Analyzing the Complexity of Java Programs using Object - Oriented Software Metrics

Arti Chhikara[1]and R.S.Chhillar[2]

[1]Maharaja Agrasen College, Delhi, India.
[2]Deptt. Of Computer Sc. And Applications, Rohtak, India.

Abstract

Object-oriented technology has rapidly become accepted as the preferred paradigm for large-scale system design. With the help of this technology we can develop software product of higher quality and lower maintenance cost. It is evident that the available traditional software metrics are inadequate for case of object-oriented software, as a result a set of new object oriented software metrics came into existence. Object Oriented Metrics are the measurement tools adapted to the Object Oriented paradigm to help manage and foster quality in software development.

Measurement of software complexity has been of great interest to researchers in software engineering for some time. Software complexity has been shown to be one of the major contributing factors to the cost of developing and maintaining software. In this research paper we investigate several object oriented metrics proposed by various researchers. These object oriented metrics are than applied to several java programs to analyze the complexity of software product.

Keywords: Object Oriented Software Development, Software Metric, Software Product, Java.

1. Introduction:

Object-oriented technologies reflect a natural view of the world. Object-oriented software is easier to maintain because its structure is inherently decoupled. Object Oriented Analysis and Design of software provide many benefits to both the program designer and the user. This technology promises greater programmer productivity, better quality of software and lesser maintenance cost [1,2].

OO approaches control complexity of a system by supporting hierarchical decomposition through both data and procedural abstraction [3]. However, as Brooks points out, "The complexity of software is an essential property, not an accidental one" [4]. The OO decomposition process merely helps control the inherent complexity of the problem; it does not reduce or eliminate the complexity. Measurement of the software complexity of OO systems has the potential to aid in the realization of these expected benefits. Software complexity has been shown to be one of the major contributing factors to the cost of developing and maintaining software [6]. According to Coad and Yourdon [5], a good OO design is one that allows trade-offs of analysis, design, implementation and maintenance costs throughout the lifetime of the system so that the total lifetime costs of the system are minimized. Software complexity measurement can contribute to making these cost trade-offs in two ways. These are:

1) To provide a quantitative method for predicting how difficult it will be to design, implement, and maintain the system.

2) To provide a basis for making the cost trade-offs necessary to reduce costs over the lifetime of the system.

In this research paper different java programs are studied and object oriented software metrics are applied to them and a study of complexity is made based on the results obtained by applying object oriented metrics to different java programs.

The rest of the paper is organized as follows. Section 2 give a brief overview of object oriented metrics that we have used in our paper. Section 3 presents an example of java source code. Section 4 presents results obtained by applying object oriented metrics to java source code. Section 5 presents conclusion.

2. Literature Research

2.1 Object Oriented Metrics

One of the most widely referenced sets of object-oriented software metrics has been proposed by Chidamber and Kemerer [7,11]. At the 1991 Object Oriented Programming Systems, Languages and Applications conference (OOPSLA), Shyam Chidamber and Chris Kemerer presented a paper [7] outlining six metrics for use with object-oriented programming languages. The metrics used in this study are given below:

1 **Weighted Method per Class(WMC**): WMC is defined as the sum of the complexities of all methods of a class. If there are n methods of complexities

$c_1, c_2, \ldots\ldots\ldots\ldots, c_n$ are defined for a class C. The specific complexity metric that is chosen should be normalized so that nominal complexity for a method takes on a value of 1.0 [16].

$$WMC = \sum c_i \quad \text{for } i=1 \text{ to } n$$

The number of methods and their complexity are reasonable indicators of the
amount of effort required to implement and test a class. In addition, the larger the number of methods, the more complex is the inheritance tree (all subclasses inherit the methods of their parents). Finally, as the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse. For all of these reasons, WMC should be kept as low as is reasonable [16].

2 Depth of Inheritance Tree(DIT): This metric is "a measure of how many ancestor classes can potentially affect this class." [7,10] The deeper a class is in the inheritance the more behavior it is likely to inherit from its superclassses. Deep inheritance trees are indicative of complex designs. This metric is useful as a design aid in designing classes that make use of inherited methods.

