Enhancing Mood Metrics Using Encapsulation

SUNINT SAINI, MEHAK AGGARWAL
Department of CSE & IT
Guru Nanak Dev Engineering College
Gill Road, Ludhiana, Punjab
INDIA

Abstract: The requirement to improve software productivity and software quality has promoted the research on software metrics. Encapsulation is a powerful mechanism in Object-Oriented programming and it is critical for building large complex software; which can be maintained and extended. Since the emergence of the Object-Oriented approach to software development, it has been recognized that the development of Object-Oriented metrics to measure the quality of software is a new challenge. Many researchers have already risen to this challenge, most notably Chidamber & Kemerer, Abreu, Lorez and Kidd. But the metrics introduced by the above proponents lack in measuring the encapsulation mechanism. This paper purposes a new metric to measure encapsulation. Encapsulation constitutes both privacy and unity and these two attributes have been taken to purpose the above mentioned metrics. A statistical analysis on EF (Encapsulation Factor) metric is also done in this study.

Keywords: Metrics, Object-Oriented, classes, Data Visibility, Cohesion, Encapsulation

1. Introduction

Many criticisms of Object-Oriented paradigm, when closely analyzed, turn out to be a criticism of the language being used-in most cases the language like Java, C++ or Smalltalk etc. The problems with such a language stem from the programmer failing to use a pure Object-Oriented model, and falling into old, function-oriented habits. So to ensure that pure Object-Oriented design is being created is by Object-Oriented design metrics. A small set of well-defined metrics, which can measure the source code automatically and produces summary results that are easily understood and interpreted, will allow the programmer to create a clear overview of the system he is developing. The main OO mechanisms of abstraction, encapsulation, inheritance, coupling and polymorphism are well understood by programmers and designers, and it is recognized that making "good use" of these mechanisms is key to producing elegant, maintainable and reusable software system. Many Object-Oriented metrics have been proposed by Chidamber and Kemerer, MOOD metrics and Lorenz and Kidd [2,3,4,5] metrics that don't measure all the mechanisms. The challenge then is to develop Object-Oriented metrics to measure all the mechanisms. In this paper a new metric to measure the Encapsulation is proposed.

2. Encapsulation and Its Importance

Encapsulation is defined as the ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings.

Encapsulation is concerned with the packaging of data and behaviour to represent a single entity. To properly access the quality of encapsulation human understanding is required and the design also needs to be fully comprehended. There are however two aspects of the encapsulation that can be assessed automatically. The first concerns the degree to which a single class represents a single entity called Class Unity and second relates to the visibility of a class data called data visibility. This is explained in section 3.

Encapsulation is critical to building large complex software, which can be maintained and extended. Many studies have shown that the greatest cost in software is not the initial development, but the thousands of hours spent in maintaining the software. Well-encapsulated components are far easier to maintain. Once software is in place, another great expense is extending its functionality. As new features are added, there is risk that it'll break existing parts

of the application. Encapsulation helps to minimize this risk. In a well-designed program, each object should have a single area of responsibility. That object presents an interface, which defines the services the object provides.

3. Purposed Metric For Encapsulation

A class is composed of attributes and methods. In this proposal we measure the privacy in terms of the attributes (Number of private attributes in the class) and unity in terms of methods and attributes (cohesion among the attribute and methods in a class).

3.1 Measuring Class Unity (CU)

Class Unity is a measure of the similarity of methods in a class. Two methods are considered similar if they access one or more of the same instance fields. CU counts distinct cluster of methods, where a cluster is defined as a group of methods that are linked to each other, either directly or indirectly, through accessing the same field or set of fields, any one method in the cluster accessing at least one field which is accessed by at least on of the other fields in the cluster. A cluster can be conceived as a graph where the nodes are the methods. Two nodes are connected by an edge if the two methods both access the same instance variables. A measure of one indicates full unity (all methods connected).

Chidamber and Kemerer [2] proposed a measure called LCOM (Lack of Cohesion in methods) to measure the cohesion in object-oriented methods. It is based on the count of the number of paired methods that use the same instance variables directly. Here to measure cohesion as in [1] we also consider the pair of methods, which use the common attributes but the manner in which an attribute may be used is differently i.e. both the attributes used either directly or indirectly by a method are considered. An instance variable is *directly used* by a method *M* if instance variable appears as a data token in the method *M*. An instance variable is *indirectly used* by a method M if 1) the instance variable is directly used by another method M_1 that is called directly or indirectly by M, and 2) the instance variable directly used by M_1 is in same object as M.

