Metrics for Object Oriented Design Software Systems: A Survey


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
In this paper, we present obtainable and new Software metrics useful in the different phase of the Object-Oriented Software Development Life Cycle. Metrics are used by the software industry to itemize the development, operation and maintenance of software. The practice of applying software metrics to a software process and to a software product is a complex task that requires study and restraint, which brings knowledge of the status of the process and / or product of software in regards to the goals to achieve. In this paper, we have presented metrics for Object Oriented Software Systems. They provide a basis for measuring all characteristics.

Keywords: metrics, object oriented design, software, quality.

INTRODUCTION
An effort is ready to investigation of Software Metrics applicable to measure the Object Oriented Software Systems. Object Oriented Software is vitally different from software developed using unadventurous methods. It is therefore reasonable that, the metrics for Object Oriented Software Systems must be oriented to the characteristics that distinguish Object Oriented from predictable software. The metrics for Object Oriented Software Systems focus on measurement that can be applied to the class and the design characteristics viz., localization, encapsulation, inheritance, information hiding, polymorphism, messaging and object abstraction techniques, that make the class unique. A number of metrics have been proposed for computer software, but not all provide practical support to the software designers, developers and managers. (William Frakes, and C. Terry, 1996). The design of complex software-based systems often proceeds with virtually no measurement. To be useful in a real world context, software metric must be simple and computable, persuasive, consistent, and objective. It should be independent of programming language and provide effective feedback to the software engineers. Most of design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence. Software Metrics makes it possible for software engineers to measure and predict software processes, and necessary resources for a project and work products relevant for a software development effort. A software measure provides software engineers with a means of quantifying the assessment of a software product. Software process metrics are used for strategic purposes. Software project measures are tactical and occur during estimation (AmjanShaik, 2010). A software engineer collects measures and develops metrics so that indicators can be obtained. An indicator is a metric or combination of metrics that provides insight into the software process, a software project or the product itself, and a metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute. A key element of any engineering process is measurement. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment and project control. Measurement can be applied to the software process with the intent of improving it on a continuous basis. Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved. There are four reasons for measuring software processes, products and resources: characterization, evaluation, prediction and improvement. A software engineer collects measures and develops metrics so that an indicator will be obtained. Metrics provide the insight necessary to create effective analysis and design models, solid code, and thorough tests. Software metrics provide a quantitative way to assess the quality of internal product attributes, there-by enabling the software engineers to assess quality before the product is built.
A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process. One of the biggest problems of software development is the management of software quality, both in terms of defects and easiness of maintenance. Software production and evolution is intangible. This nature has caused problems for software developers since there is no sense feedback. The obvious idea is for future development to learn from past experience i.e. avoiding practices leading to poor software quality and emphasizing those increasing it. Static analysis is a way to provide feedback in software Development (Chidamber, S.R.; Kemerer, 1994). 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 require also 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. Natural target for software measurements has always been the actual source code. However, compared to traditional software, Object Orientation changes the structure of source code. One of the main reasons for this is the concept of a class which at least in the most dogmatic sense is equivalent to a module. Hence traditional definition of a module and its dependencies is not sufficient in object world. Other architectural differences between traditional and Object Oriented implementations include inheritance and information hiding or encapsulation. These both are features that simply do not exist in traditional environments. Because of all this, it is easy to understand why some of the traditional metrics can not be used in Object Oriented environments in a meaningful way. On the other hand, some metrics can be transformed to serve their purpose in new environment. The software development process is no doubt a complicated one. The end product follows a chain of analysis, design, development and testing process. At each stage, it is important to follow a well-defined methodology to ensure a quality end product. For large scale projects, each stage in the whole process is a challenge. In this context, the software design and coding metrics play an important role in ensuring the desired quality. In this paper, we would examine couple of important Object Oriented Metrics.

