An Empirical Validation of Object Oriented Design Metrics in Object Oriented Systems


1Department of Computer Science and Engineering, Ellenki, College of Engineering and Technology, Hyderabad, India.
2Department of Computer Science and Engineering, Bharathi Institute of Technology, Hyderabad, India
3Department of Computer Science and Engineering, R&D, Ellenki College of Engineering and Technology, Hyderabad, India.

Correspondence Author: Amjan Shaik

Abstract
Object Oriented (OO) Design and Development have become popular in today’s software development environment. To produce high quality object oriented applications, a strong emphasis on design aspects, especially during the early phases of software, development is necessary. Design metrics play an vital role in helping developers to appreciate design aspects of software ie improve software quality and developer productivity. Metrics data provides a quick response about the software product quality to software designers and managers. By analyzing the metric data, we can forecast the quality of the OO system. Through this paper, we are trying to edify about the OOD, metrics, quality and the relationship between these. In this paper, we provide empirical evidence underneath the role of OOD Metrics specifically a subset of the CK Metric suite.

Keywords: object oriented design, metrics, quality, OODMAJ

INTRODUCTION
Object Oriented Design and Development is an interesting area of current research and many authors have done great deal of work in recent years. In fact Object Oriented Development requires not only a different approach to design and implementation, but also a different approach to software metrics. To produce high quality Object Oriented applications a strong emphasis on design aspects is highly necessary. Software metrics make it possible for software engineers to measure and predict software processes, necessary resources for a project and products relevant for a software development effort. A software measure provides software engineers with a means of quantifying the assessment of a software product. Measurement can be used throughout a software project to assist in estimation, productivity assessment, quality control and project control. The design of complex software based systems often proceeds with virtually no measurement.

The introduction of Object Oriented Methods to software development has changed the process of building and managing software in a profound way. Changes in design and implementation techniques also require new ways of measuring software systems. While some of the traditional or non Object-Oriented Software Metrics can be transformed to Object Oriented environments, completely new metrics are also needed. Design Metrics (Albert Dieter Ritzhaupt, 2004) play an important role in helping developers to understand design aspects of software techniques and, hence, improve software quality and developers productivity. In addition, the focus on process has increased the demand for software measures or metrics with which one can manage the process. The need for such metrics is particularly rare, when an organization is adopting a new technology for which a suitable metric design plays a crucial role. We present existing and new software metrics useful in the different phases of the Object Oriented Software Development cycle. The first step towards dividing the software development process into pieces is to separate it time wise into four phases: Inception, elaboration, construction, and transition. Each phase is further divided into one or more iterations. The primary goal of the inception phase is to establish the business case for going forward with the project. The primary goal of the elaboration phase is a stable architecture, to guide the system through its future life. The general objective of the construction phase is indicated by its initial operational capability that signifies a product ready for beta testing. The transition phase ends with formal releases. The importance of properly defined metrics is of immense importance in the Object Oriented design. If the metrics are properly defined, we can avoid problems that will be more expensive to
correct during the latter phases of Object Oriented Software Development. This helps researchers and practitioners better in understanding and selects software metrics suitable for their purposes. In today’s software industry, the aim is to deliver high quality software product to the customers. No doubt that the software quality can make or break a company. So software Quality plays a vital in the software industry. Table: 1 Key terms in object oriented systems development

