CHAPTER 2
SURVEY AND REVIEW OF LITERATURE

2.1 Introduction
2.2 Concepts and Terminology of OOSD Approach
   2.2.1 Exploring Key Terms of OOSD Approach
      2.2.1.1 Classes and Objects
      2.2.1.2 Encapsulation
      2.2.1.3 Abstraction
      2.2.1.4 Inheritance
      2.2.1.5 Polymorphism
   2.2.2 Quality of Good Object Oriented Software Systems
   2.2.3 Benefits of OOSD Approach
2.3 Survey of Software Measurement and Metrics
   2.3.1 Categorization of Metrics
   2.3.2 Issues Related to Metrics
   2.3.3 Guidelines for Developing Metrics
   2.3.4 Traditional Non-Object Oriented Metrics
2.4 Review of Existing Object Oriented Software Metrics
   2.4.1 Taxonomy of Object Oriented Software Metrics
   2.4.2 Existing Object Oriented Metrics and its Validations
   2.4.3 Specifications of Object Oriented Metrics
2.5 Summary

2.1 INTRODUCTION

Many software development approaches have evolved over a period of time. The primarily reason for that is the increasing complexity and rapidly growing cost of software maintenance. Among them object oriented software development approach is
one of the most effective approach which has most vital influence on the software
development. This approach of software development comes into existence to overcome
the problems arises in procedure oriented approach of software development. Procedure
oriented approach of software development is not capable to fulfill the demands of
today’s high quality software systems. Developing software using procedure oriented
approach is expensive and time-intensive process. Object oriented approach is widely
accepted by software industries because it is more cost-effective and faster way to
develop software systems. Object oriented technology cuts development time and
overhead and enable software engineers to develop more flexible and easily maintainable
applications. Object oriented approach proven powerful means to control the complexity
inherent in the development process of large and distributed applications.

[Khan08] defines object oriented software methodology is a way of structuring software
as a collection of discrete objects reflecting real-world entities and mapping them into
design constructs to represent relationships and functionalities efficiently. It reflects a
natural view of the domain and handles inherent complexity in better way. Object
Oriented Paradigm (OOP) provides mechanisms to package data and process together in
order to represent real word concepts [Noran03]. Real world concepts include - Entity,
Relationships and Occurrences. Entity consists of structural and behavioral
characteristics. Relationships exist among entities via structural characteristics.
Occurrences means behavioral characteristics shared among entities. OOP supports 1)
Better modeling techniques such as Object Modeling Technique (OMT) and Unified
Modeling Language (UML) which provides easy communication between stakeholders
involved in problem solving. 2) Better facility of reusing assets. 3) Better control on
change management and complexity.

The OOP focuses on the behavioral and structural characteristics of entities as a complete
unit. It is concept-centric approach that focuses on all types of features that constitute
any given concept. This paradigm encompasses and supported the following pillars:
abstraction, encapsulation, inheritance and polymorphism [Sinan98]. There are number of
advantages of using OOP; some of these include simple maintenance, an advanced
analysis of complicated programs and reusability.
The foundation of OOP is the fact that it will place an emphasis on objects and classes. There are number of programming languages that use OOP, some of these are C++, Java and C#. In object oriented programming, a problem is broken down into a number of units. These units are called objects. Object oriented software is collection of objects which interacts with each other in a number of different ways.

Traditional metrics are not fully suitable for measuring attributes of OOSD therefore, several new metrics have been proposed by the many researchers for OOSD. In this research work extensive literature survey is carried out in the area of object oriented software measurement by focusing on purpose of metrics, approach used to define metrics, definition of metrics and validation studies. In addition various methods of formal specification of metrics are also investigated. The survey is presented in this chapter in section 2.4. Before presenting the survey elementary concepts and terminology of OOSD is given in section 2.2 which presents fundamental concepts of OOSD approach, quality of good object oriented software systems and benefits of using OOSD approach. Section 2.3 formulates the general background on software measurement and metrics. This section presents classification of metrics, guidelines for developing metrics and traditional metrics that exists in literature before OOSD approach comes in practice.

2.2 CONCEPTS AND TERMINOLOGY OF OOSD APPROACH

2.2.1 Exploring Key Terms of OOSD Approach

Object oriented software is developed in the form of classes and objects which provide benefits of encapsulation, abstraction, inheritance and polymorphism. Class, Object, Encapsulation, Abstraction, Inheritance and Polymorphism are key terms of OOSD which are described as follows:

2.2.1.1 Classes and Objects

A class is a unit that holds data and functions which will carry out operations on the data. Class is a description of objects with common attributes, operation implementations, semantics, associations and interactions [Sinan98]. Class is a static definition that is used to create objects. It is also known as template for group of objects. It works like a factory for objects. It is only concept which helps us to understand the real world entities. For
example *Book* is a class having attributes title, author, publisher and price while print and search are its operations.

Each member of class has an access modifier. Generally there are three types of access modifiers - private, public and protected. A member that is public can be accessed by objects outside the class and can be inherited. A member that is defined as private can be accessed by members of class only; it can't be accessed outside the class. In addition to this, it can't be inherited. While a member who is protected can be inherited, it can't be accessed by objects which reside outside of the class hierarchy.

Object is an instance of class or any run time entity. An object is an abstraction of something in the real world or our imagination comprising its attributes and the activities which can be performed by or on it. Each class can have possibly infinite number of individual objects. It encapsulates structural characteristics known as attributes and behavioral characteristics known as operations. Attributes of an object determine its possible states and operations determine the possible behavior of an object, called upon as a response to receiving a message. Each object has its own inherent identity. Object can be concrete such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocessing operating system [Rambaugh91].

### 2.2.1.2 Encapsulation

Encapsulation is mechanism of packaging of data and code to manipulate data into a single unit. It provides a layer of security to protect data from external interference and misuse. It restricts the data safely inside the class, where it can be accessed only by trusted methods of the class. Encapsulation is a concept which promotes modularity and it is also important for hiding information. Encapsulation makes each object like a "black box" which provides a known set of outputs when provided with a known set of inputs. Therefore, to utilize an object, a programmer needs only to understand what inputs an object expects and what it will return. Any changes that may be made to an object are of no concern to the user of the object, so long as the required inputs and outputs do not change. Advantage of encapsulation is that it builds the system which can be extended easily. Every class of the system can changed independently without any impact on the other class. Encapsulation is also particularly important in case of reuse. Ideally changes
to classes, whether to the structure of the data or the implementation of the methods, should also not have adverse effects on other applications that reuse these classes, but sometimes this is more difficult to achieve [PostOl]. Another advantage of encapsulation is low coupling, for good systems low coupling is desirable. Encapsulation results in lesser dependencies of one object on other objects, therefore, provides low coupling.

2.2.1.3 Abstraction
Abstraction is one of the essential elements of object oriented programming. According to [Sinan98] abstraction involves the formulation of representations by focusing on similarities and differences among a set of entities to extract intrinsic essential characteristics (relevant common features) and avoid extrinsic incidental characteristics (irrelevant distinguishing features) in order to define a single representation having those characteristics that are relevant to defining every element in the set. In other words abstraction involves focusing on relevant issues while avoiding the incidental things, so as to gain the thorough understanding of the subject [Noran03]. In object oriented environment abstraction is achieved by separating interface with implementation.

Abstraction differs from encapsulation because abstraction focuses upon observable behavior of an object whereas encapsulation focuses on implementation that gives this behavior.

There are two types of abstractions – Function abstraction and Data abstraction. Function abstraction means we can call any function without focusing on its internal characteristics (logic). Data abstraction helps to tie data and functions together, which defines a new data type. In object oriented software classes use the concept of abstraction to define a new type, therefore, a class is also known as Abstract Data Type (ADT).

2.2.1.4 Inheritance
Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship [Rambaugh91]. It allows the extension and reuse of existing classes to create new classes without having to repeat or rewrite the code from scratch.

The existing class which is being reused is called base/super class and class which is inherited from existing class is known as derived/sub class. The derived class inherits the members of the base class and can also add its own methods. Instance of derived class is
also instance of base class. Inheritance is useful for extension and specialization. Extension uses inheritance to develop new classes from existing ones by adding new features. Specialization makes use of inheritance to refine the behavior of a general class. Mainly there are two types of inheritances - Single Inheritance and Multiple Inheritance. In single inheritance a class is derived from only one class whereas in multiple inheritance a class is derived from two or more base classes. Some object oriented programming languages such as Java and C# don’t support multiple inheritance through classes.

2.2.1.5 Polymorphism
Polymorphism enables same message to invoke different method in different situations. There are two types of polymorphisms - Compile time polymorphism (Static Binding) and Run time polymorphism (Dynamic Binding). Static binding is implemented using method overloading. In method overloading more than one method have same name but different parameters. Binding of message to method is done on the basis of parameters matching at compile time. Dynamic binding enables the same message to invoke the appropriate method based on the class of the receiver when a more specific instance is substituted for a more general instance. Dynamic binding is achieved using method overriding. If derived class has a method with same name and parameters as method of base class then method of derived class overrides the method of base class.

2.2.2 Quality of Good Object Oriented Software Systems
The primary objective of any software engineering paradigm is to develop high quality software in a timely and predictable manner. [Khan08] discusses that software design is the backbone of quality of any software system. Object oriented design is concerned with developing an object oriented model of a software system to implement the identified requirements. Three major factors that influence the quality of an object oriented design are as follows:

Reliability: It determines how well customers' requirements are fulfilled by software. This is the most important factor in evaluating quality of software design. Higher reliability means higher quality and lower reliability means lower quality.
Complexity: Complexity is a measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships. In general, object oriented design shortens the distance between the conceptual model of the problem domain and software implementation i.e. it reduces the complexity. Complexity of object oriented design is determined by complexity of its classes and relationships among classes. Complexity of classes can be further determined by complexity of its methods.

Reusability: It enables reuse of parts of software in other related applications. The number of newly created classes should be small, which means one is encouraged to reuse existing classes. Higher reuse in the software is indicator of better quality.

[Jacobson94] gives quality attributes of object oriented software systems which are classified into two categories - Suitability and Maintainability. Suitability includes Reliability, Correctness, Accuracy, Efficiency and Usability. Maintainability includes Understandability, Modifiability, Traceability, Testability, Portability and Reusability. These characteristics are not exhaustive and not even independent of each other. Additionally they often tend to conflict in a development. Therefore, it is often good to set priority of attributes before starting development. In OOSD the focus is on the maintainability characteristics. Maintenance of the product is the major objective of OOSD. However, this will also have effects on the suitability criterion, for example if it is easy to introduce changes in the software, it will also decrease the number of faults introduced when the system is modified and thus gives the highly reliable product.