Synopsis of Metrics
Necessity of Software Metrics
In Object Oriented software development process, the system is viewed as collection of objects. The functionality of the application is achieved by interaction among these objects in terms of messages. Whenever, one object depends on another object to do certain functionality, there is a relationship between those two classes. In the light of modern day .Net/J2EE/J2ME/Lotus Notes like development, it is recommended than application software is split into multiple layers. This ensures "separation of concern". With this, we have objects from one layer talking to the objects of another layer. In order to achieve perfect "separation of concern", objects should rely on the interfaces and contracts offered by another object without relying on any underlying implementation details. For example, the application layer depends on Database Access Layer to access data. The Application layer however should never need to know how the data is physically accessed and what the underlying data store is. This is called abstraction. Thus, correct level of abstraction helps build a flexible and scalable application. It is not an easy job to reach the correct level of abstraction and the correct relationship between classes. It is better if we can detect any possible faults at an early stage of the design process, so that the design can be corrected in accordance. Object Oriented Design metrics can be a very helpful measuring technique to evaluate the design stability. Also, given a correct abstraction of layers and appropriate relationship between the classes, there are still chances that the coding process might introduce vulnerability, which is not of defective coding as such but more to do with the internal structure of the code. At this stage also Object Oriented Metrics can be of help to identify, if we need to pay further attention to any of the code to make it more maintainable. This is what the role of software design and development metrics are. They are used to ensure a better quality and maintainability as a whole. It is also observed that following these metrics make writing test cases easier. Any application that can be tested easily is easier to maintain and debug. Despite the fact that the Software Engineering field does not have a unified set of metrics that the community has agreed to use, it is advised to use them. Often during the software process, the members of the development team do not know if what they are doing is correct and they need a guide that could help them orientate further improvement and to objectively know if the improvement is being achieved. Software metrics are tools that help to track software improvement. Most large companies dedicated to develop software, use metrics in a consistent way. Many companies have created their own standards of software measurement, so, the way those metrics are applied usually varies form one company to another one. Nevertheless, as they are used in a consistent way through different projects, the software groups get many benefits from them. What to measure in regards of software process or product depends on the nature of the project, but in all cases, the customer satisfaction is the goal and measures.
should be taken to achieve that goal not only at
delivery, but through the entire development process.

**Definition of Software Metrics**

It is important to further define the term software
metrics as used in this module. Essentially, software
metrics deals with the measurement of the software
product and the process by which it is developed. In
this discussion, the software product should be viewed
as an abstract object that evolves from an initial
statement of need to a finished software system,
including source and object code and the various forms
of documentation produced during development.
Ordinarily, these measurements of the software
process and product are studied and developed for use
in modeling the software development process. These
metrics are then used to estimate or predict product
costs and schedules and to measure productivity and
product quality. Information gained from the metrics
and the model can then be used in the management and
control of the development process, leading, to
improved results. Good metrics should facilitate the
development of models that are capable of predicting
process or product parameters, not just describing
them.

Software Metrics, as the term is most commonly used
today, concerns itself almost exclusively with the first
and last of the above characteristics, *i.e.*, functionality
and economy. Performance is certainly important, but
it is not generally included in discussions of software
metrics, except regarding whether the product meets
specific performance requirements for that product.
The evaluation of performance is often treated
extensively by those engaged in performance
evaluation studies, but these are not generally included
in what is referred to as software metrics. It is possible
that, in the future, the scope of software metrics may
be expanded to include performance evaluation, or
both activities may be considered part of a larger area
that might be called software measurement.

**Classification of Software Metrics**

Software metrics can be classified into three
categories: product metrics, process metrics, and
project metrics (Victor Basili and Walcelio L Melo,
1996). Project metrics are those that describe the
characteristics of the product such as size, complexity,
design features, performance, and quality level.
Process metrics are those that can be used for
improving the software development and maintenance
process. Examples include the effectiveness of defect
removal during development, the pattern of testing
defect arrival, and the response time of the process.
Project metrics are those that describe the project
characteristics and execution. Examples include the
number of software developers, the staffing pattern
over the life cycle of the software, cost, schedule, and
productivity (ETH Zurich, 2006).

