static Analysis

Static analysis is the process of examining source code without executing the program. This leads to the need of methodologies that support reusability not only at the code level but also at higher (semantic) levels, in order to minimize the effort of proving correctness of the analyses. Aims to adopt static analysis of the system are [13]:

1. The conceptual simplicity and soundness of this technique
2. The design of abstract interpreters which has already been successfully developed for other language paradigms

The Static analysis can briefly be described by the following three steps [1, 7].

1. Define a concrete (operational) semantics, i.e., a formal representation of concrete execution states and of the transition rules corresponding to statement executions. This step, of course, is language-dependent.
2. Define an abstract semantics, i.e. a non-standard domain whose elements represent sets of concrete execution states, and a suite of abstract operations that safely approximate the corresponding concrete ones.
3. Define a generic algorithm, parameterized on the abstract domain that computes a (post-) fix point of the abstract semantics, thus yielding safe information about concrete program executions.

3.2 Logical Design

Logical design gives the conceptual view of the solution. Source file (*addnero.cpp*) used for parsing was coded in C++. Following four components of parsed file are modeled:

1. Packages
2. Variables
3. Methods
4. Classes

1. Packages: Model includes the list of in build and user build packages or header files used in source file.

2 Variables: Variables declared in the source file listed in our model with data type of each variable and are represented with their respective notations in Table1.

3 Methods: Translator list the user defined functions which were declared in source file. It also lists the return type of methods in notated form. Notations used for each return type key word was same as in declaring variables. Variables declared in each function definition was listed and its data type was notated as per Table1.

TABLE 1: Notation for Variables and Functions

S. No.	Data Type	Notation
1.	Int	$
2	Int*	$*
3	Long	$
4	Short	$
5	Float	@
6	Float*	@*
7	Float**	@*
8	Double	@
9	Char	&
10	Char*	&*
11	Char**	&*
12	Void	0
13	Bool	!

4 Class: Model includes items encapsulated in class e.g. data member, member functions and inherited classes. It also includes the scope (Notations as per Table 2.) of each data member and member function. Data type and return type of data member or member functions respectively are notated as per Table 1. which are followed by their names.

TABLE 2: Scope Notations of Variables and Functions

S. No.	Scope	Notation
1.	Public	-
2	Private	+
3	Protected	#

4. OBJECT ORIENTED METRICS ANALYSIS

Software Metrics have become essential in software engineering for several reasons; two of them are (i) for quality assessment (ii) for reengineering. In forward engineering they are being used to measure software quality and to estimate cost and effort of software projects [16]. In the field of software evolution, metrics can be used for identifying stable or unstable parts of software systems and for identifying possibilities of application of re-factorings [14]. Moreover they measure the quality related to the structure of evolving software systems. In the area of software reengineering and reverse engineering [15], metrics are being used for assessing the quality and complexity of software systems. Metrics improve basic understanding and providing clues about sensitive parts of software systems.

Software metrics is a well-known quantitative approach used to measure software quality. This analysis is based on software complexity metrics and good programming practice guide lines, to assess the quality in the form of complexity of program. Apart of some new metrics which are proposed in this paper some predesigned metrics are also used to evaluate and measure the program at class level. It includes the following:

1. Information Hiding Factor
2. Inheritance Factor
3. Encapsulation Factor
4. Weighted Class Complexity Factor
5. No. of Ancestor Count

Each of these metrics represents the basic paradigm of OO system such as encapsulation (Information Hiding Factor, Public Factor, Private Factor), Inheritance (Class Inheritance Factor), Message passing (No. of Ancestor Count) and Classes (Weighted Class Complexity Factor). In all the above metrics (except NAC) numerator in the formula represents the actual value where as denominator represent the maximum value it can hold. So that's why these are named as a "Factor" and its values are ranged between 0 & 1.

4.1 Information Hiding Factor

Information hiding factor is the measurement of hidden factor of encapsulation at attribute level and method level. The set of two OO MOOD metrics [11] to measure this are as follow:

4.1.1 Attribute Hiding Factor (AHF): The Attribute Hiding Factor measures the hidden data members in classes. The hidden factor of a class is the percentage of the total classes from which the data members are not visible. Attributes should be 'hidden' within a class if they are kept from being not to be accessed by other objects (by being declared as a private).The Attribute Hiding Factor is a fractional measurement. The numerator is the sum of the private data members defined in all classes. The denominator is the total number of data members defined in the program. Large value of AHF shows good programming practice and design.