[Martin02] gives principles of good OOD which are categorized by [Sarkar05] into three groups in context of design metrics. These principles are discussed below:

General Principles

The Open/Closed Principle (OCP): It states that a module should be open for extension but closed for modification i.e. classes should be written so that they can be extended without requiring the classes to be modified.

The Liskov Substitution Principle (LSP): It states that subclasses should be substitutable for their base classes i.e. a user of a base class instance should still function if given an instance of a derived class instead.
The Dependency Inversion Principle (DIP): It states that high level classes should not depend on low level classes i.e. abstractions should not depend upon the details. If the high level abstractions depend on the low level implementation, the dependency is inverted from what it should be.

The Interface Segregation Principle (ISP): It states that clients should not be forced to depend upon interfaces that they do not use. Many client-specific interfaces are better than one general purpose interface.

Cohesion Principles

Reuse/Release Equivalency Principle (REP): The granule of reuse is the granule of release. Only components that are released through a tracking system can be efficiently reused. A reusable software element can’t really be reused in practice unless it is managed by a release system of some kind of release numbers. All related classes must be released together.

Common Reuse Principle (CRP): All classes in a package should be reused together. If reuse one of the classes in the package, reuse them all. Classes are usually reused in groups based on collaborations between library classes.

Common Closure Principle (CCP): The classes in a package should be closed against the same kinds of changes. A change that affects a package affects all the classes in that package. The main goal of this principle is to limit the dispersion of changes among released packages i.e. changes must affect the smallest number of released packages. Classes within a package must be cohesive. Given a particular kind of change, either all classes or no class in a component needs to be modified.

Coupling Principles

Acyclic Dependencies Principle (ADP): The dependency structure for a released component must be a Directed Acyclic Graph (DAG) and there can’t cycles.

Stable Dependencies Principle (SDP): The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.

Stable Abstractions Principle (SAP): The abstraction of a package should be proportional to its stability. Packages that are maximally stable should be maximally abstract. Instable packages should be concrete.
2.2.3 Benefits of OOSD Approach

OOSD approach closely represents the problem domain. Because of this, it is easier to produce and understand designs. The objects in the system are immune to requirement changes. Therefore, allows changes more easily. Object Oriented Design (OOD) encourages more reuse. New classes can be developed using existing classes, thereby reduces the development cost and cycle time. OOSD approach is more natural. It provides nice structures for abstracting and leads to modular design. Brief description of benefits of this approach is as follows:

**Less Error-Prone**

The object oriented approach is nearer to the systems it has to model. The structure of the problem domain can be directly represented. This has the advantage of increased understanding and is therefore, less error-prone. Since the notion of object orientation is the same at all stages of the engineering process, it is possible to use the same terminology and the same notation throughout the whole engineering process. This in turn again increases the comprehensibility of the approach.

**Enhanced Reuse**

Due to the strong encapsulation the object oriented approach facilitates software reuse. Object orientation allows reuse of classes through various relationships among classes. Increased reuse increases the productivity and decreases development cost.

**Increased Extensibility**

This feature is achieved with the help of inheritance property. Inheritance allows the class hierarchy to be further refined. Combined with polymorphism, the super class does not have to know about the new class i.e. modifications do not have to be made at the super class.

**Understandability**

Object oriented approach solves the problem in the same way as human beings perceive and understand the real-world. It is more natural for developers to decompose a problem into objects. The notation and approach of the object oriented data model builds on the paradigm that people constantly use to cope with complexity.
Reduced Maintenance Cost
Locating and fixing errors becomes easier because objects can be maintained separately as self-contained units. The principles of good OOD contribute to application's maintainability and reduce the maintenance cost.

Real-World Modeling
Object oriented systems tend to model the real world in a more complete fashion as compared to traditional methods. Objects are organized into classes of objects, and objects are associated with behaviors. The model is based on objects, rather than on data and processing [Link1]. Real world models reduce the complexity and make the structure of program very clear.

Modularity
Object oriented systems have a natural structure for modular design: objects, subsystems, framework and so on. It means all these objects are independent of each other and are maintained separately. One can make modifications in the required object without affecting the functionality of other objects.

Increased Quality
Object oriented systems provide more quality than traditional systems, primarily because new behaviors can be built from existing objects. New applications are consists of proven, existing components. Only new code has to be test from scratch. This implies reduction in number of defects.

Scalable
Object oriented applications are more scalable then structured programming based applications. As an object's interface provides a roadmap for reusing the object in new software, it also provides all the information needed to replace the object without affecting other code. This makes it easy to replace existing code with faster algorithms and newer technology.

Information Hiding
Objects provide the benefit of information hiding. Only a limited access to information is provided to the user. Information hiding ensures data security in a program. An end-user is given an access to the essential details only by hiding the complex and non-essential
details that user might not be aware of. Information hiding is necessary to keep the user away from the complex part of the application or a program. This way the application appears to be simple and also secure [Link2].

**Resilience to Change**

Object-oriented programming also enables to evolve various versions of software. When a change is suggested, the old system need not be completely changed and re-built from scratch. Resilience to change enables easier maintenance of the program. For the same reason, even during construction, parts of the system under development can be refined without any major change in other parts [Link2].

### 2.3 SURVEY OF SOFTWARE MEASUREMENT AND METRICS

Software measurement plays a very significant role in software engineering. Importance of measurement is collectively recognized by researchers’ community, software industries and software standards. Metrics are used for measurement in software development and management to provide quantitative and objective base to software engineering. Metrics provide quantitative support to decision making for researchers, managers and developers. Software metrics can help to address the most critical issues in software development and provides support for planning, predicting, monitoring, controlling and evaluating the quality of both software products and processes [Basili88], [Kitchenham95]. Software metric is a collective term used to describe the very wide range of activities concerned with measurement in software engineering [Fenton00]. Appropriately selected metrics can help both management and engineers to maintain their focus on goals [Linda02]. The term measurement can be defined as “Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules” [Fenton94]. IEEE standard 1061-1998 defines measurement as the act or process of assigning a number or category to an entity to describe an attribute of that entity [IEEE98]. Assessment and prediction are two important uses of metrics. Monitoring progress of software for taking corrective actions, evaluating product and process are some examples of assessment. Planning resource and time, predicting size, quality or other attributes of
the delivered software are some examples of prediction. Metrics are used to measure three types of entities: Process (any activity), Product (any artifact) and Resource (people, hardware, software etc.). Process metric used to measure characteristics of the methods, techniques and tools employed in developing, implementing and maintaining the software systems [IEEE98]. Product metric is used to measure the characteristics of any intermediate or final product of the software development process [IEEE98]. Each entity (Process/Product/Resource) has some attributes which can be classified into two categories: Internal and External attributes. Internal attributes of any entity depends on only entity itself and can be measured directly for example size of a program. External attributes can’t be measured directly because they depend upon the external environment. Reliability, Productivity and Maintainability are few examples of external attributes. External attributes are measured using indirect measures. Indirect measure is represented in the form of an equation involving other measures. The equation defining an indirect measure acts as a form of measurement instrument [Kitchenham95]. Metrics can be used at each phase of software development from requirements analysis to maintenance.

2.3.1 Categorization of Metrics

Categorization of metrics discussed by [Fenton98][El-Wakil04][Gill08] are as follows:

Direct and Indirect Metrics

A direct metric is a metric that does not depend upon a measure of any other attribute [IEEE98]. It directly measures the object that we want to measure and can be computed without any interpretation. Direct metrics are important because a direct metric is presumed valid and other metrics are validated in terms of it. For example Line of Code (LOC), Number of errors in source code, Memory size etc. Indirect metric indirectly measures the object that we want to measure. Indirect metric requires interpretation. Examples of indirect metrics include defect density, program productivity, cost of software etc.

Elementary and Composite Metrics

Elementary metrics are computed directly from the product artifacts whereas composite metrics are computed via mathematical combination of elementary metrics. For example
Number Of Methods (NOM) in a class is elementary metric and WMC is a composite metric.

**Static and Dynamic Metrics**

Static metrics are collected from static artifacts of the software such as documents, design models and code listings. Examples of static metrics include LOC, WMC and CBO. Dynamic metrics are collected during run time of software. For example: Extent of Class Usage, Dynamic Coupling, and Dynamic Lack of Cohesion.

**Product, Process and Resource Metrics**

Process metrics measures the attributes of software development process and are used to improve development process. For example time, effort and cost are most important attributes of process. Team leaders are most concerned with this type of metrics. Product metrics are used to quantify outputs of software development life cycle process such as requirement specification documents, design diagrams, source code etc. Product metrics may be further classified into internal or external metrics. Internal product metrics measures the characteristics of products visible to development team only. Examples of internal product metrics include LOC, McCabe’s Cyclomatic Complexity, Halstead metrics etc. External product metrics measures the characteristics of products visible to users of the product such as reliability, performance, functionality etc. Resource metrics quantify characteristics of resources that are used to develop the software such as people, team, organization, hardware etc.

**Subjective and Objective Metrics**

Subjective metrics are dependent on the collector’s judgment. They are unreliable due to inherent incoherency. Examples include programmers’ experience, average learning time of a personnel and ease of use of the software. Objective metrics are coherent and reliable due to their irrelevance to the collector. All internal product metrics are of this category.

**Procedure Oriented Vs Object Oriented Metrics**

Procedure oriented metrics are used to measure the properties related to software developed in procedure oriented programming languages. Procedure oriented metrics include LOC, McCabe’s cyclomatic complexity, Knot metric etc. Object oriented metrics are used to measure the properties of software developed using object oriented

---

**Study and Design of Complexity, Fault-Proneness and Reusability Metrics for Object Oriented Software Systems**

23
mechanisms such as encapsulation, inheritance and polymorphism. Examples of object oriented metrics include WMC, NOC and RFC etc. Object oriented metrics are a set of standards against which one can measure the effectiveness of object oriented analysis and design techniques. Object oriented metrics are different from procedure oriented metrics because of localization, encapsulation, information hiding, inheritance and object abstraction techniques.