For product metrics, the size of the product measured
in lines of code (LOC) is an objective measure, for
which any informed observer, working from the same
definition of LOC, should obtain the same measured
value for a given program. An example of a subjective
product metric is the classification of the software as
“organic,” “semi-detached,” or “embedded,” as
required in the COCOMO cost estimation model.
Although most programs might be easy to classify,
those on the borderline between categories might
reasonably be classified in different ways by different
knowledgeable observers. For process metrics,
development time is an example of an objective
measure, and level of programmer experience is likely
to be a subjective measure. Another way in which
metrics can be categorized is as primitive metrics or
computed metrics. Primitive metrics are those that can
be directly observed, such as the program size (in
LOC), number of defects observed in unit testing, or
total development time for the project. Computed
metrics are those that cannot be directly observed but
are computed in some manner from other metrics.
Examples of computed metrics are those commonly
used for productivity, such as LOC produced per
person-month (LOC/person-month), or for product
quality, such as the number of defects per thousand
lines of code (defects/KLOC). Computed metrics are
combinations of other metric values and often more
valuable in understanding or evaluating the software
process than are simple metrics.

If the metric values are to be used in mathematical
equations designed to represent a model of the
software process, metrics associated with a ratio scale
may be preferred, since ratio scale data allow most
mathematical operations to be meaningfully applied.
However, it seems clear that the values of many
parameters essential to the software development
process cannot be associated with a ratio scale, given
our present state of knowledge.

Measurement enables managers and practitioners to
better understand the software engineering process and
the product that it produces (Rachel et, al. 1998).
Measures are used to better understand the attributes of
the models that we create and to assess the quality of
the engineered products or systems that we build.
Conventional software emphasizes function as a
localization mechanism, software metrics have focused
on the internal structure or complexity of functions or
the manner in which functions connect to one another
(Stephen Kan, 2002).

**Logic Structure Metrics**

Logic structure metrics attempt to quantify a feature
that is the most profound in any code:
**Reachability Metric**
A reachability software measure was proposed by Schneidewind and Hoffman in 1979. It attempts to characterize the complexity of a flowchart by qualifying a number of unique ways one can reach from each node of the resulting graph.

\[ R = \frac{\text{Total number of paths}}{\text{Number of nodes}} \]

**Nesting Levels Metric**
The nesting levels measure is useful when it comes to expressing the depth of nesting of loops, one inside another. The nesting levels measure is computed by assigning a level of depth to each executable statement in a code and averaging these values. The average nesting level, \( NL = \text{Sum of all nesting levels} / \text{Number of statements} \).

**Entropy Based Software Metrics**
The use of the entropy measure in the framework of software engineering calls for the attachment of meaning to the basic symbols. The entropy of a software module is computed as follows:

\[ H = - \sum_{i=1}^{j} n_i \log_2 \left( \frac{n_i}{n} \right) \]

Where \( j \) denotes the number of classes of the operators, \( n_i \) is frequency of occurrence of operator, and \( n \) stands for the number of all occurrences of the operators.

**Software Quality Metrics**
McCall and his colleagues have proposed a useful categorization of factors that affect software quality (Stephen H. Kan, 2002). A quality metric that provides benefit at both the project and process level is Defect Removal Efficiency (DRE) (Rachel et al. 1998).

**Defect Removal Efficiency**

\[ \text{DRE} = \frac{E}{E + D} \]

Where \( E \) is the number of errors found before delivery of the software to the end-user and \( D \) is the number of defects found after delivery.

**Reliability Metrics**
Reliability is the most important dynamic characteristic of almost all software systems. Software reliability is a function of the number of failures experienced by a particular user of that software. Reliability metrics are all based around the probability of a system failure (S. Henry and W. Li, 1993).

- Mean time to failure (MTTF)
- Mean time to repair (MTTR)
- Mean time between failure
- MTBF = MTTF + MTTR
- Probability of failure on demand (POFOD)
- Rate of failure occurrence (ROCOF)
- Availability

\[ \text{AVAIL} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\% \]

**Metrics for Small Organizations**
Kautz describes a typical scenario that occurs when metrics programs are suggested for small software organizations (Rachel et al. 1998). A small organization might select the following set of easily collected measures:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete, \( t_{\text{queue}} \)
- Effort (person-hours) to perform the evaluation, \( W_{\text{eval}} \)
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personal, \( t_{\text{eval}} \)
- Effort (person-hours) required to make the change, \( W_{\text{change}} \)
- Time required (hours or days) to make the change, \( t_{\text{change}} \)
- Errors uncovered during work to make change, \( E_{\text{change}} \)
- Defects uncovered after change is released to the customer base, \( D_{\text{change}} \)