3 Number of Children(NOC): The NOC is the number of immediate subclasses in the hierarchy. High NOC indicates high reuse. But, if there are a large number of children of a class, then the abstraction level of that parent class is reduced. If a class has too many children, it may indicate misuse of sub-classing. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing[7,10].

4 Response For a Class (RFC): This metric is a count of all member functions called by any member function in the class being measured. Member functions in the class and member functions of other classes are both counted equally. It is "considered a measure of attributes of an object. Since it specifically includes methods called from outside the object, it is also a measure of communication between objects." [7]

Several studies have been conducted to validate CK's metrics. Their metrics have been criticized, specially the LCOM metric, for being too ambiguous for practical applications and for not being language independent [12]. Basili et al. [13] presented the results of an empirical validation of CK's metrics. Tang et al. [14] validated CK's metric suit using real time systems.

Li, et al. have also empirically evaluated C&K's metrics as being predictors of maintenance effort [15]. In addition, Li, et al. [15] proposed new metrics that were used in their study including:

5 Message passing coupling: The Message Passing Coupling metric measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It allows for conclusions on the message passing (method calls) between objects of the involved classes. This allows for conclusions on reusability, maintenance and testing effort.

6 Data abstraction coupling: Data abstraction coupling is a count of total number of instances of other classes within a given class. It is the count of total number of external classes the given classes uses.

7 Number of local subunits: The number of local subunits is the total number of functions and procedures defined for a class. Classes with large number of operations are harder to maintain and are more fault prone.

Morris [8,9] in 1989 made some important observations on OO code and proposed candidate metrics for productivity measurement:

8 Inheritance Dependencies: This metric is intended to reflect characteristics of the inheritance tree. Morris suggests that "it may be possible to determine a range of values within which the inheritance tree depth should be maintained."[9]:

This metric is calculated using the following equation:

Inheritance tree depth = max (inheritance tree path length)

9 Factoring Effectiveness: Morris states that "inheritance hierarchies are optimized via a process called factoring. The purpose of factoring is to minimize the number of locations within an inheritance hierarchy in which a particular method is implemented."[9]
It is calculated as below:

Factoring Effectiveness = Number of unique methods / Total number of methods

10 Reuse Ratio: The reuse ratio, RR is given by [18]:

Reuse Ratio = Number of Superclasses/Total number of Classes

11 Specialization Index: Specialization Ratio (SR): Specialization ratio, SR is given as [18]:

Specialization Index = Number of Subclasses/Number of Superclasses

3. Definition of Metric

To better define and understand how these metrics are calculated using java source code example is used.