3.1.1 Graphical Model of the class [1]

To measure the class cohesion, two basic components of the class are required i.e. methods and instance variables.

In the graphical representation of the class, methods are represented by rectangle and ovals are used to represent the instance variables. A link between a rectangle and an oval indicates that the method corresponding to the rectangle uses the instance variables corresponding to the oval. Figures below shows the connections for each instance variable. Here, the instance variable end is used by the methods queue*, insert, delete, Isempty*. All of the methods that use the variable *end* are connected through the variable *end*.

A class constructor (e.g. method queue) is an initialization function. It will generally access all instance variables in the class, and thus, share instance variables with virtually all other methods. Constructors create connections between methods even if the methods don't have any other relationships. Thus, constructor functions are removed from this model. Links between constructors queue and instance variables in figures below are represented as dashed lines. Destructor functions are also excluded from this model. The code for c++ class queue is shown as under and Figures 1,2,3,4,5 shows the graphical model of the class queue. The source code for the class queue is shown as under.

```
Class Queue
{

Private:    int *element, beg, end, size;

Public:   Queue(int s)
    { size=s; element=new int[size]; beg=0,end=s;}

     int Isempty()   {return end;}
     int Size()        {return size;}
      void insert(int item){
                  if (end==size+1)
                  printf("Queue is full\n");
                  else element[end++]=item;}
      int Delete()     { if (Isempty()=0)
                   printf("Queue is empty\n");
                   else{ beg++;
                   if(beg==size) end=0;  }}
};
```

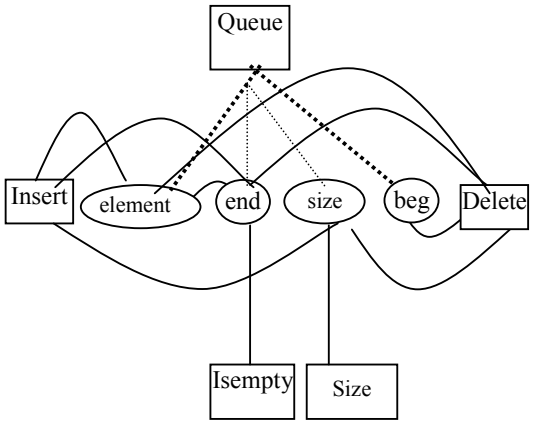

Fig.1: Graphical model for Class *Queue*

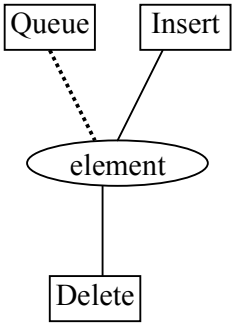

Fig.2: Connection for instance variable *Element*

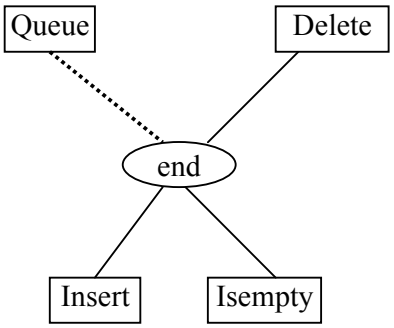

Fig.3: Connection for instance variable *End*

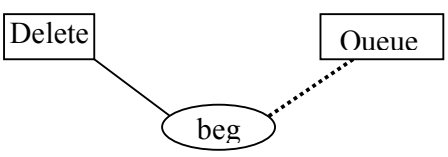

Fig.4: Connection for instance variable *beg*

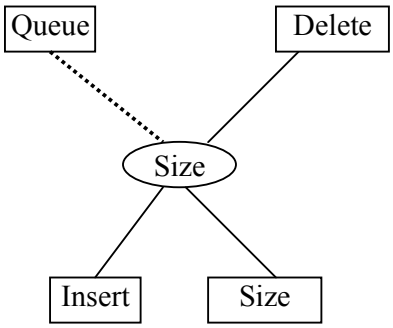

Fig.5: Connection for instance variable *Size*

The graphical model includes the information to define class cohesion. A method is represented as a set of instance variables directly or indirectly used by the method. We call the representation of a method an *abstracted method*, AM.