<table>
<thead>
<tr>
<th>Class</th>
<th>A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which objects can be instantiated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.</td>
</tr>
<tr>
<td>Method</td>
<td>An operation upon object, defined as part of the declaration of a class. Methods are operations but not all operations are actual methods declared for a specific class.</td>
</tr>
<tr>
<td>Message</td>
<td>A request that an object makes of another object to perform an operation.</td>
</tr>
<tr>
<td>Instantiation</td>
<td>The process of creating an instance of the object and binding or adding the specific data.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>A relationship among classes wherein one class shares or methods defined in one (for single inheritance) or more (for multiple inheritance) other classes.</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>The ability of an object to interpret a message differently at execution depending upon the super class of the calling object.</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>The process of bundling together the elements of an abstraction that constitute its structure and behavior.</td>
</tr>
<tr>
<td>Cohesion</td>
<td>The degree to which the methods within a class are related to one another.</td>
</tr>
<tr>
<td>Coupling</td>
<td>Object A is coupled to object B if and only if A sends a message to B.</td>
</tr>
<tr>
<td>Design Size</td>
<td>A measure of the number of classes used in a design.</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>Hierarchies are used to represent different generalization-specialization concepts in a design.</td>
</tr>
<tr>
<td>Abstraction</td>
<td>A measure of the generalization-specialization aspect of the design.</td>
</tr>
<tr>
<td>Complexity</td>
<td>A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.</td>
</tr>
</tbody>
</table>

Metrics help validate and calibrate generic models of software productivity and reliability an object oriented program paradigm uses encapsulation, inheritance and polymorphism, and has different program structure than in procedural languages. The design components that are exclusive and define the architecture of an OOD are objects, classes and the relationship between them. Object oriented design is intended to capture the fundamental structure of an object oriented program. Thus a set of components which can help to evaluate, represent and implement an object oriented design should include attributes, methods, objects (classes), relationships and class hierarchies. Table: 1 Summarizes the key terms in object oriented development environment. Software quality must be addressed during the whole process of software development. Measuring software quality in the early stages of software development is the key to develop high quality software. Product quality has some attributes such as functionality, effectiveness, understandability, reusability and maintainability. Large number of software metrics have been proposed in software engineering to measure the quality attributes of the software in early stages. We can estimate the overall design quality of the system from design information. There are so many models, which can be effectively used in monitoring the quality of software product

**QUALITY FACTORS**

In reality object oriented development has proved its value for systems that must be maintained and modified. The concepts of object oriented design metrics (Jiemet, al. 2008) are well established and many metrics relating to product quality have been developed and used. With object oriented analysis and design methodologies gaining popularity, it is time to start investigating object oriented design metrics with respect to software quality (Robert Martin,1994). Measuring quality in the early stage of software development is the key to develop high quality software. There must be a way to assess object oriented software quality as early as possible in the development cycle. The factors (McCall, 1977) that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects recovered during testing) and
2. Factors that can be measured only indirectly (e.g. usability or maintainability)

One of the earliest software product quality models was suggested by McCall (1977) and his colleagues. They proposed a useful categorization of factors that affect software quality. McCall’s Quality factors are shown in figure 1.

- Maintainability
- Flexibility
- Testability
- Interoperability

**Product Revision**

- Correctness
- Usability
- Efficiency
- Reliability
- Integrity

**Product Transition**

- Portability
- Reusability

**Product operation**

Figure1: McCall’s Quality factors

Source: (Pressman, Fifth Edition)
International efforts have led to development of a standard for software product quality measurement, ISO9126. All these models vary in their hierarchical definition of quality. ISO 9126 standard was developed to identify quality attributes for software products. The international standard ISO 9126 is built on six quality attributes those are Functionality, Reliability, Usability, Efficiency, Maintainability and Portability.

SYNOPSIS OF OBJECT ORIENTED DESIGN SOFTWARE METRICS

Software metrics are very useful to assess different attributes related to the quality of software product and the software development process. Metrics help developers and designers to better handle quality and have a better overview of the system. A key element of engineering is measurement. Measures are used to better understand the attributes of the model that we create. But, most important, we use measurements to assess the quality of the engineered product or the process used to build it. We attempt to derive a set of indirect measures that lead to metrics that provide an indication of the quality of some representation of the software. Realizing the importance of software metrics, numbers of metrics have been defined for software. Software metrics are numerical data related to software development. Metrics strongly support software project management activities. They relate to the four functions of management as follows:

Planning - Metrics serve as a basis of cost estimating, training planning, and resource planning, scheduling, and budgeting.

Organizing - Size and schedule metrics influence a project's organization.

Controlling - Metrics are used to status and track software development activities for compliance to plans.