3.1: Java source code[1,2]

```
import java.io.*;
public class employee
//employee class
{
  DataInputStream in=new
DataInputStream(System.in);
  private String name;
//employee name
  public  int number;
//employee number
  public void getinfo()
   {
     System.out.println("Enter  name:");
     name= in.readLine();
     System.out.println ("enter number :") ;
      number=Integer.parseInt(in.readLine()) ;
   }
  public void putinfo()
   {
     System.out.println("The name is:" +name);
     System.out.println ("Number=" +number);
      }
public void show()
{
 System.out.println("End of Employee Class");
}
}
public class generalmanager extends employee
//generalmanager class
{
 private String title ;
 private double dues ;
 private int count;
 count = total
 public  void getinfo()
  {
    super.getinfo();
   System.out.println("enter title :");
    title=in.readLine();
    Console.WriteLine("enter golf club dues:");
    dues=double.parseDouble(in.readLine());
  }
public void putinfo()
  {
   super.putinfo();
   System.out.println(count);
   System.out.println ("title:" +title);
   System.out.println("dues:"+dues);
  }
 public void show1()
  {
System.out.println("End of manager class");
 }}
public class engineer extends  employee
// engineer class
{
 private int pubs ;
 public void getinfo()
 {
  super.getdata();
  System.out.println ("enter number of pubs:") ;
  pubs=Integer.parseInt(in.readLine());
 }
 public void putinfo()
 {
 super.putinfo();
 System.out.println ("number of pubs:" +pubs);
 }
}
public class worker extends employee
// worker class
{
  private int a;
  public int hours;
  public void getinfo()
  {
   super.getdata();
   System.out.println ("Enter number of hours:") ;
   hours=Integer.parseInt(in.readLine());
  }
 public void calculate( )
{
 int total=0;
 total = LEN*40;
}
public void putinfo()
{
 super.putinfo();
 System.out.println ("number of hours :" +hours) ;
 System.out.println ("Total:" +total);
 }
}
public class hourlyemployee extends worker
//hourlyemployee class
```

```
{
 private double sal;
 public void getinfo()
{
  super.getinfo();
  System.out.println (”enter number of hours:”) ;
 hours=Integer.parseInt(in.readLine());
 }
  public void salary()
{
 sal=super.hours*250;
// calling superclass instance variable
 }
  public void putinfo()
{
 super.putinfo();
 System.out.println (“The salary is: “ +sal);
 }
public static void main(String args[])
//main method
{
  generalmanager m1 = new generalmanager();
  generalmanager m2 = new generalmanager();
  technician s1= new scientist();
  worker L1 = new worker();
  hourlyemployee h1 = new hourlyemployee();
 System.out.println (“Enter data for manager 1”);
//get data for several employees
 m1.getinfo();
 System.out.println (“Enter data for manager 2”);
 m2.getinfo();
 System.out.println (“Enter data for scientist 1”);
 s1.getinfo();
 System.out.println (“Enter data for laborer 1”);
 L1.getinfo();
System.out.println (“Enter data for hourlyemployee
1”);
 h1.getinfo();
 System.out.println (“Data on manager 1”);
 m1.putinfo();
 System.out.println (“Data on manager 2 “);
 m2.putinfo();
 System.out.println (“Data on scientist 1”);
 s1.putinfo();
 System.out.println (”Data on Laborer 1”);
 L1.putinfo();
 System.out.println (“Data on hourly employee”);
 h1.putinfo();
}
```

3.2 Class Diagram for java source code:

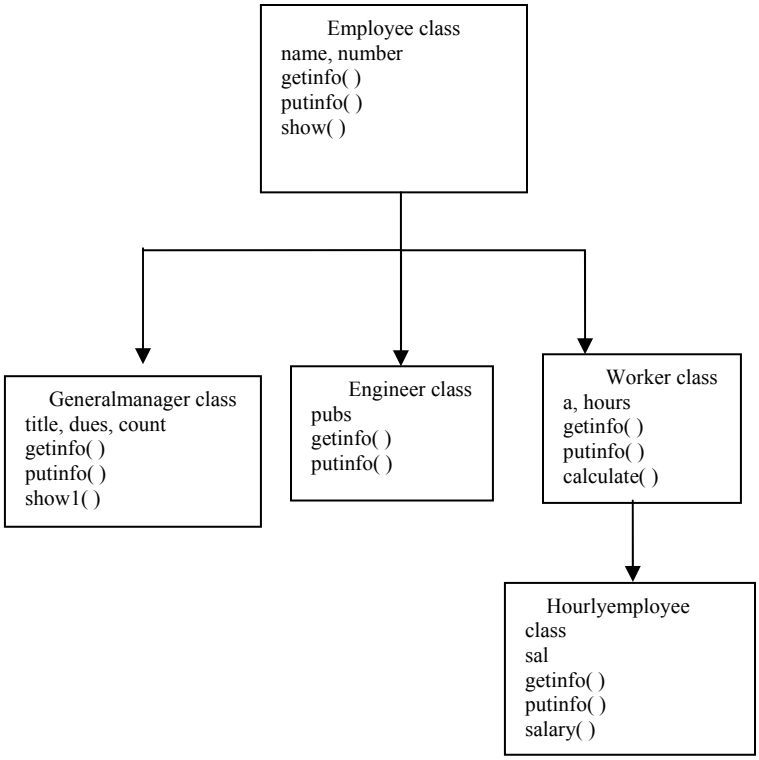

Fig 1: Class Diagram

3.3 Object Oriented Software Metrics Applied on Example 1:

1. WMC (Weighted Method per Class): WMC is calculated by counting the number of methods in each class [4].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
WMC	3	3	2	3	3

2. RFC (Response for a Class): The RFC is the number of functions or procedures that can be potentially be executed in a class. Specifically, this is the number of operations directly invoked by member operations in a class plus the number operations themselves [4].