**Metrics for Object-Oriented Systems**
A software measurement is a technique or method that applies software measures to a class of software engineering objects to achieve a predefined goal. Object Oriented System with their distinct design objectives and goals calls for a completely new suite of software measures (Banker et al. 1993). In a nutshell, the Object-Oriented Methodology centers on objects. Objects stand in sharp contrast to the function-oriented view of software development and analysis. Berard defines five characteristics that lead to specialized metrics-localization, encapsulation, information hiding, inheritance, and object abstraction techniques. The use of metrics for Object Oriented systems has progressed much more slowly than the use of other Object Oriented methods (Rachel et al. 1998).

Whitmire describes nine distinct and measurable characteristics of an Object Oriented design: Size, Complexity, Coupling, Sufficiency, Completeness, Cohesion, Primitiveness, Similarity, and Volatility (Rachel et al. 1998).

The primary objectives for Object Oriented metrics are no different than those for metrics derived for conventional software (Rachel et al. 1998).

- To better understand the quality of the product
- To assess the effectiveness of the process
- To improve the quality of work performed at a project level

**Lorenz and Kidd Metrics**
Lorenz and Kidd divide class-based metrics into four broad categories: size, inheritance, internals, and externals (Lionel C. Briand, John W. Daly, and Jurgen K. Wust, 1999).

- Class Size (CS)
- Number of Operations Overridden by a Subclass (NOO)
- Number of Operations Added by a Subclass (NOA)
- Specialization Index (SI)
Operation Oriented Metrics
Operation Oriented metrics focus on the size and complexity of individual operations. Three simple metrics, proposed by Lorenz and Kidd are (Lionel et al, 1999):

- Average Operation size ($O_{avg}$)
- Operation Complexity (OC)
- Average number of parameters per operation ($P_{avg}$)

Metrics for Object Oriented Testing
Binder suggests a broad array of design metrics that have a direct influence on the testability of an Object Oriented system. The metrics are organized into categories that reflect important design characteristics (Lionel et al, 1999).

- Lack of cohesion in methods (LCOM)
- Percent public and protected (PAP)
- Public access to data members (PAD)
- Number of Root classes (NOR)
- Fan in (FIN)
- Number of children (NOC)
- Depth of the inheritance tree (DIT)

Metrics for Object Oriented Projects
A key issue that faces a project manager during planning is an estimate of the implemented size of the software. Size is directly proportional to effort and duration. The following Object Oriented metrics can provide insight into software size (Rachel et al, 1998). Lorenz and Kidd suggest the following set of project metrics.

- Number of scenario scripts (NSS)
- Number of Key Classes (NKC)
- Number of Subsystems (NSUB)
- Number of support classes (NSC)
- Average number of support classes per key class

MOOD Metrics Suite
Harrison, Counsell, and Nithi propose a set of metrics for Object Oriented design that provides quantitative indicators for Object Oriented design characteristics. A set of metrics for Object Oriented design called the MOOD metrics (Rachel et al, 1998). There are six MOOD metrics.

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Coupling Factor (CF)
- Polymorphism Factor (PF)

Brito and Carapuca Inheritance Metrics
Brito and Carapuca have proposed candidate inheritance metrics and have taken into account all relevant metric proposals (Gursaran and Gurdev Roy, 2001):

- Total progeny Count (TPC)
- Total Parent Count (TPAC)
- Total Ascendancy Count (TAC)

Coupling Metrics
Coupling measures have been proposed by Chidamber and Kemerer, Lii and Henry, Martin, Lee et. al. and Briand et.al (S. Henry and W. Li,1993) (Chidamber, S.R.; Kemerer, 1994).

- Coupling between Objects (CBO)
- Response for Class (RFC)
- Message Passing Coupling (MPC)
- Data abstraction Coupling (DAC)
- Ce and Ca (cfferent and afferent coupling)
- Coupling factor (COF)
- Information-flow-based coupling (ICP)
- Inheritance-based coupling As ICP (IH-ICP)
- Non Inheritance-based coupling As ICP (NIH-ICP)
- Polymorphic ally invoked methods (PIM)
- Export coupling version of PIM (PIM_EC)
- Average number of parameters per method ($NP_{avg}$)

Design Metrics
The structural and behavioral design properties of classes, objects, and their relations are evaluated using a suite of Object-Oriented design metrics (ETH Zurich, 2006).