2.3.2 Issues Related to Metrics

Software metric is one of the most focused areas of software engineering because of its potential benefits such as planning, prediction, monitoring, controlling and evaluation. Lots of metrics are available in literature out of which only few are universally accepted by software industries due to following four reasons:

- Lack of Explicitly Well Defined Goals
- Undefined Context
- Lack of Validation
- Inconsistency

These four problems make the metrics useless. There is great significance to understand these problems and their causes/sources before proposing any metric. To overcome these problems the metric development process must deals with all of these issues. This section addresses all these issues and next section discusses few guidelines collected from literature to develop useful metrics.

2.3.2.1 Lack of Explicit and Well Defined Goals

The first major problem with metrics is inadequately defined goals. Metrics are not always defined in the context of some explicit and well defined measurement goals derived from objectives of industrial interest [Briand02a]. This is one of the major reason due to which metrics proposed by various researchers are not accepted by industries. Therefore, without well defined goals, metrics development is useless. Following are few causes of ill defined goals:

- Too generic goal e.g. the goal “To reduce complexity of software” is not well defined goal.
- Sub goals are not defined.
• Goals are not derived from industrial interest.
• Goals specifications are not formalized.
• Criteria of goal achievement are not specified.

2.3.2.2 Undefined Context
Unfortunately most of the metric specification lacks the precise context specification. Software metrics defined without specifying the context or environment, in which they will be used, may be misinterpreted by its users. For example applying component oriented metrics in object oriented environment.

2.3.2.3 Lack of Validation
It is necessary to validate the metrics before implementing it. But most of the metrics suffers from lack of validation. Without validation it is not sure that metric is measuring the same for which it is developed and correlates with existing metrics. Many metrics validation approaches are available which can be classified into analytical and empirical validation.

2.3.2.4 Inconsistency
Another major problem with metrics is inconsistency. Many metrics are not uniformly interpreted by all users. Inconsistency is the big issue in software measurement right for the beginning i.e. LOC counting. Now when software engineering is so mature even then we are searching for a consistent terminology of measurement. For instance, software measurement researchers and practitioners have not reached an agreement on the precise meaning of some terms commonly used such as ‘measurement’, ‘measure’, ‘metric’, ‘measurable attribute’ etc. [Garcia06]. The cause of this problem is lack of universally standardized terminology and formalization.

2.3.3 Guidelines for Developing Metrics
Following are some guidelines derived from literature for designing useful metrics.

2.3.3.1 Metric must have Use and User
Obviously if some metric is being developed it must have some user who will use it and benefited by its feedback. The user may be project manager or developer or tester or any person involved in software management and development. The uses of the metric also
must be clearly specified which guides the user why and what metrics should be used to get its maximum benefits.

### 2.3.3.2 Formalization

The basic definitions of measurement suggest that any measurement activity must proceed with very clear objectives or goals [Fenton94]. Some formal methods must be used to specify the goals. A metric should be unambiguously defined i.e. metric should be interpreted in the same way by all its users. One way of doing so is to define precisely what mathematical properties characterize these concepts, regardless of the specific software artifacts to which these concepts are applied [Braind96]. Some non mathematical methods are also available to formally specify the metrics. For example Object Constraint Language (OCL) can be used to define object-oriented metrics formally. The selection of metrics to fulfill the specified goals also must be formally specified.

### 2.3.3.3 Metric should be Cost Effective

Metric cost includes cost of data collection, applying metrics, training and change in software development process etc. Benefits include increase in quality, predictable software process and better decision making. It is hard to quantify the benefits of metrics therefore, heuristics may be used for cost benefit analysis. If cost exceeds the benefits then some alternates should be considered to reduce its cost. If it is not possible to reduce the cost then management will decide whether to use the metric or not.

### 2.3.3.4 Metric should be Applicable as Early as Possible

It is important to find out the artifact(s) required to compute the metric. It is widely recognized that the production of better software requires the improvement in the early development phases and the artifacts they produce [Briand94]. For example computing quality of software by design is more advantageous as compared to using test cases. Briand et al. [Briand94] emphasis on the early availability of significant metrics for better software development and management process because it allows early detection of problems, better software quality monitoring, quantitative comparison of techniques, empirical refinement of process and more accurate planning of resources.
2.3.3.5 Environment and Assumptions must be Specified

Development of metrics must consider the environment or context in which they will be applied. Metrics must be driven by context’s (process, problem domain, environmental factors etc.) characteristics in which it will be used [Briand94]. The identification of universally valid and applicable measures may be ideal, long term research goal, which can’t be achieved in the near future, if at all [Briand02a]. Also any assumption about metric must be specified. For example if an object oriented metric of reusability is based on single inheritance only then it should be specified that metric is not applicable in case of multiple inheritance.

2.3.3.6 Easy to Compute

One of the good characteristics of metric is that it should be simple to compute. If the metric is just base metric then it is very easy to compute it, because it can be directly computed. Derived metric is a function of two or more variables and difficult to compute as compared to base metric.

2.3.3.7 Apply Measurement Program

Applying metrics manually on large and complex software is not only time consuming but also cumbersome and error prone. Therefore, methods are required to compute metrics automatically. Metrics can be computed automatically using specially designed measurement programs.

2.3.3.8 Side Effects must be Identified and Controlled

It is essential to identify that how metrics can be misused in any organization and the side effects of metrics. The unintended side effects may be slowing rather than streamlining the organization and can even serve to obscure our understanding of test results and reduce the overall product quality [Hoffman00]. Douglas Hoffman [Hoffman00] and Kaner et al. [Kaner04] discusses some side effects due to the measurement. Measurement can change the employee’s behavior in order to make the results look better artificially rather than reflecting the actual status of attributes by ignoring the actual goals of organization. There must be some control strategies to avoid these side effects. Organization should motivate employees to produce actual results of measurement rather than criticizing any employee based on the measurement results.
23.3.9 Use Validated Metrics

Validation of metrics is necessary for successful software measurement because non-validated metric can be misapplied i.e. metric can be used that have no relevance to the property being measured. IEEE Std. 1061-1998 [IEEE98] defines validated metric as a metric whose values have been statistically associated with corresponding quality factor values. This definition is particularly for quality metrics but same is true for all types of metrics. According to [Briand03] a measure is valid if it actually measures what it purports to measure and it is useful i.e. it helps reach some goal (e.g. assessment, prediction). IEEE Std. 1061-1998 [IEEE98] emphasize on the use of validated metrics only i.e. direct metrics or metrics validated with respect to the direct metrics. A metric validated in one environment need not necessarily be valid in other environments or future applications. Therefore, metric shall be revalidated before it is used in other environment or application. According to Fenton [Fenton94] valid measures in assessment sense must follows representational condition i.e. there must be some mapping which maps an empirical relation system to numerical relation system in such a way that empirical relations are preserved. For validation of measures in predictive sense all the components of measure and hypotheses must be properly specified before starting validation. Empirical and analytical validations are two types of validation techniques. Empirical validation shows that metric being validated is correlated with existing metrics. Valid metric must have high degree of association with existing metrics. It is a data based validation technique coming with conclusions which is verified. For empirical validation it is necessary to specify in advance the experimental hypotheses and dependent variables. Analytical validation is a theoretical validation technique which validates the measures by predefined properties or models. This type of validation is concerned with demonstrating that a measure is measuring the concept it is purporting to measure [Briand03]. Kaner et al. [Kaner04] provided a framework for proposed metric evaluation to solve the question “How do you know that you are measuring what you think you are measuring” the evaluation framework consists of ten questions which must be answered. Weyuker [Weyuker88] proposed nine properties for analytical validation of complexity measures. Weyuker’s properties are criticized by many authors. According to Fenton
[Fenton94] the main drawback of Weyuker’s axioms is that they try to validate all types of complexities by same properties. It is impossible to measure all type of complexities using a single measure. Fenton proves that we can’t be even measure control flow complexity using single measure therefore, general complexity measure is impossible. Fenton proves in informal way that Weyuker’s properties are incompatible because a “size” type complexity measure should satisfy property-5 but “comprehensibility” type complexity measure can’t satisfy property-5. On the other hand property-6 has much to do with comprehensibility and little to do with size. According to [Mishra08] one should be careful while applying Weyuker’s properties on object oriented domain. Object oriented and component oriented metrics evaluated by Weyuker’s properties should also be evaluated by other measurement criterion to fulfill sufficiency. Briand et al. [Braind96] proposed generic properties of size, length, complexity, cohesion and coupling measures for theoretical validation.

2.3.4 Traditional Non-Object Oriented Metrics

Traditional metrics have been used for measurement of structured software systems since 1976. The ability to quantify the complexity of designs and software is a necessary condition for the creation of acceptable quality standards and refinement of estimating techniques [Tegarden92]. Traditional software metrics such as LOC, McCabe’s Cyclomatic Complexity, Halstead’s metrics etc. are used as indicators of quality. Traditional software metrics are inappropriate for object oriented software systems because traditional metrics do not address the aspects of object oriented software systems. This section focuses on various traditional software metrics. Review of object oriented metrics is given in next section.

2.3.4.1 Software Size Metrics

Many metrics attempts to quantify the size of software. The most commonly used size metric LOC, suffers from the observable deficiency that its value can’t be measured until the coding process has been completed. Function Point metric has the advantage of being measurable earlier in the development process. Some of the Halstead’s metrics are also used to measure software size [Mills88].
**LOC:** LOC is the simplest and most widely used metric of program size. It is used to measure the quantitative characteristics of source code. LOC is also represented by Thousand Source Lines Of Code (KSLOC). LOC is used as an indicator of program complexity, efforts to develop program and programmer productivity. There are number of definitions of LOC which differs in treatment of handling of blank lines, comment lines, non-executable statements, multiple statements per line, multiple lines per statement and reused lines. The most common definition of LOC seems to count any line that is not a blank or comment line, regardless of the number of statements per line [Boehm81].

**Advantages:**

- It is easy to compute.
- Counting process can be automated.

**Disadvantages:**

- It has no standard method of counting.
- It is defined on code only and does not measure the size of specification.
- It is language dependent. Therefore, programs having same functionality but developed in different languages can’t be compared.
- It is not useful for comparing manually developed and auto-generated code.
- It is poor predictor of productivity because it characterizes only one specific view of size, namely length and it takes no account of functionality or complexity.

**Token Based Metrics:** Another method to quantify the size of program is token metrics. By counting the tokens in the source code one can find out the size of program. A token is the entity that makes up a program for example variable, operator, keyword etc. [Halstead75] proposed many metrics based on token count which can measure various attributes of a program for example program vocabulary, program length, program volume, program level etc.