Metric	Employee class	Manager Class	Engineer class	Worker class	Hourlyemployee class
RFC	3	5	4	5	7

3. DIT (Depth of Inheritance tree): The depth of inheritance is defined to be the level of the class in the inheritance hierarchy, with the root class being Zero [4].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
DIT	0	1	1	1	2

4. NOC (Number of Children): The number of children is the number of direct descendents for a class [4].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
NOC	3	0	0	1	0

5. MPC (Massage Passing Coupling): Message Passing coupling is the count of total number of function and procedure calls made to external units [7].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
MPC	0	2	2	2	4

6. DAC (Data Abstraction Coupling): Data Abstraction coupling is the count of total number of instances of other classes within a given class [7].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
DAC	0	1	0	1	0

7. NUS (Number of Subunits): The number of subunit is the total number of functions and procedures defined for the class [7].

Metric	Employee class	Manager Class	Engineer class	Laborer class	Hourlyemployee class
NUS	3	3	2	3	3

8. Inheritance dependencies(ID): This metric is calculated using the following equation:

Inheritance tree depth=max(inheritance tree path length)

So referring the above class diagram

Inheritance tree depth = 3

9.Factoring effectiveness(FE): This metric is calculated using the following equation:

Factoring effectiveness = No. of unique methods / Total no. of methods

= 4/14

= 0.29

10.Specialization index(SI): This metric is calculated using the following equation:

Specialization index= Total no. of subclasses / total no. of superclasses

From the above class diagram

Specialization index =5/2

= 2.5

11.Reuse ratio(RR): This metric is calculated using the following equation:

Reuse ratio=Total no. of superclasses / Total no of classes

Referring above class diagram

Reuse ratio =2/5

=0.4

4. Study of the complexity of Java programs

These metrics were calculated and tested on 20 Java programs and following results are obtained.

Table 1 shows the metric value for 20 java programs and Table 2 shows the statistical values calculated for the metric values obtained from 20 java programs

Table 1: Metric Values Calculated for JAVA Programs

Metrics Type	Program Number																			
	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**P14**	**P15**	**P16**	**P17**	**P18**	**P19**	**P20**
WMC	3.00	2.25	1.65	2.00	2.00	1.25	2.25	2.00	2.00	1.65	3.33	1.50	2.00	2.00	1.00	1.67	2.00	3.33	2.00	2.00
RFC	2.00	3.00	3.33	2.00	3.33	3.33	2.67	3.00	4.48	1.50	3.00	3.33	2.50	2.00	3.33	3.33	2.00	4.10	3.00	2.00
DIT	2.00	1.00	1.00	1.00	0.50	2.00	0.50	0.33	1.00	1.00	0.75	0.50	0.50	1.00	1.00	2.25	0.33	0.33	0.50	0.50
NOC	2.00	0.75	0.50	1.00	0.50	1.50	0.50	0.50	0.50	0.65	0.65	1.00	1.00	1.00	0.75	1.75	0.67	0.50	0.50	0.50
MPC	2.00	0.33	0.20	0.33	0.00	0.00	0.20	0.00	0.00	0.00	0.50	0.50	0.33	0.33	0.33	0.00	0.00	0.33	0.00	0.00
DAC	0.30	0.00	0.00	0.40	0.67	0.00	0.00	0.00	0.33	0.50	0.67	0.50	0.50	0.67	0.00	0.33	0.33	0.30	0.30	0.40
NUS	3.00	2.00	1.65	1.65	2.00	2.00	2.00	1.67	1.67	1.33	1.50	1.50	1.50	2.50	2.00	1.67	1.67	2.50	2.00	2.00
ID	2.00	1.00	0.50	0.50	1.00	0.33	2.00	1.00	1.00	1.00	0.50	0.50	1.00	1.00	2.00	2.25	2.00	0.50	0.50	0.33
FE	0.50	0.50	0.30	0.30	0.50	0.67	0.67	0.67	1.25	0.50	0.33	0.67	0.67	0.33	0.33	0.33	0.50	0.67	0.50	0.50
SI	2.00	2.00	1.00	1.00	1.00	3.00	2.00	2.00	1.00	1.00	2.00	3.00	1.00	1.00	1.00		2.00	3.00	2.00	2.00
RR	0.25	0.33	0.25	0.25	0.50	0.50	0.33	0.33	0.30	0.33	0.25	0.75	0.50	0.50	0.30	0.25	0.25	0.30	0.50	0.30