An instance variable is directly used by a method M if the instance variable appears as a data token in the method M. The instance variable is defined in the same class as M. DU(M) is the set of instance variables directly used by a method M.

A direct/indirect call relation defines the indirect use of an instance variable. A method M_1 is directly called by a method M if M is predecessor of M_1 in the call graph. Indirect call relations are the transitive closure of the direct call relations. Thus a method M_1 is indirectly called by method M if there is path from M to M_1 in statically determined call graph.

An instance variable is indirectly used by a method M if firstly the instance variable is directly used by another method M_1 which is called directly or indirectly by M, and secondly the instance variable directly used by M_1 is in the same object as M. IU(M) is a set of instance variables indirectly used by method M.

A class is represented as a collection of AM's where each AM corresponds to visible method in the class. The representation of a class is called an abstracted class, AC.

$AM(M) = DU(M) \cup IU(M)$
$AC(C) = [\ AM(M) \mid M \varepsilon V(C)\ ]$
V(C) is the set of all visible methods in class C and AM corresponds to visible methods defined only within the class.
The abstracted class of queue of Figure 1 is:
AC(Queue) = [{end}, {size}, {end, size, element}, {beg, size, end}]
The direct connectivity between methods is determined from class abstraction. If there exist one

or more common instance variables between two method abstractions then the two corresponding are directly connected as shown in Fig 6.

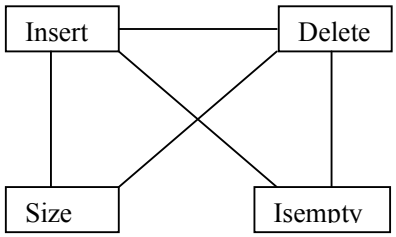

Fig 6:Connectivity between methods in class Queue

There is one measure of class cohesion given in [1] based on direct connectivity between methods. Let NP(C) is the total number of pairs of abstracted methods in AC(C). NP is the maximum possible number of direct or indirect connections in a class. If there are N methods in a class, Then NP(C) = N * (N-1)/2. Let NDC(C) is the number of direct connections in AC(C). Then Class cohesion is given by relative number of directly connected methods given by: **CC(C) = NDC(C) /NP(C)**

3.2 Measuring the data visibility of a class

DV is a measure of how visible (and therefore directly accessible) a class's data is. A class data is considered hidden (invisible) if all its instance fields are private and can only be manipulated via methods. DV uses a scale of 0-1 for each fields, where 0= fully hidden (private), 1= fully visible (public) and the value for partially visible (e.g. protected) fields lies between 0 and 1 whose value is calculated by dividing the number of classes in the system that can access the field divided by the total number of classes in the system, less one (i.e. the class itself). The approach taken to measure the level of data visibility is a class is based on the Attribute Hiding Factor (AHF) metric proposed by MOOD metrics team [3].

The DV is given by:-

$$DV = \frac{\sum_{m=1}^{A_d(C)} (1 - V(A_m))}{A_d(C)}$$

The final DV value for a class is the average visibility across all fields in the class. The closer the total measurement is to 1, more hidden is the data. Ideally a class should have a 1 (total hidden) value for DV. The description of the variables is shown in table 1.

Table 1: Description of variables used in DV

Variable	Description	Implementation in C++
TC	Total classes	Total number of classes
$A_d(C)$	Attributes defined in a class (not inherited)	Data members
$V(A_m)$	Visibility - % of the total classes from which A_m is visible	= 1 for attributes in public clauses; = 0 for those in private clauses; = $DC(C_i)/(TC-1)$ for attributes in protected clauses note: $DC(C_i)$ = descendants of C_i

3.3. Measurement of Encapsulation Factor (EF) for a class

EF is a metric to measure the encapsulation level of a class or a system. It is a singular metric that takes the two components of the class i.e. the attributes and methods. It is a function of two metrics (data hiding and cohesion) in the class. Ideally a class will have a value of 1 for DV and Value of 1 for CU its value lies in the range [0,1] where 0 means (minimal) and 1 means (maximal) encapsulation in the system or class. Classes with higher value are desirable.