- Design Size in Classes (DSC)
- Number of Hierarchies (NOH)
- Average Number of Ancestors (ANA)
- Data Access Metric (DAM)
- Direct Class Coupling (DCC)
- Cohesion Among Methods of Class (CAM)
- Measure of Aggregation (MOA)
- Measure of Functional Abstraction (MFA)
- Number of Polymorphic Methods (NOP)
- Class Interface Size (CIS)
- Number of Methods (NOM)

Cohesion Metrics
Cohesion is the property that specifies how tightly bound the elements of a module are. It is desirable to have each class highly cohesive-all the elements are together to support a well-defined abstraction. A class is naturally a cohesive entity as all its data and operations are packaged together for a purpose. Cohesion can be modeled by the degree of similarity between methods of the same class (Chidamber, S.R.; Kemerer, 1994).

- Lack of cohesion in methods (LCOM1, LCOM2, LCOM3, LCOM4, LCOM5)
- Tight class cohesion (TCC)
- Loose class cohesion (LCC)
- Connectivity (Co)
- A variation on LCOM5 (Coh)
Estimation of software size is a crucial activity among software and the process by which it is developed. Size-oriented software metrics are direct measures of size. Object Oriented system reuse attributes include client perspective, and system perspective. Measurable Object Oriented system reuse attributes include percentages of code and classes that are new or derived and specific client/server relationships. Bieman identifies three perspectives from which to view reuse in the Object Oriented environment: server perspective, client perspective, and system perspective.

**Reuse Metrics**

Bieman (1992) and Bieman and Karunanithi (1993) have proposed reuse metrics for Object Oriented systems. Bieman identifies three perspectives from which to view reuse: server, client, and system. There are three perspectives from which to view reuse in the Object Oriented environment: server perspective, client perspective, and system perspective. Measurable Object Oriented system reuse attributes include percentages of code and classes that are new or derived and specific client/server relationships. (William Frakes, and C. Terry, 1996)

A general measure of reuse in Object Oriented systems, termed reuse leverage is defined as (Lionel C. Briand, John W. Daly, and Jurgen K. Wust, 1999).

\[
R_{\text{lev}} = \frac{\text{OBJ}_\text{reused}}{\text{OBJ}_\text{built}}
\]

Where \( \text{OBJ}_\text{reused} \) is the number of objects reused in a system \( \text{OBJ}_\text{built} \) - is the number of objects built for a system.

**Inheritance Metrics**

Inheritance is a mechanism that enables the responsibilities of one object to be propagated to other objects. Inheritance occurs throughout all levels of a class hierarchy. Because inheritance is a pivotal characteristic in many Object Oriented systems, many Object Oriented metrics focus on it.

- Depth of Inheritance Tree (DIT)
- Average inheritance depth of a class (AID)
- Class-to-leaf depth (CLD)
- Number of children (NOC)
- Number of Parent (NOP)
- Number of Descendent (NOD)
- Number of Ancestor (NOA)
- Number of Methods overridden (NMO)
- Number of Methods added (NMA)
- Specialization Index (SIX)

The SIX is defined as

\[
\text{SIX} = \frac{\text{NMO} \times \text{DIT}}{\text{NMO} + \text{NMA} + \text{NMINH}}
\]

**Size Metrics**

Size-oriented software metrics are direct measures of software and the process by which it is developed. Estimation of software size is a crucial activity among the tasks of software management.

- Number of methods implemented in a class (non-inherited or overriding) (NMIMP)
- Number of inherited methods in a class , not overridden(NMINH)
- Number of methods in a class (both inherited and non-inherited) (NM)
- Number of attributes in a class (excluding inherited ones) (NAIMP)
- Number of attributes in a class (both inherited and non-inherited) (NA)

- Number of parameters (NUMPAR)

**CK Metrics Suit**

One of the most widely referenced sets of Object Oriented software metrics has been proposed by Chidamber and Kemerer (CK). The authors have proposed six class-based design metrics for Object Oriented Systems (Chidamber and Kemerer, 1994).