Table2: Statistical Values Calculated for JAVA Programs

Metric Type	Minimum	Maximum	Mean	Median	Stand. Deviation
WMC	1.00	3.33	2.04	2.00	0.59
RFC	1.50	4.48	2.86	3.00	0.77
DIT	0.33	2.25	0.89	0.87	0.57
NOC	0.50	2.00	0.83	0.66	0.44
MPC	0.00	2.00	0.26	0.20	0.44
DAC	0.00	0.67	0.31	0.33	0.23
NUS	1.33	3.00	1.89	1.83	0.40
ID	0.33	2.25	1.04	1.00	0.64
FE	0.30	1.25	0.53	0.50	0.21
SI	1.00	3.00	1.75	2.00	0.71
RR	0.25	0.75	0.36	0.31	0.13

After analyzing the Table 1 and Table 2 following points are observed regarding complexity of the java programs

1 Weighted Method per Class metric predicts time and effort that is required to build and maintain a class. A high value of WMC has been found to lead to more faults. Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse. A study of 20 java programs suggest that an increase in the average WMC increases the complexity and decreases quality. As programs with large number of methods are more prone to bugs and complex to understand.

2 The RFC metric is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some methods in the class. This includes all methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class. From our study we found that Java programs are less complex as the mean value of this metric is low for java programs.

3 The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is in the hierarchy, the more methods it is likely to inherit, making it more complex. Deeper trees constitute greater design complexity, since more methods and classes are involved, but at the same time reusability also get increase due to inheritance. Java programs have intermediate value for DIT metric.

4 The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator of the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of subclassing. However high NOC indicates high reuse, since inheritance is a form of reuse.. A class with many children may also require more testing. High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desired. In Java the value of this metric depends on program to program. All classes do not have the same number of sub-classes. However, it is observed that for better results, classes higher up in the hierarchy should have more sub-classes then those lower down.

5 Message passing coupling metric measures the numbers of messages passing among objects of the class. A larger number indicates increased coupling between this class and other classes in the system. This makes the classes more dependent on each other which increases the overall complexity of the system and makes the class more difficult to change. The assumption behind this metric is that classes interacting with many other classes are harder to understand and maintain. When we applied object oriented metrics on several java programs, we observed that the value of Message Passing Coupling (MPC) metric is low for java programs.

6 Data Abstraction Coupling metric measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes. It is the count of total number of external classes the given classes uses. Software complexity increases with increasing DAC.As Java is an object oriented language so data is given more importance than procedures. Data is hidden from the outside world. The value of this metric is low for java programs.

7 The Number of Subunit metric is the total number of functions and procedures defined for the class. As the number of functions and procedures grow, class become more fault prone. The complexity also get increase with increase value of local subunits metric. The value of this metric is found to be low for java programs.

8 Inheritance Dependencies metric is intended to reflect characteristics of the inheritance tree. Morris suggests that “it may be possible to determine a range of values within which the inheritance tree depth should be maintained. inheritance tree depth is likely to be more favorable than breadth in terms of reusability via inheritance. However, A deeper tree is more difficult to test than a broader one. The greater the value of this metric, more will be the complexity of programs. Comprehensibility may be diminished with a large number of inheritance layers.

9 Morris states that “inheritance hierarchies are optimized via a process called factoring. The purpose of factoring is to minimize the number of locations within an inheritance hierarchy in which a particular method is implemented.”[9] Highly factored applications are more reliable for reasons similar to

those that argue that such applications are more maintainable. The smaller the number of implementation locations for the average task, the less likely that errors were made during coding. The more highly factored an inheritance hierarchy is the greatest degree to which method reuse occurs. The more highly factored an application is, the smaller the number of implementation locations for the average method.

10 Specialization Index metric measures the extent to which subclasses override their ancestors classes. This index is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class. This metric was developed specifically to capture the point that classes are structured in hierarchy which reuse code and specialize code of their superclasses. It is well-defined, not ambiguous and easy to calculate. However, it is missing theoretical and empirical validation. It is commonly accepted that the more the Specialization Index is elevated, the more difficult is the class to maintain. The value of this metric is high for java programs, as java classes are more usable.