Improving - Metrics are used as a tool for process improvement and to identify where improvement efforts should be concentrated and measure the effects of process improvement efforts.

Metrics habitually categorized into Project Metrics and Design Metrics. Project metrics are used to predict project needs, such as staffing levels and total effort. They measured the dynamic changes that have taken place in the state of the project, such as how much has been done and how much is left to do. Project metrics are more global and less specific than the design metrics. Unlike the design metrics, project metrics do not measure the quality of the software being developed. Design metrics are measurements of the static state of the project design at a particular point in time. These metrics are more localized and prescriptive in nature. They look at the quality of the way the system is being built. Design metrics can be divided into

- Traditional Metrics
- Object oriented Design Metrics

Figure 2: Metrics hierarchy

Source: (M. Punithavalli, 2005)

Traditional Metrics

In an object-oriented system, traditional metrics (L. Rosenberg and L. Hyatt, 1995) are generally applied to the methods that comprise the operations of a class. A method is a component of an object that operates on data in response to messages and is defined as part of the declaration of a class. Methods reflect how a problem is broken up and the capabilities other classes expect of a given class. Three traditional metrics are discussed here:

- Cyclomatic complexity
- Line counts (size).
- Comment percentage

Cyclomatic Complexity (CC):
The cyclomatic complexity (McCabe) is used to evaluate the application of an algorithm. A method with a low cyclomatic complexity may imply that decisions are deferred through message passing, not that the methods is not complex. The cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. Although this metric is specifically applicable to the evaluation of complexity, it also is related to all of the other attributes.

Line Counts (Size):
Various line counts are also applied to methods. These include counting all physical lines of code the number of statements and the number comment lines. Thresholds for evaluating the meaning of size measures may have to vary greatly depending on the coding language. However, since size limitations are based on ease of understanding by the developers and maintainers, routines of large size will always pose a
higher risk in attributes such as Understandability, Reusability, and Maintainability. This metric can be used to evaluate all the attributes, but most often is a measure of Understandability, Reusability, and Maintainability.

**Comment Percentage:**
The Line Count metric can be expanded to include a count of the number of comments, both on line and stand-alone. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. Since comments assist developers and maintainers, this metric is used to evaluate the attributes of Understandability and Reusability

**Object Oriented Design Metrics:**
Object oriented design metrics is an essential part of software environment. This study focus on a set of object oriented metrics that can be used to measure the quality of an object oriented design. The metrics for object oriented design focus on measurements that are applied to the class and design characteristics. These measurements permit designers to access the software early in process, making changes that will reduce complexity and improve the continuing capability of the design. A significant number of object oriented metrics have been developed in literature. For example, metrics proposed by Chidamber & Kemerer metrics (Chidamber and Kemerer, 1994), MOOD metrics, Lorenz and Kidd metrics etc. C.K metrics are the most popular among them. Another comprehensive set of metrics is MOOD metrics.

**C.K. Metrics**
C.K metric suite (Chidamber and Kemerer, 1994) offers informative insight into whether developers are following object oriented principles in their design. They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software. Chidamber and Kemerer proposed six metrics; the following discussion shows their metrics.

**Weighted Methods per Class (WMC)**
The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods (method complexity is measured by cyclomatic complexity). The second measurement is difficult to implement because not all methods are accessible within the class hierarchy because of inheritance. Consider a Class $C_i$ with Methods $M_{i1},...M_{ia}$ that are defined in the class. Let $c_{i1},...,c_{ia}$ be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{r} c_i$$  \hspace{1cm} (1)

The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children, since children inherit all of the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse (Jaana Lindroos, 2004). These metric measures are understandability, reusability, and maintainability.

In figure 3, WMC for Library is 3.