**Function Point (FP) Metric:** LOC and Token count are code based metrics i.e. these can’t be applied before development of program. One of the features of good metric is that it should be applicable early in software development life cycle. FP
[Albrecht79][Albrecht83] determines the size of software in terms of functionality of software at analysis phase. On the basis of FP several other metrics such as Productivity (FP / Person Months), Quality (Number of Defects / FP) and Documentation (Number of Pages of Documentation per FP) can be calculated.

FP metric is based on the following parameters:

- Number of external inputs
- Number of external outputs
- Number of logical internal files
- Number of external interface files
- Number of external inquires

Each parameter is assessed individually for complexity and assigned a weight value that varies from 3 to 15. Table 2.1 shows the weight of each function point.

### TABLE 2.1 FUNCTION POINT WEIGHT

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Input</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>External Output</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Logical Internal File</td>
<td>7</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>External Interface File</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>External Inquiry</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

FP is calculated as follows:

\[ \text{FP} = \text{UFP} \times \text{TCF} \]

Where UFP is Unadjusted Function Point calculated as sum of all the weighted parameters. TCF is Technical Complexity Factor, which is calculated as follows:

\[ \text{TCF} = 0.65 + \text{Sum of factors} / 100 \]
There are fourteen technical complexity factors - data communication, performance, heavily used configuration, transaction file, online data entry, end user efficiency, online update, complex processing, reusability, installation ease, operation ease, multiple sites, facilitate change and distributed functions. Each complexity factor is rated on the basis of its degree of influence, from no influence to very influential.

Advantages:
- FP is language independent metric because code is not required to compute FP.
- FP can be calculated early in software development life cycle.

Disadvantages:
- FP is based on subjective counting.
- FP do not considers quality of output.
- It hard to automate.
- FP is confined to only traditional data processing applications.

**Bang Metrics:** The bang metric [DeMacro82] measures the size of the project based on the total functionality of the system delivered to user. Bang can be computed using data primitives available from certain algorithms and from a set of formal specifications for the software.

**ABC Metric:** ABC metric [Fitzpatrick97] also measures the software size. This metric is based on following three components:
- Assignment count (A) - Counts explicit transfer of data into a variable.
- Branch count (B) – Counts function calls.
- Condition count (C) – Counts condition.

\[
\text{Size of Program} = \sqrt{A^2 + B^2 + C^2}
\]

Advantage of this metric is that it is independent of programming style. This metric is not widely used for measuring size because it does not represent actual size of program.

2.3.4.2 Complexity Metrics

Many metrics are used to measure program complexity out of which two most commonly used metrics are McCabe’s Cyclomatic Complexity and Information Flow metric of Kafura and Henry.
Cyclomatic Complexity ($V(G)$): The Cyclomatic Complexity [McCabe76] is based on control flow structure of the program. It computes complexity of a program as a total number of independent paths through a program. To compute cyclomatic complexity first control flow graph ($G$) of program is developed, wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point in the program. The cyclomatic complexity of such a graph can be computed by a simple formula from graph theory, as

$$V(G) = e - n + 2P$$

Where $e =$ total number of edges in the $G$.
$n =$ total number of nodes in the $G$.
$P = $ connected components in $G$. The value of a single connected program component is 1. $V(G)$ can also be calculated by counting the number of predicates, using the formula $V(G) = p + 1$. Where $p$ is the total number of predicates.

Advantages:
- It is easy to use and apply.
- It can be used as an ease of maintenance metric.
- It can be used as a quality metric because it gives relative complexity of various designs.
- It can be computed early in the software development life cycle.
- It gives total number of independent paths in the program therefore, used to generate test cases for path testing.

Disadvantages:
- Data complexity is not considered by cyclomatic complexity. It is measure of the program’s control complexity only.
- The same weight is given to nested and non-nested control structures. However, deeply nested control structures are harder to understand as compared to non-nested structures.
- It may give a misleading figure with regard to a lot of simple comparisons and decision structures.
**Information Flow Metrics**

Henry and Kafura proposed a metric based on information flow connection between a procedure and its environment [Kafura81]. Information flow metric is used to measure the complexity of a software module. It is defined as follows:

\[ C = (\text{Procedure Length}) \times (\text{Fan-in} \times \text{Fan-out})^2 \]

Where \( C \) = complexity of the module.

Procedure length = length of the module, measured by LOC or Halstead’s size metric.

Fan-in = number of calls to the module.

Fan-out = number of calls from the module.

Advantages:

- It takes into account data-driven programs.
- It can be applied at design time.
- It can be used as an indicator of maintainability.

Disadvantages:

- It can give complexity value of zero if a procedure has no external interactions.

### 2.3.4.3 Halstead’s Software Science Metrics

Halstead [Halstead77] proposed a unified suite of metrics that can be used to measure several aspects of programs, as well as the overall software production efforts. Halstead metrics are based on following parameters.

- \( n_1 \) = Distinct number of operators in a program.
- \( n_2 \) = Distinct number of operands in a program.
- \( N_1 \) = Total number of operators in a program.
- \( N_2 \) = Total number of operands in a program.
- \( n_1^* \) = Number of potential operators i.e. minimum possible number of operators for a program.
- \( n_2^* \) = Number of potential operands i.e. minimum possible number of operands for a program.

On the basis of above parameters Halstead defines many metrics to measure the properties of code such as vocabulary, length, volume etc. Halstead also defines metrics
for computing the total effort (E) and development time (T) for the software products. Following is the description of metrics defined by Halstead.

**Program Vocabulary (n)**

Halstead defines program vocabulary as:

\[ n = n_1 + n_2 \]

Where

- \( n \) = total number of distinct tokens or vocabulary of program.
- \( n_1 \) = number of unique or distinct operators in the program.
- \( n_2 \) = number of unique or distinct operands in the program.

**Program Length (N)**

Halstead defines the program length (N) as:

\[ N = N_1 + N_2 \]

Where

- \( N_1 \) = total number of operators in the program.
- \( N_2 \) = total number of operands in the program.

Thus \( N \) is clearly measure of the program's size and one that is derivable directly for the program itself. Halstead also give metric \( N' \) to compute size of a program using \( n_1 \) and \( n_2 \).

\[ N' = n_1 \log_2 n_1 + n_2 \log_2 n_2 \]

It is experimentally observed that \( N' \) gives a rather close agreement to program length. Thus \( N \) is a primitive metric, directly observable from the finished program, while \( N' \) is a computed metric which can be calculated from \( n_1 \) and \( n_2 \) before the final code is actually produced.

**Program Volume (V)**

Halstead defines a metric Program Volume (V) to measure storage volume required to represent the program.

\[ V = N \log_2 n \]

Halstead observed that code complexity increases with increase in program volume.

**Potential Volume (V*)**

Potential Volume is defined as follows:

\[ V^* = (n_1^*+n_2^*) \log_2 (n_1^*+n_2^*) \]
It is a metric for denoting the corresponding parameters in an algorithm’s shortest possible form.

**Program Level (L)**

This metric computes program quality level using formula:

\[ L = \frac{V^*}{V} \]

The closer \( L \) is to 1, the better is the program quality. The ideal value of \( L \) is 1. Halstead claim that program complexity increases as program level decreases.

**Total Effort (E)**

This metric measure the efforts required to develop the program.

\[ E = \frac{V}{L} = \frac{V^2}{V^*} \]

The unit of this metric is elementary mental discriminations.

**Development Time (T)**

This metric measure the development time as follows:

\[ T = \frac{E}{S} \]

Where \( S \) is Stroud number which represents number of elementary discriminations the human being can perform in a moment. It ranges from 5 to 20. Value of \( S \) is usually taken as 18.

Advantages:
- Halstead’s metrics do not require in-depth analysis of program structure.
- Halstead’s metrics can be used to predict maintenance efforts.
- Halstead’s metrics can measure overall quality of the programs.
- Halstead’s metrics are programming language independent.
- Numerous industries studies support the use of Halstead’s metrics in predicting programming efforts and mean number of programming bugs.

Disadvantages:
- Halstead’s metrics depends on completed code.

2.3.4.4 **Quality Metrics**

Software quality can be measured using metrics in term of reliability, maintainability adaptability, portability, correctness, efficiency etc. Software must be measured at each phase of software development because it’s hard to improve quality of software after
developing it. Quality must be achieved at each phase of software development. Two most important metrics of quality are defect and reliability metrics.

**Defect Metrics**

Defect management activities need to count the number of defects in the software product. However, there is no effective procedure for counting the defects in the program. Generally number of defects is measured in the form of number of design changes, number of errors detected by code inspections, number of errors detected in program tests, number of code changes required etc. Two defect metrics - Defect Density (DD) and Man-Hours per Major Defect (M) are defined by ANSI/IEEE standards.

\[ DD = \frac{D}{KSLOC} \]

Where D is the number of defects and KSLOC is the number of thousand lines of source code.

Man-Hours per Major Defect metric assess defects during testing. It is given as follows:

\[ M = \frac{\sum_{i=1}^{n} (T_1 + T_2)}{\sum_{i=1}^{n} S_i} \]

Where \( T_1 \) is the time spent by the test team in preparation for test execution.

\( T_2 \) is the time spent by the test team during test execution.

\( S_i \) is the number of major defects uncovered during the test execution.

\( n \) is the total number of test execution.

**2.3.4.5 Reliability Metrics**

Software reliability is the probability of failure-free software operation for a specified period of time in a specified environment. No single metric is universally accepted for measuring software reliability. Different metrics are used in different contexts to estimate software reliability; brief description of these metrics is as follows:

Mean Time To Failure (MTTF) metric attempt to measure and predict the probability of failure based on time interval between two successive failures. MTTF is suitable for systems with long processing time. Mean Time Between Failure (MTBF) is defined as the sum of MTTF and Mean Time To Repair (MTTR). MTTR metric is a measure of the time to repair or bring the system to back in operational status. System availability metric
is a measure of the time during which the system is available over long time period. This metric is suitable for non-stop, continuously running systems. Probability of Failure On Demand (POFOD) is the probability that the system will fail when a service is requested. POFOD is suitable measure for safety critical systems. Rate of Failure Occurrence (ROCOF) measures number of failure occurring in unit time period. ROCOF metric is suitable for systems where system has to process large number of similar requests.