4.1.2 Method Hiding Factor: The Method Hiding Factor measures the hidden methods in classes. The Hiding factor of a method is the percentage of the total classes from which the methods are hidden.

The Method Hiding Factor is a fraction measurement where the numerator is the sum of the private methods defined in all classes and denominator is the total number of methods defined in the program.

Methods should be encapsulated (hidden) within a class and not available for use to other objects. Method hiding increases reusability in other applications and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden. Thus hiding methods also reduces modifications to the code [16]. Large value of MHF shows good programming practice and design.

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{n} M_d(C_i)}$$

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{n} A_d(C_i)}$$

4.2 Inheritance Factor:

Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The need for this metrics factor is as follow

Suppose two methods with same names are defined in two different classes. When a call to child class function has been made then it will clearly inherit the behaviour defined in super class. In this case coding have not to be rewritten again, it is re-used. This will affects both the 'Method Inheritance Factor (MIF) & Polymorphism Factor (POF) metrics. A system consist of the two classes defined above would return a value of 0 for MIF, which shows that there is no inheritance, which is misleading which is an anomaly.

Definitions for MIF[11] and AIF[11] are inconsistent with the 0-1 scale. The methods available in a child class, *Ma*(*Child Class*) are not inheritable (within the system). The value of denominator in the definition of MIF does not give the max value of inheritance but it represents value of any system greater than 1 logically it can't. It is required to fix the inconsistency in MIF and AIF to measure the inheritance factor. So for this purpose some class level metrics should be developed as because inheritance is the class level OO paradigm. So to remove these anomalies following definition of Class Inheritance Factor (CIF) has been proposed [12] where numerator includes the total number of ancestor count (AC) of class C_i and denominator includes the maximum possible inheritance in the program.

$$CIF = \frac{\sum_{i=1}^{TC} AC(C_i)}{TC * (TC-1)/2}$$ {TC: Total Classes}

4.3 Encapsulation Metrics

Encapsulation means that "all that is seen of an object is its interface, namely the operations that can perform on the object [9]." As information hiding and encapsulation are different concepts because encapsulation is consist of two concepts i.e. integrity and visibility. So separate metrics is required which can measure both the above concept. As MHF & AHF are considered to be a measure of encapsulation but it only measures the visibility of class member functions & data members separately. As encapsulation is the measure of a unity and integrity of attributes and methods visibility. In this work we are measuring both parameters (unity and integrity) together. So to remove this anomaly following two metrics has been proposed (i) Public Factor and (ii) Private Factor.

4.3.1 Public Factor (PuF): This proposed metrics count the measure of encapsulation as it is the measure which counts the composite scope of methods and attributes. In this proper metric effort has been made to count the public factor. The range of this factor is from 0 to 1. PuF=0 when there are no methods and attributes of Public Scope. PuF= 1 when there no methods and attributes are defined under Private Scope. Visibility of any member under protected scope is counted as :

V_i= DC (C_i)/(TC-1)

(note: DC(C_i) = descendants of *Curent Class* C_i *and TC= Total Clasess)* . Where V_i=1 for Public Members and V_i=0 for Private Members

$$PuF = \frac{\sum_{i=1}^{TC}(Pu(A_i)*V_i) + \sum_{i=1}^{TC}(Pu(M_i)*V_i)}{\sum_{i=1}^{TC}(A_{ti}+M_{ti})}$$

$Pu(A_i)$= Public Attributes in Class i
$Pu(M_i)$= Public Methods in Class i
V_i= Visibility Factor
A_{ti}=No. of total Attribute declared in class i

4.3.2. Private Factor (PrF): This metrics has been proposed to count the measure of encapsulation as it is the measure which counts the composite visibility and integrity scope of methods and attributes. In this proper metric effort has been made to count the

private factor. The range of this factor is from 0 to 1. PrF=0 when there are no methods and attributes of Private Scope. PrF= 1 when there no methods and attributes are defined under Public Scope.

$$\Pr F = \frac{\sum_{i=1}^{TC} (\Pr(A_i) * (1 - V_i)) + \sum_{i=1}^{TC} (\Pr(M_i) * (1 - V_i))}{\sum_{i=1}^{TC} (A_{ti} + M_{ti})}$$