11 Reuse ratio measures reuse via inheritance. A high value of this metric indicates a deep class hierarchy with high reuse. Reuse ratio is the percentage of classes that are derived from. Reuse ratio varies in the range {0,1}. When the value of this metric is zero, there is no inheritance. As the value of this metric approaches 1, the inheritance tree deepens in a chain form with exactly one root and one leaf. When this metric is applied to several java programs we got intermediate results.

5 Conclusion and Future Work

The primary objective of this study was to investigate the applicability of Object–Oriented software metrics to measure the complexity of a Java software application. Complexity of Java applications can be evaluated at several dimensions(Size,method,class,inheritance,cohesion etc) using a variety of available software metrics from Software Engineering Domain. In this research paper we have presented a set of eleven well established object-oriented metrics that can be used to rank programs on their complexity values, to assess testability and maintainability of the programs. From this study we conclude that there should be a compromise among internal software attributes in order to maintain a high degree of reusability while keeping the degree of complexity and coupling as low as possible.

However it is still insufficient, needs further in depth study and future work will focus on empirical validation of object oriented metrics in multi languages environment. But we still expect that our analysis can be used as a reference by software developers for building a fault free, reliable, and easy to maintain software product in Java

References:

[1] Patrick Naughton & Herbert Schildt “java:The complete reference”, McGraw-Hill Professional, UK, 2008.

[2] Er. V.K. Jain. “The Complete Guide to java programming”, First Edition, 2001.

[3] G. Booch, Object-Oriented Design with Applications (The Benjamin/Cummings Publishing Company, Redwood City, CA , 1991; ISBN: 0-8053-0091-0).

[4] F.P. Brooks, No Silver Bullets: Essence and Accidents of Software Engineering, Computer, Vol. 20, No. 4 (Apr 1987) 10-19.

[5] P. Coad and E. Yourdon, Object-Oriented Design (Yourdon Press, Englewood Cliffs, NJ, 1991; ISBN: 0-13-630070-7).

[6] R.B. Grady, Practical Software Metrics for Project Management and Process Improvement (Prentice Hall, Englewood Cliffs, NJ, 1992; ISBN: 0-13-720384-5).

[7] S. Chidamber, and C. Kemerer, “Towards a Metrics Suite for Object Oriented Design,’’ Object Oriented Programming Systems, Languages and Applications (OOPSLA), Vol 10, 1991, pp 197-211

[8] Michael W. Cohn, William S. Junk, “Empirical Evaluation of a Proposed Set of Metrics for Determining Class Complexity in Object-Oriented Code”, A Thesis, College of Graduate Studies University of Idaho, April 1994

[9]K. Morris, “Metrics for Object-oriented Software Development Environments,” Masters Thesis, MIT, 1989.

[10]Chidamber, S. and Kemerer, C.” A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493,1994.

[11]Chidamber, S., Darcy, D., Kemerer, C." Managerial use of Metrics for Object Oriented Software": an Exploratory Analysis, IEEE Transaction on Software Engineering, vol. 24, no. 8, pp. 629-639,1998.

[12] Churcher, N.I. and M.J. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics," ACM Software Engineering Notes, vol. 20, no. 2, April 1995,pp. 69–76.

[13] Basli VR, Briand LC, Melo WL. "A validation of object oriented design metrics as quality indicators". Technical Report, University of Maryland, Department of Computer Science,1-24, 1995.

[14] Tang MH, Kao MH. "An empirical study on object-oriented metrics". Proceedings 23rd Annual International Computer Software and Application Conference. IEEE Computer Society, 242-249,1999.

[15] Li. W. "Another Metric suit for object-oriented programming". The journal of system and software 44(2),155-162,1998.

[16] Roger S. Pressman: Software Engineering, A practioner's Approach, Fifth Edition,2001.

[17] R. Kolewe, "Metrics in Object-Oriented Design and Programming,'' Software Development, Vol. 1, No. 4, October 1993, pp. 53-62.

[18] Jacobson Ivar : Object Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Publishing Company,1993