2.4 REVIEW OF EXISTING OBJECT ORIENTED SOFTWARE METRICS

Object oriented metrics are units of measurement that are used to characterize: object oriented software engineering products (e.g. design, source code, test cases etc.), processes (e.g. activity of analysis, designing and coding) and resources (e.g. efficiency of an individual tester, productivity of individual designer etc.) [Gill08]. Huge numbers of metrics have been proposed in literature by many researchers to help both managers and developers for increasing quality of object oriented software. Importance of software metrics increases if it is applicable early in software development process. Modeling techniques such as OMT and UML increases the opportunity to apply metrics at early stages of software development and to detect the flaws at design time.

The present section surveys literature for understanding and analyzing object oriented metrics, its validations and specifications. The survey is presented by dividing it into three sections. Section 2.4.1 presents the taxonomy of object oriented software metrics. Section 2.4.2 presents the various important object oriented metrics collected from literature along with their purpose and approach used in defining the metrics. Section 2.4.2 also presents many studies that test the validity the object oriented metrics empirically or theoretically. Section 2.4.3 discusses the various specification methods of object oriented metrics and provides various approaches available for formal specification of metrics.

2.4.1 Taxonomy of Object Oriented Software Metrics

Apart from generic classification of metrics such as – Product, Process and Resource metrics; Direct and Indirect metrics; Elementary and Composite metrics, various other classifications of object oriented metrics are available in the literature. One very common
classification of object oriented metrics available in literature is based on the characteristic that is measured by the metrics e.g. coupling, cohesion, inheritance, polymorphism and information hiding metrics etc. This classification does not clearly categorized the metrics i.e. same metric may belong to more than one category. [Abreu94a] proposed a TAPROOT (TAxonomy PRecis for Object Oriented meTrics) classification framework which gives two dimensional classification of object oriented metrics. First dimension represents category and second dimension represents granularity. Category represents the properties of the product or process where metric can be used. It includes – Design, Size, Complexity, Reuse, Productivity and Quality. Granularity level includes – Method, Class and System. Intersection of these two dimensions provides total eighteen types of metrics as show in Table 2.2.

<table>
<thead>
<tr>
<th></th>
<th>Method</th>
<th>Class</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Method Design</td>
<td>Class Design</td>
<td>System Design</td>
</tr>
<tr>
<td>Size</td>
<td>Method Size</td>
<td>Class Size</td>
<td>System Size</td>
</tr>
<tr>
<td>Complexity</td>
<td>Method Complexity</td>
<td>Class Complexity</td>
<td>System Complexity</td>
</tr>
<tr>
<td>Reuse</td>
<td>Method Reuse</td>
<td>Class Reuse</td>
<td>System Reuse</td>
</tr>
<tr>
<td>Productivity</td>
<td>Method Productivity</td>
<td>Class Productivity</td>
<td>System Productivity</td>
</tr>
<tr>
<td>Quality</td>
<td>Method Quality</td>
<td>Class Quality</td>
<td>System Quality</td>
</tr>
</tbody>
</table>

[El-Wakil04] discusses two types of object oriented design metrics - Intra class and Inter classes metrics. Intra class metrics measure characteristics related to one class while inter class metrics measure features between a set of classes. Another classification divides object oriented metrics into two categories- Static and Dynamic metrics. Static metrics evaluates the characteristics of software by static analysis of design or code. [Chhabra10] conducts a survey on dynamic metrics and gives the difference between static and dynamic metrics. Dynamic metrics are used to evaluate the runtime characteristics of the software. [Yacoub99][Aine03][Khurana09] discusses the need of dynamic metrics in addition to static metrics and introduced various dynamic metrics for coupling, cohesion
and complexity. [Purao03] also discusses two types of object oriented metrics - Syntax based and Execution based. Syntax based metrics are static metrics that can be computed on artifacts without considering the execution environment whereas execution based metrics are dynamic metrics can be computed by considering run time behavior of the environment. [Etzkorn07] discusses two types of metrics – Syntactic and Semantic metrics. Syntactic metrics provides indirect mapping from metric to quality whereas semantic metrics provides a more direct mapping from the metric to its associated quality factor as compared to syntactic metrics. Authors proposed a suite of semantically - derived metrics using knowledgebase, program understanding and natural language processing techniques. [Sonil10] presents a hierarchical design quality model which classifies object oriented metrics for measuring five quality attributes - Functionality, Effectiveness, Understandability, Reusability and Maintainability. However, metrics may also overlap in this type of classification.

2.4.2 Existing Object Oriented Metrics and its Validations

The first proposal of object oriented metrics found in literature is Morris's proposal [Morris89]. Since Morris proposal, large numbers of object oriented metrics have been proposed in the literature. [Morris89] neither validates the usefulness of metrics nor determines the degree of correlation and interdependence between metrics. Only the use of the metrics to predict the productivity attributes are discussed theoretically. [Abreu94a] framework discusses the many metrics without any validation. Metrics for Object Oriented Design (MOOD) [Abreu94b] and CK [CK94] metrics suite are most commonly used metrics. Since the proposal of MOOD and CK metrics suite many studies have been conducted so far to validate the usefulness of object oriented metrics such as fault-proneness prediction and maintenance. Many other object oriented metrics are also proposed and validated by many researches. Some of major metrics are given below along with their purpose, approach and validations.

2.4.2.1 Morris et al. [Morris89]

Purpose

To help software managers for measuring software process productivity.
CHAPTER-2 SURVEY AND REVIEW OF LITERATURE

Approach
Theoretical approach is used to define the metrics that are useful for predicting productivity in terms of maintainability, reusability, extensibility, testability, reliability, comprehensibility and authorability. Metrics are neither theoretically nor empirically validated.

Proposed Metrics

Methods per Object Class

Average Number of Methods per Object Class = \[
\frac{\text{Total Number of Methods}}{\text{Total Number of Object Classes}}
\]

Maximum Number of Methods per Object Class = \(\max(\text{Number of Methods in an Object Class in the Application})\)

Minimum Number of Methods per Object Class = \(\min(\text{Number of Methods in an Object Class in the Application})\)

Higher number of methods in a class is analogous to higher complexity at class level. Less number of classes is also analogous to higher complexity at application level due to large number of objects involved in application. Higher numbers of methods have favorable as well as adverse effect on reusability. Favorable effect is due to the more opportunity for method inheritance through subclasses and adverse effect is due to more application specificity of classes. Higher complexity and application specificity of classes makes it less extensible and customizable. Testing becomes more complex with larger number of methods.

Inheritance Dependencies

Inheritance Tree Depth = \(\max(\text{Inheritance Tree Path Length})\)

Higher depth of inheritance tree has favorable effect on reusability. Deeper inheritance tree are more difficult to test. Deeper inheritance tree also diminish the comprehensibility.

Degree of Coupling between Objects

Average Number of Uses Dependencies per Object = \[
\frac{\text{Total Number of Arcs in an Object Uses Network}}{\text{Total Number of Objects}}
\]
Maximum Number of Uses Dependencies per Object =
\( \text{MAX} (\text{Number of Uses Arcs Attached to any Single Object in Uses Network}) \)

Higher number of interactions between objects means high coupling between objects. Higher coupling have negative effect on encapsulation, maintainability, reusability, extensibility, testability, comprehensibility, reliability and authorability.

**Degree of Cohesion of Objects**

Degree of Cohesion of Object = \( \frac{\text{Total Fan-in for all Objects}}{\text{Total Number of Objects}} \)

Where Fan-in is the number of other objects, which pass data to the object via messages either directly or indirectly. Lower cohesion has negative effect on reusability and reliability.

**Object Library Effectiveness**

Percent of Library Objects in an Application = \( \frac{\text{Number of Library Objects in the Application}}{\text{Total Number of Objects in the Application}} \)

Average Number of Times a Library Object is Reused = \( \frac{\text{Total Number of Objects Reuses}}{\text{Total Number of Library Objects}} \)

Existence of Object Library is boolean metric. For reusability, object library must exist so that the applications can reuse the objects from library. Higher values of object library effectiveness metrics have adverse effect on reusability.

**Factoring Effectiveness**

Factoring Effectiveness = \( \frac{\text{Number of Unique Methods}}{\text{Total Number of Methods}} \)

Value of this metrics ranges from 0 to 1.

**Degree of Reuse of Inheritable Methods**

Percentage of Potential Method Uses Actually Reused = \( \frac{\text{Total Number of Actual Method Uses}}{\text{Total Number of Potential Method Uses}} \times 100 \)

Percentage of Potential Method Uses Overridden = \( \frac{\text{Total Number of Methods Overridden}}{\text{Total Number Potential Method Uses}} \times 100 \)
Higher degree of reuse of inheritable methods means higher reuse.

2.4.12 CK Metrics Suite [CK91][CK94]

Purpose
To measure the complexity of object oriented software at design time.

Approach
Measurement theory and Bunge's ontology is used as the base to propose set of object oriented metrics. Each metric is discussed with theoretical basis, viewpoints and empirical data obtained from two sites. However, many authors argue that CK metrics suite partially follows the measurement theory [Hitz96]. Six class level metrics are proposed by [CK91] which are later refined by [CK94].

Proposed Metrics

*Weighted Methods per Class (WMC)*

\[ WMC = \sum_{i=1}^{n} c_i \]

Where \( n \) is number of methods in the class and \( c_i \) is the complexity of it method.

The complexity of the method can be measured using traditional cyclomatic complexity metric. WMC includes only those methods that are defined in class, not inherited. WMC is considered as the predictor of efforts required to develop the software, to maintain the software, application specificity and reusability. High value of WMC is not desirable because complex methods are more difficult to maintain and test.

*Depth of Inheritance Tree (DIT)*

\[ DIT = \text{Maximum path length from class to root node in the inheritance hierarchy.} \]

Higher DIT is considered as the indicator of higher design complexity and higher reusability.

*Number of Children (NOC)*

\[ \text{NOC} = \text{Number of immediate decedents of a class.} \]

Higher value of NOC is indicator of higher reuse and higher testing efforts. Class having higher number of subclasses may also indicate the improper sub classing.

*Coupling Between Objects (CBO)*

\[ \text{CBO} = \text{Number of other classes to which class is coupled.} \]
A class is coupled with other class if it uses data members or methods of other class. Higher coupling have adverse effect on maintenance and reusability. Higher coupling is also indicator of need of more rigorous testing.

**Response For a Class (RFC)**

RFC = Cardinality of response set for the class.

Where response set of a class contains all the methods of the class and other methods that can be directly called by methods of class. Higher value of RFC complicates the testing and debugging of the class.