$Pr(A_i)$= Private Attributes in Class i
$Pr(M_i)$= Private Methods in Class i
V_i= Visibility Factor
A_{ti}=No. of total Attribute declared in class i
M_{ti}=No. of total Methods declared in class i

4.4Weighted Class Complexity Factor (WCCF)

WCCF measures the complexity of an individual class. A class with more member functions than its peers is considered to be more complex and therefore more error prone [21]. More is the number of methods in a class; the greater is impact on children since children inherit all the visible methods defined in a class. Classes with large numbers of methods are limiting the possibility of reuse of data members and functions. So this reasoning shows that class should have the less number of data members for reusability. But recent study differ from the above study, in the way like, smaller methods over fewer, larger methods to reduce complexity, it increases the readability, and improving the understanding of system [20]. So, the result come out to be is, a large inheritance tree but this is not advisable. Often, the WCCF calculation considers complexity and the count of the number of methods applies a weighted complexity factor [17]. But classes are not only composed of methods it also includes the attributes. So we can't ignore the attributes while calculating the weighted complexity factor. A new definition of WCCF is as follow:

$$WCCF = \frac{(M_n) + (A_n)}{\sum_{i=1}^{TC} (A_i + M_i)} \quad \{0 \le n \le TC\}$$

Numerator includes the sum of methods and attributes of a particular class and denominator includes the summation of all the attributes and methods declared in the classes.

4.5 Ancestor Count

This metric measurement gives the potential effect on the class by the ancestor classes. Inheritance is used to spread the implementation of the entity on different classes. By this way Inheritance diversifies the complexity of a class to different classes. Ancestor count measure gives us the depth of inheritance by counting number of steps from the class node. More number of ancestor count make use of more methods and classes and so, the greater the complexity is, more is the count, more is the reusability, since inheritance is an example of reuse. More count may also be a case of misuse of subclassing. More classes need more testing of methods in a class and give the potential influence of other classes in present class.

5. RESULT AND DISCUSSION

This model will help to understand the problem well by knowing the complexity of the problem by measuring the various OO paradigms. The above model will also help to analyse the structure of OO language files. This model will not be distracted

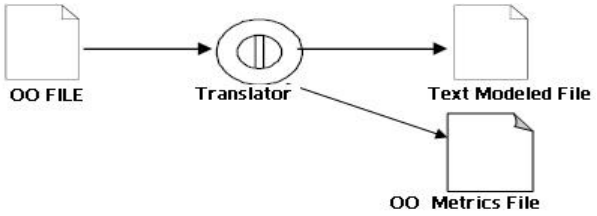

Fig. 1. Translator Working Model

the user but it made a task simple. Model gives a detail picture of a file as how many variables are included in global part, in a particular function or in a class. It also includes the information of relationship between different classes by measuring the encapsulation metrics. Previously most of the work has been done to simply analyse the file it does not include the relationships of a particular variables. Translator gives a model of the files which revealed the complexity and analyse it. Block model of a class as shown in Figure 2, encapsulates variables and functions is an easy way to design efficient program. This model help in better understanding of source program by presenting the code file in simple text mode and analyse it with the help of metrics. The programmer's effort of manual checking and analysation of complexity has been reduced(by metrication) with this model. Metrics has been designed to measure the system, because of expectation of different quality product as well as productivity gain. Translator produced model has implemented the various previously designed metrics and an effort has been made to design a new set of metrics(PuF & PrF) for measuring encapsulation of class. Metrics or measurements are the key foster for reuse or re-engineering analysis model. These metrics examine the analysis model with intent of predicting the complexity of resultant system. Complexity here is an indicator of design complexity. Fig. 1, 2& 3 show the result of above discussion. Fig. 1 shows that OO file (addnero.cpp, this is a student program to implement the addition of a neros) is parsed into translator and two (Text modeled file & Metrics File) new files are generated. Figure 2 shows text model result of C++ file, firstly it shows the list of header files then some global functions declarations. After that various attributes and methods in a *'andnero'* class are listed under private and public scope as per notations assigned above. It also shows the ancestor classes of '*andnero'* class which is blank. After the class definition, each function definition is given which includes the variables declared in it with their notated data types. Whereas Fig 3 shows the result of metrics measured of the parsed file. So, can the above metrics be used, will measure the design quality of OO? In respect to this question, results shows that it reflect some aspect of design quality by giving OO static analysis model, methods/class level and encapsulation measurement.