The CK metrics suite defines class-oriented software metrics that focus on the class and the class hierarchy. Chidamber and Kemerer worked in collaboration with experienced software engineers and collected data from two different commercial projects. They made some interesting and practically useful observations on when analyzing their results.

**Weighted Method per Class (WMC)**

This is defined as the sum of the complexity of all the methods defined in a class. If all the method complexities are reduced to unity, then WMC becomes equal to the number of methods. (AmjanShaik, 2010).

\[
\text{WMC} = \text{sum of cyclomatic complexities of all the methods.}
\]

**Response for a Class (RFC)**

The RFC is defined as the total number of methods that can be executed in response to a message to a class. This count includes all the methods available in the whole class hierarchy. If a class is capable of producing a vast number of outcomes in response to a message, it makes testing more difficult for all the possible outcomes. (AmjanShaik, 2010).

**Lack of Cohesion in Methods (LCOM)**

Cohesion is the degree to which the methods in a class are related to one another and work together to provide a set of behaviors. LCOM measures the degree of similarity of methods by data inputs or the instance variables of the class. To elaborate more, there are two main types of LCOM calculation methods available. LCOM measures the dissimilarity of methods in a class (Rachel et al., 1998).

Let C1 is class with methods \( f_1, f_2, \ldots f_n \) and a set of instance variables \( \{I_i\} \). Then for any method \( f_i \) we can define the partitioned set and LCOM.

\[
\text{P} = \{(I_i, I_j) | I_i \cap I_j = \emptyset\} \quad \text{and} \quad \text{Q} = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}
\]

then LCOM = \( |P| - |Q| \), if \( |P| > |Q| \) else LCOM=0

Henderson-Sellers revise the LCOM metric to normalize it for the number of methods and variables that represent in the class (Victor Basili and Walcelio L Melo, 1996):

\[
\text{LCOM} = \frac{1}{a} \sum_{j=1}^{a} \mu(A_j) - m
\]

Where the number of methods is \( m \) and the number of instance variables (attributes) a set of \( \{A_i\} (j=1, 2, \ldots a) \). Let \( \mu (A_j) \) be the number of methods which access each datum. With this new measure the metric is simplified and normalized.
**LCOM1**
Take each pair of methods in the class. If they access different sets of instance variables, increase P by one. If they share at least one variable, increase Q by one.

\[
LCOM1 = P - Q \quad \text{if} \quad P > Q
\]

\[
LCOM1 = 0 \quad \text{otherwise}
\]

LCOM1 = 0 indicates a cohesive class.
LCOM1 > 0 indicates that the class is not quite cohesive and may need refactoring into two or more classes. Classes with a high LCOM1 can be fault-prone. (Chidamber and Kemerer, 1994).

\[m = \text{number of procedures (methods) in class}
\]

\[A = \text{number of variables (attributes) in class}
\]

\[mA = \text{number of methods that access a variable (attribute)}
\]

\[
\text{sum(mA)} = \text{sum of mA over attributes of a class}
\]

**LCOM2**
LCOM2 is counted as the percentage of methods that do not access a specific attribute averaged over all attributes in the class.

\[
LCOM2 = 1 - \frac{\text{sum (mA)}}{(mA \cdot a)}
\]

If the number of methods or variables is zero, LCOM2 is undefined and displayed as zero.

**LCOM3**

\[
LCOM3 = \frac{m - \text{sum (mA)}}{a} / (m-1)
\]

LCOM3 varies between 0 and 2.