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

\[
\text{LCOM} = \begin{cases} 
|P| - |Q| & \text{if } |P| > |Q| \\
0 & \text{otherwise}
\end{cases}
\]

Where \(|P|\) = number of method pairs having no common instance variable and \(|Q|\) = number of method pairs having common instance variable.

Lack of cohesion in a class indicates disparateness of functionality provided by the class, increases complexity and fault-proneness of the class.

**Validations**

[CK94] analytically evaluates the metrics using Weyuker's properties. All the metrics satisfy majority of the properties except property-6. In some conditions DIT and LCOM also fails to satisfy the property-4.

[Basili95] empirically validates CK metrics suite as a predictor of fault-prone classes using eight medium-sized information management systems of C++ language. Data is analyzed using statistical methods. Study concludes that all CK metrics except LCOM are useful to predict class fault-proneness during the early phases of the life-cycle.

[Briand96] proposed many generic mathematical properties to theoretically validate size, length, complexity, cohesion and coupling measures. Application of these properties on CK metrics concludes that none of CK metric is complexity metric. WMC and NOC are size measures, DIT is a length measure, CBO is a coupling measure and RFC is size as well as coupling measure. LCOM can't be classified using these properties.

[Daly96] empirically validates depth of inheritance on the maintainability of object oriented software using two C++ programs. Study concludes that software with three levels of inheritance depth are quicker to maintain as compared to software with no
inheritance and maintenance of software having five levels of inheritance is slower as compared to the software without inheritance.

[Unger98] repeats the study of [Daly96] with more complex and big programs and reports that complexity of the program and type of maintenance task affects the maintenance rather than inheritance depth.

[CK98] empirically validates CK metrics suite for managerial use (productivity, rework effort and design effort) using three financial services applications. Study concludes that high value of CBO and LCOM means lower productivity, greater rework and greater design effort. Stepwise regression analysis shows that CBO and LCOM are significant explanatory variables.

[Ajmal00] empirically validates CK metrics suite as an indicator of changeability using three systems developed in C++ language. Statistical methods (Mean, Median, Standard Deviation, Correlation Coefficient etc.) are used for validation by authors.

[Subramanyam03] empirically validates role of CK metrics (WMC, CBO and DIT) in determining software defects using software of C++ and Java language. Study reports the effect of metrics on defects and concludes that even after controlling the size of the software; metrics are significantly associated with defects.

[Sandhu05] theoretically validates CK metrics suite using Weyukar's axioms. Study concludes that none of metric comply property-7 and property-9. DIT and RFC also do not comply property-5 and property-6.

[Yuming06] empirically validates usefulness of CK metrics suite and SLOC metric, in predicting fault-proneness of classes when taking fault severity into account. Validations are performed using logistic regression and machine learning method. Data for validation is collected from public domain NASA data set. Authors conclude that CBO, WMC, RFC and LCOM are significantly statistically related to fault-proneness. DIT is not significantly correlated with fault-proneness. However, the prediction capability is better in case of low severity faults as compared to high severity faults in fault-prone classes.

[Yogesh09] empirically validates the CK metrics suite and SLOC metric with respect to fault-proneness using Support Vector Machines (SVM) model. The CBO, RFC, and
SLOC metrics are found to be related to fault-proneness. NOC and DIT metric are not found to be significantly related to fault-proneness.

[JieXu08] empirically validates CK metrics suite and SLOC metric, for fault prediction using statistical and neuro-fuzzy approach. Study concludes SLOC, WMC, CBO and RFC are reliable metrics for defect estimation.

2.4.2.3 Li et al. [Li93]

**Purpose**

To evaluate the maintainability of object oriented software.

**Approach**

Metrics are defined theoretically and validated as predictor of maintenance efforts using statistical analysis.

**Proposed Metrics**

*Coupling through Inheritance*

No specific formula is given for this metric. Both DIT and NOC can be used for this purpose. Excessive coupling may violate encapsulation and information hiding and introduces extra complexity.

*Message Passing Coupling (MPC)*

MPC = number of send-statements defined in a class.

It measures the coupling through message passing. Higher number of messages sent is indicator of higher coupling. It is based on only number of message sent out from class, type of message sent is not considered.

*Data Abstraction Coupling (DAC)*

DAC = number of ADT’s defined in a class i.e. the number of attributes that are objects of another class.

Higher DAC is indicator of more complex coupling. This type of coupling may cause violation of encapsulation.

*Number of Methods (NOM)*

NOM = number of local methods.

It is a class interface increment metric. Higher value of NOM is indicator of more complex interface of class.
SIZE1 and SIZE2

SIZE1 = number of semicolons in a class.
SIZE2 = number of attributes + number of local methods.
SIZE1 is traditional LOC metric.

Validations

The same study which propose Li metrics suite also validates CK metrics (DIT, NOC, RFC, LCOM and WMC) [CK91] and Li metrics (MPC, DAC, NOM, SIZE1 and SIZE2) using two commercial software products, UIMS (User Interface Management System) and QUES (Quality Evaluation System), developed in Classic-Ada. Validation shows that metrics are useful predictor of the maintenance efforts.

2.4.2.4 Etzkorn et al. [Etzkorn99]

Purpose

To measure the complexity of a class using code and design both.

Approach

Object oriented design and code complexity metrics are discussed, and identified the aspect not measured by prior metrics. Proposed metrics are empirically validated and compared with two versions of WMC metric.

Proposed Metrics

Average Method Complexity (AMC)

AMC = \frac{1}{n} \sum_{i=1}^{n} c_i

Where n is total number of methods in class. c_1, c_2, c_3, ..., c_n be the static complexity of methods.

AMC is a code based metric. It solves the problem of WMC, that it does not correctly measure the complexity of a class with a large number of simple member functions. At design time WMC computes complexity of class just by counting the methods in the class and in some cases it is not good predictor of implemented class.

Class Design Entropy (CDE)

\text{CDE} = -\sum_{i=1}^{n_1} \left( f_i / N_1 \right) \log_2 \left( f_i / N_1 \right)

Study and Design of Complexity, Fault-Proneness and Reusability Metrics for Object Oriented Software Systems
Where $n_1$ is the number of unique special string names.

$N_1$ is the total number of non-unique string names and $f$, is the frequency of occurrence of the $i$th special string name.

CDE is design based metric. The CDE metric measures the average entropy of a class design and gives an indication of the design complexity of the class. CDE only provides for a relative comparison of complexity between two or more classes. It is not recommended to use this metric for comparing classes from different design domains, since the operator sets used by the designs would be different. CDE can differentiate between the complexities of different member functions which have not yet been implemented.

**Validations**

Proposed metrics are empirically validated using various classes and hierarchies of three different C++ GUI packages. Results of metrics are compared with experts' evaluations. Authors conclude that the AMC metric correlates well with experts' evaluations of complexity. The AMC and WMC gives two different views of the complexities. The CDE metric correlates well with experts' evaluations of complexity and with the code complexity WMC metric, when certain types of fringe classes are not included.

**2.4.2.5 MOOD [Abreu94b]**

**Purpose**

To assess encapsulation, information hiding, inheritance, coupling, clustering, polymorphism and reuse features of object oriented systems.

**Approach**

First basic elements of each metric are formally defined then on the basis on these elements eight new metrics are proposed. Metrics are proposed based on seven criterion – metric determination should be formally defined, non-size metrics should be system size independent, metrics should be dimensionless or expressed in some consistent unit system, metrics should be obtainable early in the life-cycle, metrics should be down scalable, metrics should be easily computable and metrics should be language independent. All the metrics have been defined on system level i.e. applicable on whole system. Authors follows the uniform approach for defining all the metrics such that
numerator of each metric’s formula is the actual value of design attribute being measured and denominator is the maximum possible value of the same design attribute in the design being considered.

Authors proposed following eight metrics out of which MHF and AHF were improved later. Refined definition of MHF and AHF are referenced from [Abreu96].

**Proposed Metrics**

**Method Hiding Factor (MHF)**

\[
MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}
\]

Where \( M_d(C_i) \) is the total number of methods defined in the class \( C_i \), it includes both methods which are visible and as well as invisible outside the class. Invisible methods are the private methods of class. \( M_h(C_i) \) is the total number of methods defined in the class \( C_i \) which are invisible outside the class. TC is total number of classes in the system being measured.

Refined definition of MHF

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

Where Visibility of a method \( M_{mi} \) \( V(M_{mi}) \) is defined as follows:

\[
V(M_{mi}) = \frac{\sum_{j=1}^{TC} \text{is_visible}(M_{mi}, C_j)}{TC - 1}
\]

\( \text{is_visible}(M_{mi}, C_j) = 1 \) if \( i \neq j \) and class \( C_j \) can call method \( M_{mi} \) otherwise it is 0.

Numerator of MHF measures the sum of invisibilities of all the methods defined in all the classes. The invisibility of a method is the percentage of all the total classes from which method is not visible. Denominator is the total number of methods defined in the system being measured.

**Attribute Hiding Factor (AHF)**

\[
AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}
\]
Where $A_d(C_i)$ is the total number of attributes defined in the class $C_i$, it includes both attributes which are visible and as well as invisible outside the class. Invisible attributes are the private attributes of class. $A_h(C_i)$ is the total number of attributes defined in the class $C_i$ which are invisible outside the class.

Refined definition of AHF

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

Where Visibility of a attribute $A_{mi}$ ($V(A_{mi})$) is defined as follows:

$$V(A_{mi}) = \frac{\text{is_visible}(A_{mi}, C_j)}{TC - 1}$$

is_visible($A_{mi}, C_j$) = 1 if $i \neq j$ and class $C_j$ can call attribute $A_{mi}$ otherwise it is 0.

Numerator of AHF measure the sum invisibilities of all the attributes defined in all the classes. The invisibility of an attribute is the percentage of the entire total classes from which attribute is not visible. Denominator is the total number of attribute defined in the system being measured.

**Method Inheritance Factor (MIF)**

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

Where $M_i(C_i)$ is total number of method inherited in class $C_i$. $M_d(C_i)$ is total number of methods available in the class $C_i$.

**Attribute Inheritance Factor (AIF)**

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Where $A_i(C_i)$ is total number of attributes inherited in class $C_i$. $A_d(C_i)$ is total number of attributes available in the class $C_i$.

**Coupling Factor (COF)**

$$COF = \frac{\sum_{i=1}^{TC} \left( \sum_{j=1}^{TC} \text{is_client}(C_i, C_j) \right)}{TC^2 - TC}$$
is_client(C_i,C_j) = 1 if C_i reference at least one method/attribute of C_j otherwise 0.