**Depth of Inheritance (DIT)**
The depth of a class within the inheritance hierarchy is maximum number of steps from the class node to the root of the tree and is measured by number of ancestor class. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. Deeper trees constitute greater design complexity, since more methods and classes are involved. The deeper a particular class is in the hierarchy, the greater potential reuse of inherited methods. For languages that allow multiple inheritances, the longest path is usually taken (AmjanShaik, 2010).
Number of Children (NOC)
The NOC metric equals to number of immediate subclasses subordinated to a class in the class hierarchy. Greater the number of children, greater the reuse, since inheritance is a form of reuse. The greater is the number of children the greater is the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing. 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 of the methods in that class (Ramanath Subramanyam, 2003).

C1

C1

C2

C3

C21

C31

C32

C33

C311

Figure 5: Number of Children Source: (Amjan Shaik, 2010)
In the preceding example the NOC for C3 is 3 i.e. C31, C32, C33.

Coupling Between Object Classes (COB)
CBO for a class is a count of the number of other classes to which it is coupled. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one uses methods or instance variables of another. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. Direct access to foreign instance variable has generally been identified as the worst type of coupling (Magnus Andersson Patrik Vestergren, 2004). The value of metric CBO for class Library is 2 and for class Author and Availability is 0.

RFC=|RS|
where RS, the response set of the class, is given by

\[ RS = \bigcup_{j} \{ R_{ij} \} \]

where \( M_{i} = \text{set of all methods in a class (total n)} \) and \( R_{i} = \{ R_{ij} \} = \text{set of methods called by } M_{i} \).

RFC is more sensitive measure of coupling than CBO since it considers methods instead of classes

Lake Of Cohesion in Methods (LCOM)
The LCOM is a count of the number of method pairs whose similarity is 0 minus the count of method pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter-relatedness between portions of a program. If none of the methods of a class display any instance behavior, i.e., do not use any instance variables, they have no similarity and the LCOM value for the class will be zero.

Consider a class C1 with n methods \( M_{1}, M_{2}, \ldots, M_{n} \). Let \( I_{i} = \text{set of all instance variables used by method } M_{i} \). There as n such sets \( \{ I_{1}, I_{2}, \ldots, I_{n} \} \). Let \( P = \{ (I_{i}, I_{j}) \mid I_{i} \cap I_{j} = \emptyset \} \) and \( Q = \{ (I_{i}, I_{j}) \mid I_{i} \cap I_{j} \neq \emptyset \} \). If all n sets \( \{ I_{1}, I_{2}, \ldots, I_{n} \} \) are 0 then \( P=0 \). If \( |P| > |Q| \) then

\[ \text{LCOM}=|P| - |Q|, \text{ if } |P|>|Q| \]

\[ \text{LCOM}=0, \text{ otherwise} \]
MOOD Metrics

MOOD (Metrics for Object Oriented Design) metrics refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). Each metrics is expressed as a measure where the numerator represents the actual use of one of that feature for a given design. In MOOD metrics model, two main features are used in every metrics; they are methods and attributes. Methods are used to perform operations of several kinds such as obtaining or modifying the status of object.

Method Hiding Factor (MHF)

MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible.

\[
\text{MHF} = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{A_d(C_i)} (1 - V(A_{g_j}, C_j))}{\sum_{i=1}^{TC} A_d(C_i)}
\]

Where \( A_d(C_i) \) is the number of methods declared in a class, and

\[
V(A_{g_j}, C_j) = \frac{\text{is\_visible}(A_{g_j}, C_j)}{(TC-1)}
\]

Where TC is the total number of classes, and

\[
\text{is\_visible}(A_{g_j}, C_j) = \begin{cases} 1 & \text{if } g_j \neq i \land C_j \text{ may call } A_{g_j} \\ 0 & \text{otherwise} \end{cases}
\]

Attribute Hiding Factor (AHF)

AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration. It is defined formally as

\[
\text{AHF} = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{A_d(C_i)} (1 - V(A_{a_j}, C_j))}{\sum_{i=1}^{TC} A_d(C_i)}
\]

Where \( A_d(C_i) \) is the number of attributes declared in a class, and

\[
V(A_{a_j}, C_j) = \frac{\text{is\_visible}(A_{a_j}, C_j)}{(TC-1)}
\]

Where TC is the total number of classes, and

\[
\text{is\_visible}(A_{a_j}, C_j) = \begin{cases} 1 & \text{if } A_{a_j} \text{ is visible from } C_j \\ 0 & \text{otherwise} \end{cases}
\]

Method Inheritance Factor (MIF)

MIF is defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.