**LCOM4**

LCOM4 is the lack of cohesion metric we recommend for Visual Basic programs. LCOM4 measures the number of "connected components" in a class. A connected component is a set of related methods (and class-level variables).

\[LCOM4 = 1 \quad \text{indicates a cohesive class, which is the "good" class.}
\]

\[LCOM4 = 2 \quad \text{indicates a problem. The class should be split into so many smaller classes.}
\]

\[LCOM4 = 0 \quad \text{happens when there are no methods in a class. This is also a "bad" class. Methods access its own variables, LCOM3 varies between 0 (high cohesion) and 1 (no cohesion). A LCOM3 value of 1 indicates high lack of cohesion. When LCOM3=0, each method accesses all variables. This indicates the highest possible cohesion.}
\]

**LCOM5**
It is specified from the perspective of the number of functions accessing each attribute.

\[\text{Co (Connectivity). Let } V \text{ be the vertices of graph } G \text{ from LCOM4, and } E \text{ its edges. Then}
\]

\[
\text{Coh} = \sum_{1 \leq j \leq m} \text{a} \mu(A_i)
\]


**Coupling between Object Classes (CBO)**
Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are ignored. If a method call results in calling (using) other classes, all the classes to which the call can go are included in the coupled count (Chidamber and Kemerer, 1994).

**Depth of Inheritance Tree (DIT)**
Depth of Inheritance is the maximum length from a given class to the root of the inheritance tree. In Java, as all the classes inherit from Object class, the minimum DIT in Java is 1. It is a measure of the depth of the class hierarchy. The higher the value of DIT, child classes inherits more number of methods from the base classes. In such situations, it becomes too difficult to evolve the base classes and child classes. Thus, it is important to keep a low value of DIT in design.

**Number of Children (NOC)**
Number of Children is the immediate number of subclasses to a base class. If we have a very large number of children to a base class, it might be a candidate for refactoring to create a more sustainable and maintainable hierarchy.

**Method Invocation Coupling (MIC)**
It is defined as the relative number of classes that receive messages from a particular class. Modularity of classes is inversely proportional to coupling between those classes J.M. Bieman, B.K. Kang, 1995). Reduced coupling between classes improve the modularity. Low coupling between classes in Object Oriented System is tendency of current software engineering practices that facilitates the easy evolution (S. Henry and W. Li, 1993). Besides, Class coupling at higher level tends to ripple-effect (Higher changes required for version updates) that addresses the maintenance issues. The ability to analyze of a highly coupled class is difficult since analysis of the coupled classes need to be done simultaneously. Fault proneness is correlated with few of existing coupling metrics and their relationship (Victor Basili and Walcario L Melo, 1996).

**Tight Class Cohesion (TCC)**
The Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class (Gursaran and Gurdev Roy, 2001). In the scope of a class two methods are said to be connected if and only if they are using common instance of that class. Let N is number of methods in a class, NP is total possible connections and NDC is number of direct connections. The NP can be measured as follow

\[
NP = N \times (N-1) / 2
\]

Then TCC can be measured as NDC / NP
**Number of Operations (NOO)**
Total number of functions in a class represented by NOO, Degree of reusability and cohesion is inversely proportional to NOO.

**Loose Class Cohesion (LCC)**
Let NIC be the number of direct or indirect connections between public methods. Then LCC is defined as the relative number of directly or indirectly connected public methods. LCC = NIC / NP. CBMC (R.D. Banker, S.M. Datar, C.F. Kemerer, and D. Zweig, 1993) connectivity factor multiplied by structure factor.

**CONCLUSION**
This manuscript contributes to an increased understanding of the state-of-the-art. A mechanism is provided for comparing measures and their potential use, integrating existing measures, which examine the same concepts in different ways, and facilitating more rigorous decision-making, regarding the explanation of new measures and the selection of existing measures for a specific goal of measurement. The increasing significance being placed on software measurement has led to an increased amount of research on developing new software measures. In this paper, we have presented metrics for Object Oriented Software Systems. They provide a basis for measuring all characteristics. Traditional software product metrics that evaluate product characteristics such as size, complexity, performance, and quality must be changed to rely on some fundamentally different notions such as encapsulation, inheritance, and polymorphism and messaging which are inherent in Object Orientation. If the metrics are properly defined, we can avoid problems that will be more exclusive to correct during the latter phases of Object Oriented Software Development. It is useful to validate a value such as the effort, with different metrics to see how close they are. This paper helps researchers and practitioners better understanding and select software metrics suitable for their purposes.

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

**REFERENCES**


Martin Hitz and Behzad Montazeri (1996) “Chidamber and Kemerer’s Metrics Suite: A