**Clustering Factor (CLF)**

\[
CLF = \frac{TCC}{TC}
\]

Where TCC is total number of class clusters.

**Polymorphism Factor (PF)**

\[
PF = \frac{\sum_{i=1}^{TC} \left( \sum_{j=1}^{DC(C_i)} M_0(C_j) \right)}{\sum_{i=1}^{TC} \left[ M_0(C_i) \cdot DC(C_i) \right]}
\]

Where \( M_0(C_j) \) is total number of methods overridden in class \( C_j \) and \( DC(C_i) \) is total number of decedent classes of class \( C_i \).

**Reuse Factor (RF)**

\[
RF = \frac{\sum_{i=1}^{TC} \text{in}_{\text{library}}(C_i)}{TC} + \frac{\text{MIF} \cdot \sum_{i=1}^{TC} \left[ 1 - \text{in}_{\text{library}}(C_i) \right]}{TC}
\]

Where \( \text{in}_{\text{library}}(C_i) = 1 \) if \( C_i \) belongs to class library otherwise 0.

This metric considers the two aspects of reusability i.e. using classes from library of existing classes and reusing non-library classes through inheritance.

**Validations**

[Abreu95] use statistical (correlation) and filter metaphor approach to validate MOOD metrics suite using five class libraries written in C++ language. Theoretically design heuristics are derived which are classified into two categories: High-Pass heuristic and Band-Pass heuristic. High-Pass heuristic is the one that suggests that there is a lower limit for a given metric. Going below that limit is a hindrance to resulting software quality. Band-Pass heuristic is similar, except that we have two cutoff zones (a lower and a higher one). Except AHF all other metrics are classified into Band-Pass. Statistically relationships among size and MOOD metrics are established and results are interpreted accordingly.

[Abreu96] empirical validates MOOD metrics with software quality characteristics such as defect density, failure density and normalized rework, using eight projects of C++. Correlation of quality measures and MOOD metrics show that MIF, COF and PF are better correlated with defect density, failure density and normalized rework. MHF and
AHF are negatively correlated with defect density. COF is very highly positively correlated with defect density and normalized rework. This study also gives three predictive models based on MOOD metrics, to predict defect density, failure density and normalized rework. Finally, proposed models are also validated statistically.

[Harrison98] applies MOOD metrics on various projects and discusses the results theoretically. Authors conclude that metrics are valid within the context of theoretical framework.

[Jubair04] evaluates many Java programs using a system to evaluate and grade Java programs using MOOD metrics. Grading is done according to interval and weight factor of each MOOD metric.

2.4.2.6 Kim et al. [Kim96]

**Purpose**

To compute complexity of object oriented programs.

**Approach**

Program complexity is defined using three dimensions - Syntax, Inheritance and Interaction. Metric to compute complexity of a program is defined as a function of three dimensions. Further each dimension is defined as function of various attributes.

**Proposed Metrics**

**Complexity of a Object Oriented Program (P) (COMP(P))**

\[ \text{COMP}(P) = f( \text{SX}(P), \text{IH}(P), \text{IT}(P) ) \]

Where

\[ \text{SX}(P) = \text{Syntax complexity of program P} \]
\[ \text{SX}(P) = f(\text{IMC}, \text{NOM}, \text{NOCL}, \text{LCOM}, \text{UOC}) \]
\[ \text{IH}(P) = \text{Inheritance complexity of program P} \]
\[ \text{IH}(P) = f(\text{DIT}, \text{NODC}, \text{NODA}, \text{DOR}, \text{NOD}) \]
\[ \text{IT}(P) = \text{Interaction complexity of program P} \]
\[ \text{IT}(P) = f(\text{CBI}, \text{RFC}, \text{UCL}, \text{VOD}, \text{MPC}) \]

**Attribute’s Description**

IMC: Degree of internal method complexity

NOM: Number of methods in a class
NOCL: Number of classes in a program P
LCOM: Degree of lackness of cohesion in a class
UOC: Ratio of used classes to defined classes in a program
DIT: Depth of inheritance trees of a class
NODA: Number of all inheriting ancestors of a class
DOR: Degree of reuse by inheritance for a class
NOD: Number of disjoint inheritance trees in a program
CBI: Degree of coupling of inheritance in a class
RFC: Degree of response for a class
UCL: Number of classes used in a class except for ancestors and children
VOD: Number of violation of the law of Demeter in a class
MPC: Number of send statements in a class

Validations
The same study which proposes metric also validates the metrics analytically using Weyuker's measurement principles and found that metrics follows essential Weyuker's axioms.

2.4.2.7 Braind et al. [Briand97]

Purpose
To measure the level of coupling between classes in object oriented software systems and to empirically validates the defined metrics for predicting fault-proneness of classes.

Approach
All the metrics are proposed based on followings:

Type of relationship - It may be friend, inheritance or other.

Locus - Indicates flow of impact of change. It may be Import Coupling (IC) or Export Coupling (EC).

Type of interaction between classes - It may be Class-Attribute (CA), Class-Method (CM) and Method-Method (MM).

Proposed Metrics
\textit{IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, IFMMIC, AMMIC, OMMIC, FMMEC, DMMEC, OMMEC}
All the metrics counts the particular type of interactions between classes. Generic form of metric for class $C_i$ is

$$\text{Metric}(C_i) = \sum_{C_j \in \text{Relationship}(C_i)} \text{Interactions}(C_i, C_j)$$

First one/two letter(s) of each metric represents the type of relationship it may be:

- IF: Inverse Friend
- F: Friend
- D: Descendant
- A: Ancestor
- O: Others

Middle two letters of metric name represents the type of interaction it may be:

- CA: Class-Attribute
- CM: Class-Method
- MM: Method-Method

Last two letters of metric name represents the type of direction of coupling it may be:

- IC: Import Coupling
- EC: Export Coupling

Validations

The same study which proposed Braind’s metric suite also validates these metrics as a predictor of fault-proneness classes using C++ programs. This study concludes that these metrics are reasonable predictor of fault-proneness classes and complementary to CK measures as quality predictors.

2.4.2.8 Genero et al. [Genero00]

Purpose

To measure the complexity of UML class diagrams.

Approach

All the metrics are applicable on UML class diagrams and captures the various relationships such as association, aggregation, generalization and dependencies. Therefore, once class diagram is available any of the metric can be applied. All the metrics are proposed as closed-ended metrics i.e. falls within given range $([0, 1])$.

Proposed Metrics

**Associations vs. Classes (ASvsC)**

$$\text{ASvsC} = \frac{N_{AS}}{N_{AS} + N_C}$$
\[(N^{AS} + N^{C}) > 0\]

Where \(N^{AS}\) and \(N^{C}\) is the number of associations and classes in the UML class diagram respectively.

**Aggregations vs. Classes (AGvsC)**

\[AGvsC = \left( \frac{N^{AG}}{N^{AG} + N^{C}} \right)\]

\[(N^{AG} + N^{C}) > 0\]

Where \(N^{AG}\) is the number of aggregations in the UML class diagram.

** Dependencies vs. Classes (DEPvsC)**

\[DEPvsC = \left( \frac{N^{DEP}}{N^{DEP} + N^{C}} \right)\]

\[(N^{DEP} + N^{C}) > 0\]

Where \(N^{DEP}\) is the number of dependencies in the UML class diagram.

**Generalizations vs. Classes (GEvsC)**

\[GEvsC = \left( \frac{N^{GE}}{N^{GE} + N^{C}} \right)\]

\[(N^{GE} + N^{C}) > 0\]

Where \(N^{GE}\) is the number of generalizations in the UML class diagram.

**Generalization Hierarchy (M_{GH})**

\[M_{GH} = \begin{cases} 0 & \text{if class diagram has no generalization hierarchy} \\ \sum_{i=1}^{n} C_{i} & \text{otherwise} \end{cases}\]

Where \(C_{i}\) is the complexity of the \(i\)th generalization hierarchy within a class diagram and \(n\) is the number of generalization hierarchies within a class diagram.

\[C_{i} = FLEAF_{i} - \frac{FLEFA_{i}}{ALLSUP_{i}}\]

\[FLEAF_{i} = \frac{N_{i}^{LEAF}}{N_{i}^{C}}\]
Where $N_{i}^{\text{LEAF}}$ is number of leaf classes in the $i$th generalization hierarchy and $N_{i}^{C}$ is the number of classes in the $i$th hierarchy. $\text{ALLSUP}_{i}$ is average number of direct and indirect super classes per non-root class.

**Multiple Inheritance (MMI)**

$$M_{\text{MMI}} = \begin{cases} 0 & \text{if class diagram has no generalization hierarchy} \\ \sum_{i=1}^{n} CMI_{i} & \text{otherwise} \end{cases}$$

Where $CMI_{i}$ is the complexity of multiple inheritance of the $i$th generalization hierarchy within a class diagram and $n$ is the number of generalization hierarchies within a class diagram.

$$CMI_{i} = \frac{N_{i}^{\text{EX}}}{N_{i}^{C}}$$

Where $N_{i}^{\text{EX}}$ is number of extra parents of the $i$th class and $N_{i}^{C}$ is the number of classes in such generalization hierarchy.

**Attributes vs. Classes (AvsC)**

$$\text{AvsC} = \left(\frac{N^{A}}{N^{A} + N^{C}}\right)^{2}$$

$(N^{A} + N^{C}) > 0$

Where $N^{A}$ and $N^{C}$ are number of attributes and classes in UML class diagram respectively.

**Methods vs. Classes (MEvsC)**

$$\text{MEvsC} = \left(\frac{N^{ME}}{N^{ME} + N^{C}}\right)^{2}$$

$(N^{ME} + N^{C}) > 0$

Where $N^{ME}$ and $N^{C}$ are number of methods and classes in UML class diagram respectively.
2.4.2.9 Genero et al. [Genero02]

Purpose
To present a set of metrics based on UML relationships, which measure UML class diagram's structural complexity and to empirically validate the metrics as a maintainability predictor.

Approach
Metrics are presented theoretically and validated empirically.

Proposed Metrics

*Number of Associations (NAssoc)*
The total number of association relationships.

*Number of Aggregation (NAgg)*
The total number of aggregation relationships within a class diagram (each whole-part pair in an aggregation relationship).

*Number of Dependencies (NDep)*
The total number of dependency relationships.