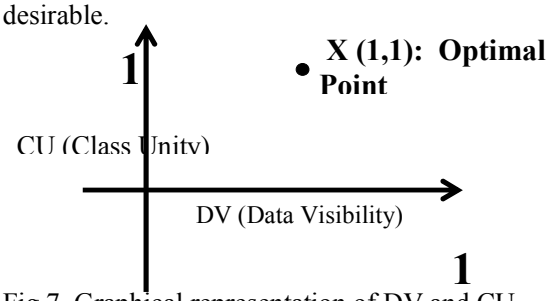

Fig.7. Graphical representation of DV and CU

As shown in figure 7 DV (Data visibility of a class) be the X- axis in range [0, 1] and CU (class unity) is the Y-axis in range [0, 1]. A class having 100% encapsulation (desired value) will have a value 1 for both DV and CU.

Using DV and CU values, we can get a point in the graph. To determine the encapsulation of a given class, we should observe the distance of the point to the optimal point. Using the Pythagoras theorem [6,7] , we can easily find the distance of a point to the optimal point.

$$EF = 1 - \frac{\sqrt{(1-DV)^2 + (1-CU)^2}}{\sqrt{2}}$$

$\sqrt{2}$ is hereby regarded as a normalization factor for the equation to obtain EF.

EF aims to indicate the level of encapsulation in the system or class. In general we aim for the high level of encapsulation i.e. EF is close to 1 is desired. The following are the best and worst possible cases.

Best Case (EF=1):

The best case for EF is 1 when both DV and CU are 1. Point obtained would be at (1,1) which is the optimal point and therefore has a distance of 0. So EF = 1-0 =1 for this case i.e.100%

Worst case (EF=0):

The worst case for EF=0 when both DV and CU are 0.Therefore the point obtained is at (0,0), which is the maximum distance from the optimal point. EF=1-1=0. Consider the C++ code shown above.

EF factor is calculated as:

Class cohesion CC= 5/6

Data Visibility =1

Encapsulation Factor= 0.83

Above values indicates that class is well encapsulated.

4. EF Metric Analysis

We give an outline of our approach. In our approach we perform a statistical analysis on EF metric as stated in section 3. We had collected three applications to statistically analyze EF. The applications were labeled as: System A, System B and System C. System A is the "Banking Application" that handles the transactions implemented in C++ and consisting of 5 classes. System B is the "Simulation of Hospital management System" in C++ having 3 classes and System C is the "Simulation of Sorting Master" in c++ that visualizes the process of sorting the numbers using various sorting algorithms and make maximum use of graphical functions. The encapsulation factor metric value for system A, B, C is shown in fig8, fig9 and fig10.

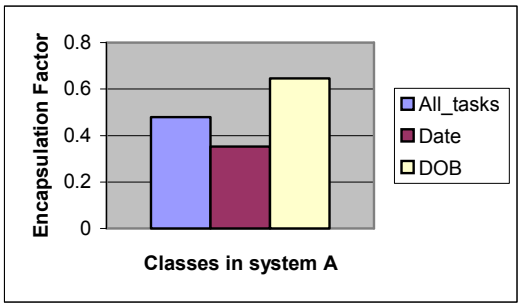

Fig. 8: Encapsulation factor for System A

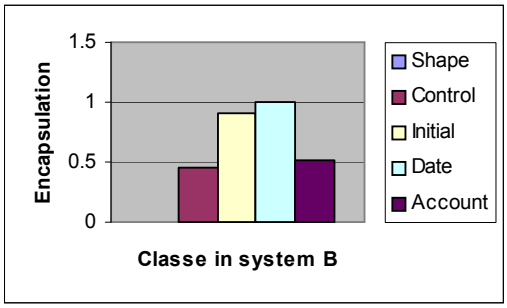

Fig. 9 : Encapsulation factor for System B.