C++ Sample Program

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void reversevideo(int,int, char);
void normalvideo(int, int, char);
class andnero
{int k;
public: int ap[20],n,m[20];
int i,j,k,li; int arr[20][20];
void box(int x1, int y1, int x2, int y2);
void addn();
void read();
private: int f;};
void andnero::box(int x1, int y1, int x2, int
y2)
{for (int col = x1; col < x2; col++)
        {gotoxy(col, y1);  cprintf("%c", 196);
         gotoxy(col, y2); cprintf("%c", 196);}
for (int row = y1; row < y2; row++)
{gotoxy(x1,row); cprintf("%c", 179);
gotoxy(x2,row); cprintf("%c", 179);}
gotoxy(x1, y1); cprintf("%c", 218);
gotoxy(x1, y2); cprintf("%c", 192);
gotoxy(x2, y1); cprintf("%c", 191);
gotoxy(x2, y2); cprintf("%c", 217);}
void andnero::read()
{ cout<<"enter the no of layer"; cin>> n;
for (i=0;i<n;i++)
{ cout<<" enter t no.of cell in"<<i+1<<"th
layer"; cin>>m[i]; }
for (i=0;i<m[0];i++)
{cout<<" enter the input to"<<i<<"th cell";
cin>>ap[i];}
for (i=0;i<n;i++)
  {   for(j=0;j<m[i];j++)
    { cout<<"input the weight";cin>>arr[i][j];}
}}
void andnero::addn()
{int s=0; read();
int mul[20][20],acin[20][20];   i=1;
  for(j=0;j<m[i];j++)
{ mul[i][j]=0;          while(s<m[0])// && k <
m[i-1])
{ mul[i][j]=mul[i][j] + ap[s] * arr[i-1][s];//[k];
     s=s+1; }      }
cout<<"\n\n result is = "<< mul[i][j];
                s=0;//k=0;
cout<<"\nenter the limit";    cin>>li;
    {  if (mul[i][j]>=li)
  acin[i][j] = 1;    else    acin[i][j] = 0;
  cout<<"result="<<a; cin[i][j];  }              }
   int res[20][20];
   for (i=2;i<n+1;i++)
     {  for (j=0;j<m[i];j++)
     {res[i][j]=0;       while(s       <       m[i-1])
{res[i][j]=res[i][j]+acin[i-1][s]*arr[i-1][s];
s=s+1;}
cout<<"\n\n result is="<<res[i][j]; s=0;
cout<<"\n enter the limit";          cin>>li;
 if   (res[i][j]>li)              acin[i][j]=1;   else
            acin[i][j]=0;
cout<<"\n net result is="<<acin[i][j];}    }    }
typedef char option[15]; char menu();
 void main()
{           char choice; andnero an;
            do          {choice = menu();
                        switch (choice)
            {case '1':  an.addn(); break;
            case '2':   an.addn(); break;
            case '3':   an.addn(); break;
            case '4':   an.addn(); break;
            case '5':   an.addn(); break;
            case '6':   an.addn(); break;
            case '7':   an.addn(); break;
default :   exit(0); }} while (choice != 0);}
void normalvideo(int x, int y, char *str)
{  gotoxy(x, y);    cprintf("%s", str);}
void reversevideo(int x, int y, char *str)
{textcolor(RED); textbackground(WHITE);
 gotoxy(x, y);              cprintf("%s", str);
 textcolor(GREEN);
textbackground(BLACK);}
char menu()
{           int i, done;
            andnero an;
option a[]={" Bubble-Sort", "  Heap-sort ",
            "Selection-Sort", "Insertion-Sort",
            "  Quick-sort", "  Merge-sort",
            "  Shell_sort", "Quit  "};
an.box(20, 6, 65, 20); an.box(18, 4, 67, 22);
textcolor(5+143); gotoxy(30, 5);
textbackground(WHITE);
cprintf("S O R T I N G  -  M E N U");
textbackground(BLACK); textcolor(22);
for (i = 1; i < 8; i++)
            normalvideo(32, i+8, a[i]);
            reversevideo(32, 8, a[0]);
            reversevideo(32, 8, a[0]);
i = done = 0;
_setcursortype(_NOCURSOR);
            do{         int key = getch();
            switch (key)
{case 00:key = getch();
case 72: normalvideo(32, i+8, a[i]);
              i--;
              if (i == -1)              i = 7;
            reversevideo(32,      i+8,      a[i]);
            break;
case 80: normalvideo(32, i+8, a[i]);
            i++;
            if (i == 8)  i = 0;
            reversevideo(32, i+8, a[i]);break;
case 13:    done = 1;  }
            }           while           (!done);
_setcursortype(_NOCURSOR);
return(i+49);}
```

Static Analysis Result