It is defined as follows:

\[
\text{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}
\]

Where, \( M_d(C_i) = M_i(C_i) + M_d(C_i) \)

TC = total number of classes

\( M_d(C_i) \) = the number of methods described in a class

\( M_i(C_i) \) = the number of methods inherited in a class

Attribute Inheritance Factor (AIF)

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes. It is defined as follows:

\[
\text{AIF} = \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}
\]

Where, \( A_d(C_i) = A_i(C_i) + A_d(C_i) \)

TC = total number of classes

\( A_i(C_i) \) = number of attribute declared in a class

\( A_i(C_i) \) = number of attribute inherited in a class

AIF is 0% for class lacking inheritance

Polymorphism Factor (PF)

PF is defined as the ratio of the actual number of possible different polymorphic situation for class \( C_i \) to the maximum number of possible distinct polymorphic situations for class \( C_i \).

It is defined as below

\[
\text{PF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} [M_i(C_i) \times DC(C_i)]}
\]

Some other OO Metrics

Chen et proposed metrics are 1.CCM (Class Coupling Metric), 2.OXM (Operating Complexity Metric), 3.OACM (Operating Argument Complexity Metric), 4.ACM (Attribute Complexity Metric), 5.OCM (Operating Coupling Metric), 6.CM (Cohesion Metric), 7.CHM (Class Hierarchy of Method) and 8.RM (Reuse Metric). Metrics 1 through 3 are subjective in nature; metrics 4 through 7 involve counts of features; and metric 8 is a Boolean (0 or 1) indicator metric.

Since terminology varies among object-oriented programming languages, the authors consider the basic components of the paradigm as objects, classes,
attributes, inheritance, method, and message passing. They propose that each object oriented basic concept implies a programming behavior. They assembled metrics are:

Data Abstraction Coupling (DAC), Number of methods (NOM), Message Passing Coupling (MPC), and Number of semicolons per class (Size1), Number of methods per attributes (Size2). There is no individual breakdown of which of these metrics is significant in the prediction.

Table 2: Other Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Object-Oriented Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response for a class (RFC)</td>
<td>Class</td>
</tr>
<tr>
<td>Number of attributes per Class (NOA)</td>
<td>Class</td>
</tr>
<tr>
<td>Number of Methods per Class (NOM)</td>
<td>Class</td>
</tr>
<tr>
<td>Weighted Methods Per Class (WMC)</td>
<td>Class</td>
</tr>
<tr>
<td>Coupling between Objects (CBO)</td>
<td>Coupling</td>
</tr>
<tr>
<td>Coupling Factor (CF)</td>
<td>Coupling</td>
</tr>
<tr>
<td>Lack of Cohesion (LCOM)</td>
<td>Cohesion</td>
</tr>
<tr>
<td>Method Hiding Factor (MHF)</td>
<td>Information Hiding</td>
</tr>
<tr>
<td>Attribute Hiding Factor (AHF)</td>
<td>Information Hiding</td>
</tr>
<tr>
<td>Number of Children (NOC)</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Depth of Inheritance (DIT)</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Method Inheritance Factor (MIF)</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Attribute Inheritance Factor (AIF)</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Polymorphism Factor (PF)</td>
<td>Polymorphism</td>
</tr>
</tbody>
</table>

Source: (K.K.Aggarwal, 2006)

EMPIRICAL DATA COLLECTION

We consider seven Java projects mentioned below that are licensed as GNU open source from various domains are:

Project 1: BLACKDUCKKODERS (http://www.koders.com/): 10 versions chosen
Project 2: STRAR UML-One of the UML tool to design UML diagrams (http://www.osalt.com/staruml): 5.0 versions chosen.
Project 3: OpenOffice Draw 3.0 (http://www.openoffice.org/product/draw.html): 3.0 versions chosen
Project 4: InfraRecorder 0.50 (http://infrarecorder.org/): 0.5 versions chosen
Project 5: Gimpshop 2.2.11 (http://plasticbugs.com/?page_id=294): 2.2.11 versions Chosen

Descriptive Statistics

Table 3 gives some descriptive statistics on the above analyzed systems by using OODMAJ (18) tool.