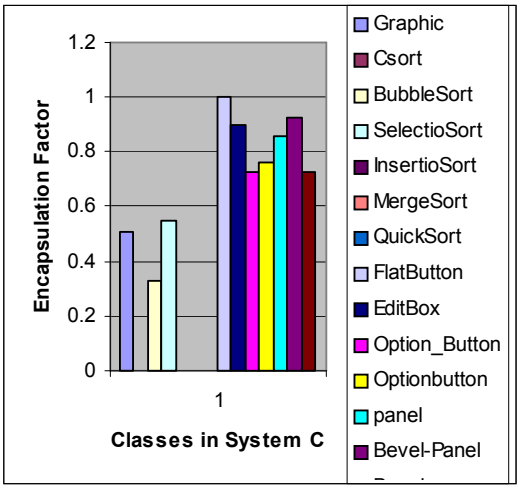

Fig 10: Encapsulation factor for System C

Table2 summarizes the software applications used in validating the proposed object-oriented metric for encapsulation. Table 3 shows the correlation analysis summary, which have been generated using the statistical formulas defined in [8].

Table 2: Summary of applications used to validate metric proposed for Encapsulation

System	A	B	C
Total no. of attributes	16	20	50
Total no. Of methods	31	19	47

Number of classes	5	3	14
Language	C++	C++	C++
Code Construct	Object-Oriented	Object-oriented	Good Object-Oriented
Use of Encapsulation Mechanism	Low	Low	High

Table 3: Correlation Analysis Summary

	EF x Cohesion	EF x DV
System A	0.75	-0.5
System B	0.40	0.91
System C	0.85	0.80

5. Discussion

The difference was presented in the metric already given by MOOD to measure the encapsulation and proposed one. The metrics MHF and AHF [3][4] are considered by MOOD to be measures of encapsulation. This indicates poor understanding of the concept of encapsulation [5].

Information hiding and encapsulation are not synonymous. Information hiding is only one part of the concept. Encapsulation can be thought of as an aggregate of two different but related terms, namely privacy and unity. A class that has only private data members will not necessarily be unified. Equally, a unified class may contain only visible data. So MHF and AHF are not measures of encapsulation. Further AHF metric contributes to such a measure but it is doubtful the MHF measure serves an equivalent purpose.

EF metric measures the encapsulation as a function of both attributes namely privacy and unity. Privacy is measured in terms of private data members (Number of private attributes in a class) and Unity is measured in terms of the attributes and methods (cohesion between the private, protected and public attributes and methods).

We make certain observations from table1 and table2. In table 2 it is shown that the three systems have different level of use of encapsulation mechanism. From correlation analysis summary in Table 3, we can study that in system A, cohesion is highly correlated with encapsulation resulting in lack of the data visibility. System B is highly correlated with data visibility resulting in lack of cohesion factor. System C having high correlation of encapsulation with both the factors i.e. data visibility and cohesion, suggesting well – encapsulated classes that can be reused, extended and easily maintained.

From statistical analysis, we can conclude that if a class is having encapsulation more than 80%, then there is proper use of encapsulation and there is no need to split the classes further.

6. Future Scope

We must nevertheless mention that applications used for the study were very small compared to large industry system. Therefore in terms of future scope we suggest that further characteristics of classes need to be studied to establish relationship between proposed metric and behavior of the classes. Further validation of the proposed metric can be done with an extended set of classes and further evaluation of our metric can be performed.

References

[1] James M. Bieman and Byung-kyoo kang "Cohesion and reuse in an Object-Oriented System" Symposium on Software Reusability pp 259 - 262, 1995.

[2] R. Chidamber and Kemerer "Towards a Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering Vol. 20 No. 6 pp.1994, 476-493.

[3] Rachel Harrison, Steve J. Counsell and Reuben V. Nithi. "An Evaluation of the MOOD set of Object Oriented software metrics" IEEE Transactions on Software Engineering VOL.24 No. 6 June 1998.

[4] Tobias Mayer and Tracy Hall "A Critical Analysis of Current OO Design Metrics" Software Quality Journal, Vol 8, 97-110, 1999.

[5] Tobias Mayer and Tracy Hall "Measuring OO Systems: A Critical analysis of the MOOD Metrics", Technology of object oriented languages and systems. pp 108, 1999.

[6] http://en.wikibooks.org/wiki/Geometry

[7] http://en.wikibooks.org/wiki/Statistics

[8] S.P Gupta "Statistical Methods",Published by Sultan Chand and sons 23, Daryaganj, New Delhi.