```
Packages in use are:
        1       iostream.h
        2       conio.h
        3       stdlib.h
0       reversevideo(int,int, char);
0       normalvideo(int, int, char);
Class Started  andnero
Inherited Classes class andnero:
Declaration of variables and methods
andnero
Scope   DataType        Variables/Methods
-
        $       k
+
        $       ap[20],n,m[20]
        $       i,j,k,li
        $       arr[20][20]
        0       box(int x1, int y1,
                        int x2, int y2)
        0       addn()
        0       read()
-
        $       f
 End of Class


FUNCTION DEFINATION OF box(int x1,
 int y1, int x2, int y2)

FUNCTION DEFINATION OF        read()
FUNCTION DEFINATION OF        addn()

        $       s=0;
        $       mul[20][20],acin[20][20];
        $       res[20][20];
        &       menu();
FUNCTION DEFINATION OF        main()

        &       choice;

FUNCTION DEFINATION OF
normalvideo(int x, int y, char *str)

FUNCTION DEFINATION OF reversevideo(int
x,
int y, char *str)
FUNCTION DEFINATION OF
        menu()

        $       i, done;
        $       key = getch();
```

Figure 2. C++ Example

Metrics File

No. Of Packages3
No. Of Functions7
No. Of class Functions3
No. Of Classes1
No. Of Variables in Classes10
No. Of Public Variables in Classes8
No. Of Private Variables in Classes2
No. Of Public Functions in Classes3
No. Of Private Functions in Classes0
No. Of Variables7
No. Of Global function4

List of classes
andnero

No.of PUBLIC Variables in 0 Class are: 8
No.of PRIVATE Variables in 0 Class are: 2
No.of PUBLIC Functions in 0 Class are: 3
No.of PRIVATE Functions in 0 Class are: 0

No. of Ancestor in 0 Class are: 0
Private factor: 0.153846
Public factor: 0.846154
Attribute Hiding factor: 0.2
Method Hiding factor: 0
Class Inheritence Factor: 0
Weighted Class Complexity Factor: 1

Figure3 Metrics File

6 CONCLUSION

Model proposed in this paper is performing the static analysis of CPP file. While analysing it analyses two things first the structure of the program file(modeling) and then it measure the quality of the program with help of some new and traditional metrics.

Firstly, structure of the program file was analysed by translator by abstracting the packages used, classes and their relationship with other classes, functions and their definition, and variables encapsulated in functions, in classes and declared globally. While analysing a file with translator, rules were devised to notate the analysis of files. In this efforts were made to model and structured an OO file so that it should be easy to understand.

Secondly, quality of programs was analysed with the help of metrics. In this paper effort has been made to measure the OO paradigms by set of metrics which include some pre-defined and new metrics. OO paradigms like inheritance was measured by CIF and NAC, Information Hiding by MHF and AHF, Class weight in program was measured by WCCF where as Encapsulation and Abstraction was measured by new defined metrics i.e. PuF and PrF. From this discussion we want to conclude that MHF & AHF only measures the invisibility of the system but not the integrity or unity. Which is also to be measured along with the invisibility as per the definition of encapsulation described above.

This analyses model helps in the way that programmer need not to start the work from scrap. Model gives a block model of the files from which devised concept and quality was revealed. If concept is known then it is very easy to redesign the program and with quality factors (metrics) better quality software can be designed. But concept or quality revealed is not all about the OO. Object Oriented is much more than that. Another aspect which is not measured is design pattern which can help to prepare a better quality design. Another problem is it does not tell where the problem is and how to redesign it. Question can also be raise that is there any place for the software metrics in future? It has future but as long as standard metrics are not devised and threshold values are not defined. The above translator can be implemented to other type of OO languages also, to make it universal analysis tool. Translator has some limitations too, it does not support multiple source files which should make it more active and it understands well formed gaps only.