Figure 7: Activity Diagram of OODMAJ Tool

Source: (Amjan Shaik, 2010)
Table 3: Projects Max, Min, Mean Values of Metrics

<table>
<thead>
<tr>
<th>Application</th>
<th>Blackduck Coders</th>
<th>Star.uml</th>
<th>Openoffice Draw 3.0</th>
<th>Infra Recorder</th>
<th>Gimpshp</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>1-319-21.5</td>
<td>0-235-25.75</td>
<td>0-202-26.28</td>
<td>1-87-22.81</td>
<td>0-265-15.7</td>
</tr>
<tr>
<td>LOC</td>
<td>3-2275-151.4</td>
<td>2-2539-166.19</td>
<td>3-872-125.12</td>
<td>4-481-141.2</td>
<td>0-1253-81.2</td>
</tr>
<tr>
<td>TCC</td>
<td>0-101-12.11</td>
<td>0-100-12.67</td>
<td>0-100-45.53</td>
<td>0-100-14.17</td>
<td>0-99-8.2</td>
</tr>
<tr>
<td>RFC</td>
<td>1-627-131.28</td>
<td>0-435-45.74</td>
<td>0-117-48.3</td>
<td>1-88-37.2</td>
<td>1-356-27.9</td>
</tr>
<tr>
<td>NOO</td>
<td>0-238-8.2</td>
<td>0-48-7.82</td>
<td>0-91-7.1</td>
<td>0-24-7.36</td>
<td>0-26-4.2</td>
</tr>
<tr>
<td>MIC</td>
<td>0-20-1.98</td>
<td>0-26-3.92</td>
<td>0-43-3.53</td>
<td>0-22-8.11</td>
<td>0-15-2.1</td>
</tr>
<tr>
<td>DIT</td>
<td>0-8-2.39</td>
<td>0-5-1.246</td>
<td>0-3-1.56</td>
<td>0-4-1.11</td>
<td>0-5-1.3</td>
</tr>
<tr>
<td>LCOM</td>
<td>0-32248-121.4</td>
<td>0-674-37.56</td>
<td>0-3512-62.34</td>
<td>0-76-11.5</td>
<td>0-182-11.67</td>
</tr>
<tr>
<td>CBO</td>
<td>0-126-12.6</td>
<td>0-29-5.11</td>
<td>0-61-8.93</td>
<td>0-31-8.2</td>
<td>0-23-4.69</td>
</tr>
</tbody>
</table>

Figure 8: A Bar Chart Representation of Averages

CONCLUSION
This study focuses on object-oriented design and quality metrics that can be used to measure the quality of an object-oriented design. Our cram enhances prior empirical literature on OOD Metrics by providing a new set of results in Object Oriented Systems. To our knowledge there is a little prior research on the effects of interaction between the Design metrics. Object oriented quality metrics lead to a number of inherent benefits that provide advantages at both the management and technical level. In this paper we first defined the metrics and then explained using empirical applications. We analyzed several java Open Source Projects. The results obtained from the OODMAJ tool were accurate when verified with manual testing.

ACKNOWLEDGEMENTS
The authors would like to thank Ellenki College of Engineering and Technology’s Research and Development Center for their cooperation and support in data collection.

REFERENCES


Jaana Lindroos (2004), Code and Design Metrics for Object-Oriented Systems, 1st December.


Ms Puneet Jai kaur, Ms Amandeep Verma, Mr. Simrandeep (2007) Thapar3, “Software Quality Metrics for Object-Oriented Environments, Proceedings of National Conference on Challenges & Opportunities in