7. REFERENCES

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.

[2] Nghi Truong**,** Paul Roe, Peter Bancroft Static Analysis of Students' Java Programs. In Proceeding of *sixth Australian Computing Education Conference (ACE2004)*, Dunedin, New Zealand. 2004

[3] Leach, R. J. 1995. Using metrics to evaluate student programs. *SIGCSE Bull.* 27, 2 (Jun. 1995), 41-43. DOI= http://doi.acm.org/10.1145/201998.202010

[4] Mengel, S. and Yerramilli, V., A Case Study Of The Static Analysis Of the Quality Of Novice Student Programs. *Proc. Thirtieth SIGCSE technical symposium on Computer science education*, New Orleans, Louisiana, United States,**13:**78-82, 1999

[5] McCabe, T. J., A Complexity Measure. *IEEE Transactions on Software Engineering*, **2**(4): 308-320, 1976. http://ieeexplore.ieee.org/servlet/opac?punumber=32

[6] Harrison, W. and Cook, C. R. 1986. A note on the Berry-Meekings style metric. *Commun. ACM* 29, 2 (Feb. 1986), 123-125. DOI= http://doi.acm.org/10.1145/5657.5660

[7] F. *Nielson, H.* R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[8] Magnus Andersson and Patrik Vestergren. Object-Oriented Design Quality Metrics.Master Thesis, Uppasla University, Uppsala, Sweden. 2004

[9] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

[10] A. Milanova, A. Rountev, and Ryder B. G. Parameterized object sensitivity for points-to and side-effect analyses for java. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002). ACM Press, 2002.

[11] Abreu, F. B. e., "The MOOD Metrics Set," presented at ECOOP '95 Workshop on Metrics, 1995.

[12] Mayer, T. and Hall, T. 1999. Measuring OO Systems: A Critical Analysis of the MOOD Metrics. In *Proceedings of the Technology of Object-Oriented Languages and Systems* (June 07 - 10, 1999). TOOLS. IEEE Computer Society, Washington, DC, 108.

[13] Pollet, I., Charlier, B. L., and Cortesi, A. 2001. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (June 18 - 22, 2001). J. L. Knudsen, Ed. Lecture Notes In Computer Science, vol. 2072. Springer-Verlag, London, 77-98.

[14] Serge Demeyer, St_ephane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In Proceedings of OOPSLA'2000, ACM SIGPLAN Notices, pages 166-178, 2000.

[15] Chikofsky, E. J. and Cross II, J. H. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw.* 7, 1 (Jan. 1990), 13-17. DOI= http://dx.doi.org/10.1109/52.43044

[16]Norman Fenton and Shari Lawrence Peeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, second edition, 1996.

[17]Martin Robert, "OO Design Quality Metrics An Analysis of Dependencies", http://www.objectmentor.com/resources/articles/oodmetrc.pdf, 1994

[18] Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen, *An Empirical Study on Object Oriented Metrics*, State University of New York, Albany, 1999.

[19] Lionel C. Briand, John Daly, Victor Porter, Jurgen Wust, *A Comprehensive Empirical Validation of Design Measures for Object Oriented Systems*, Fifth international Software Metrics Symposium, 20-21 Nov, 1998.

[20] Lionel C. Briand, Jurgen Wust, John W. Daly, Victor Porter, *Exploring the Relationships between Design Measures and Software Quality in Object Oriented Systems*, http://www.sce.carleton.ca/faculty/briand/pubs/jss.pdf, 2004-06-04.

[21] Michelle Cartwright, Martin Shepperd, *An Empirical Investigation of an Object Oriented Software System*, IEEE Transactions on Software Engineering, Vol. 26, Issue 8, pp. 786-796, 2000.

[22] Rachel Harrison, Steve J. Counsell, *An Evaluation of the MOOD Set of Object Oriented Software Metrics*, Vol. 24, Issue 6, pp. 491-496, 1998

[23] Saini, S. and Aggarwal, M. 2007. Enhancing mood metrics using encapsulation. In *Proceedings of the 8th Conference on 8th WSEAS international Conference on Automation and information - Volume 8* (Vancouver, British Columbia, Canada, June 19 - 21, 2007). A. Aggarwal, Ed. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, 252-257.

[24] Khan, R. A. and Mustafa, K. 2009. Metric based testability model for object oriented design (MTMOOD). *SIGSOFT Softw. Eng. Notes* 34, 2 (Feb. 2009), 1-6. DOI= http://doi.acm.org/10.1145/1507195.1507204