*Number of Generalisations (NGen)*
The total number of generalisation relationships within a class diagram (each parent-child pair in a generalisation relationship).

*Number of Aggregation Hierarchies (NAggH)*
The total number of aggregation hierarchies (whole-part structures) within a class diagram.

*Number of Generalisations Hierarchies (NGenH)*
The total number of generalisation hierarchies within a class diagram.

*Maximum DIT (MaxDIT)*
It is the maximum of the DIT (Depth of Inheritance Tree) values obtained for each class of the class diagram. The DIT value for a class within a generalisation hierarchy is the longest path from the class to the root of the hierarchy.

*Maximum HAgg (MaxHAgg)*
It is the maximum of the HAgg values obtained for each class of the class diagram.
The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.

**Number of Classes (NC)**
The total number of classes.

**Number of Attributes (NA)**
The total number of attributes.

**Number of Methods (NM)**
The total number of methods.

**Validations**
[Genero02] conducts two experiments to empirically validate the metrics and concludes that most of the metrics (NAssoc, NAgg, NaggH, MaxHAgg, NGen, NGenH and MaxDIT) are good indicators of class diagram maintainability, but NDep metric is not good indicator of class diagram maintainability.

**2.4.2.10 Bansiya et al. QMOOD [Bansiya02]**

**Purpose**
To evaluate structural and behavioral properties of classes, objects and their relationships at design level.

**Approach**
Metrics are proposed theoretically as a part of hierarchical model for the assessment of high-level design quality attributes from object oriented designs.

**Proposed Metrics**

**Data Access Metric (DAM)**
DAM is the ratio of number of private (protected) attributes to the total number of attributes declared in the class.

**Direct Class Coupling Metric (DCC)**
DCC is the count of different number of classes that a class is directly related to. Direct coupling includes coupling due to attribute declarations and message passing (parameters) in methods.
Cohesion Among Methods (CAM)
CAM is summation of the intersection of parameters of a method with maximum independent set of all parameter types in the class.

Measure of Aggregation (MOA)
MOA is count of the number of data declarations whose types are user defined classes.

Measure of Functional Abstraction (MFA)
MFA is the ratio of number of methods inherited by a class to total number of methods accessible by member methods of the class.

2.4.2.11 Sheldon et al. [Sheldon02]

Purpose
To propose metrics for measuring maintainability of class hierarchies and to compare them with CK and Li metrics.

Approach
Maintainability is measured in terms of understandability and modifiability of class inheritance hierarchy. Metrics are derived from Directed Acyclic Graph (DAG) of inheritance hierarchy. Given metrics are explained and compared using small examples.

Proposed Metrics
Total Degree of Understandability (TU) of class inheritance DAG & Average Degree of Understandability (AU) of class inheritance DAG

TU of a class inheritance DAG = \sum_{i=1}^{t} (PRED(C_i) + 1)

AU of a class inheritance DAG = \frac{TU}{t}

Where t= total number of classes in the class inheritance DAG.

PRED(C_i) = Total number of predecessors of class C_i.

Understandability is the ease of understanding the class inheritance structure. Lower value of AU indicates better understandability, because it reflects the degree of interdependency with other classes. Less interdependency is analogous to easy to understand.

Total Degree of Modifiability (TM) of class inheritance DAG & Average Degree of Modifiability (AM) of class inheritance DAG

TM of a class inheritance DAG = TU + \sum_{i=1}^{t} (SUCC(C_i)/2)
AM of a class inheritance DAG = \( AU + \sum_{i=1}^{t} (SUCC(C_i)/2) / t \)

Where \( SUCC(C_i) = \) Total number of successor subclasses of \( C_i \).

Modifiability is the ease with which a change or changes can be made to a class inheritance structure. Authors suggest AM of a class inheritance hierarchy should be under seven. Best value of AM is four.

2.4.2.12 Aggarwal et al. [Aggarwal07a]

Purpose

To measure the amount of robustness in object oriented software.

Approach

Metrics are defined using simple mathematical formulas and validated theoretically as well as empirically. Metrics are defined using the approach similar to approach used in MOOD. Numerator of each metric is the actual value of feature being measured and denominator is its maximum possible value.

Proposed Metrics

**Number of Catch Blocks per Class (NCBC)**

\[
NCBC = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} C_{ij}}{\sum_{i=1}^{n} \sum_{k=1}^{l} C_{ik}} \times 100
\]

Where \( n = \) number of methods in a class.

\( m = \) number of catch blocks in a method.

\( l = \) maximum number of possible catch blocks in a method.

\( C_{ij} = \) jth catch blocks in it method

\( C_{ik} = \) kth catch blocks in it Method

Higher value of NCBC is indicator of more robust software.

**Exception Handling Factor (EHF)**

\[
EHF = \frac{\text{Number of exception classes}}{\text{Total number of possible exception classes}} \times 100
\]

Higher value of EHF is indicator of more robust software.
2.4.2.13 Badri et al. [Badri09]

Purpose
To design a quality indicator metric which can replace several existing metrics and captures testability and maintainability aspects of object oriented systems.

Approach
Metric is proposed on the basis of control flow paths and probabilities which can be computed from control call graph (a reduced form of control flow graph). Proposed metric captures the collaboration between classes i.e. interactions between classes.

Proposed Metrics

Quality Indicator (Qi)
For a method \( M_i \), Qi is the estimation of the probability that the control flow will go through the method without any “failure”. The Qi of a method \( M_i \) is defined as follows:

\[
Qi_{M_i} = Qi_{M_i}^{*} \sum_{j=1}^{n_i} \left( P(C'_j)^{*} \prod_{k \in \sigma} Qi_{M_k} \right)
\]

\[
Qi_{M_i}^{*} = \left(1 - \frac{tf_{j}}{tf_{max}} \right)
\]

\( Qi_{M_i} \) is Intrinsic quality indicator of the method \( M_i \)

\( tf_{j} = cc_{j} \times (1 - tc_{j}) \)

\( cc_{j} \) = Cyclomatic complexity of the method \( M_i \)

\( tc_{j} \) = Unit testing coverage of method \( M_i \) \( \in [0, 1] \)

\( P(C'_j) \) = Probability of following path \( C'_j \) of the method \( M_i \)

\( Q'_{Mk} \) = Quality Indicators of the methods included in the path \( C'_j \)

\( n_i \) = Number of paths of the Control Call Graph of the method \( M_i \)

Card(\( \sigma \)) = \( m_j \) = Number of the methods included in the path \( C'_j \).

Validations
The same study which proposes the metrics also conducts an empirical study using the principal components analysis method with seven open source Java projects, and compares new metric with several well known object oriented metrics.
2.4.2.14 Other Studies

[Lionel98b] replicates the study performed by [Lionel98a] using industrial data. This study empirically finds the relationships between coupling, cohesion, inheritance measures and fault-proneness of the object oriented system classes. Various existing coupling, cohesion and inheritance design metrics are applied on industrial project consisting of 83 manually developed classes using VC++ language. Study concludes that coupling is the most important and consistent factor that affects fault-proneness of classes. Particularly strong emphasis should be put on method invocation and import coupling.

[Lnaryana99] validates the following eight definitions of LCOM metrics using principal component analysis.

LCOM1-1: the revised CK definition, with consideration of inheritance and constructor.
LCOM1-2: the revised CK definition, with consideration of inheritance but not constructor.
LCOM1-3: the revised CK definition, without consideration of inheritance but with constructor.
LCOM1-4: the revised CK definition, without consideration of inheritance and without consideration of constructor.
LCOM2-1: the Li’s definition with consideration of inheritance and constructor.
LCOM2-2: the Li’s definition with consideration of inheritance but without consideration of constructor.
LCOM2-3: the Li’s definition without consideration of inheritance but with constructor.
LCOM2-4: the Li’s definition without consideration of inheritance and without consideration of constructor.

Authors use eighteen classes of C++ and conclude that LCOM2-1 and LCOM2-3 are the most powerful LCOM measures.

[Briand00] conducts empirical study to explore the relationships between existing object oriented design metrics and quality of software in terms of probability of fault detection in system classes during testing. Authors concludes that besides the size of classes, the frequency of method invocations and the depth of inheritance hierarchies seem to be the
main driving factors of fault-proneness. The study is conducted using three systems
developed in C++ language. 28 coupling measures, 10 cohesion measures and 11
inheritance measures are used as independent variables in this study.

[EmamOl] constructs a prediction model using commercial Java applications, to identify
the faulty classes. Study concludes that export coupling have the strongest association
with fault-proneness.

[Aine03] informally defines the dynamic version of CBO and LCOM. Static and dynamic
versions of metrics are compared using applications of Java language.

[Alshayeb03] empirically validates object oriented metrics (CK and Li metrics) as a
predictor of design efforts and source lines of code added, changed and deleted using two
different iterative software processes - the short-cycled agile process and the long-cycled
framework evolution process. Study concludes that metrics are reasonably effective in
predicting evolution changes of a system design in short-cycled but not in long-cycled
framework evolution process.

[Bandi03] empirically validates three complexity metrics - Interaction Level (IL),
Interface Size (IS) and Operation Argument Complexity (OAC) to investigate the effect
of design complexity on maintenance time. Authors conclude that each of the three
metrics by itself found to be useful in the experiment in predicting maintenance
performance.

[Aggarwal06] empirically validates CK metrics suite along with eighteen other object
oriented metrics using three projects of Java language. Results are analyzed using
descriptive statistics, principal component analysis and correlation to size.

[Aggarwal07b] validates CK metrics suite together with many other metrics, by
empirically exploring the relationships between object oriented design metrics and fault-
proneness of classes using twelve academic level Java applications. The proposed model
predicts faulty classes with more than 80% accuracy.

[Sandhu09] refines the CK metrics for application in neural network. Refined metrics' (TWMC, LTDIT, LTNOC, LCBO and TLCOM) values are used as inputs for neural
network. Many neural network algorithms are used to predict reusability of object
oriented software components using refined metrics.
[Dallal10] theoretically validates sixteen class cohesion metrics using Braind’s properties of cohesion. Study concludes that only eight cohesion metrics are valid out of sixteen metrics because these satisfy all properties of cohesion metrics.

[Kaur10] conducts an empirical study using eight object oriented metrics as independent variables and maintenance efforts as dependent variable. Data of two commercial software products are used in this study. The purpose of this study is to find out the ability of soft computing techniques (Artificial Neural Networks, Fuzzy Inference Systems and Adaptive Neuro-Fuzzy Inference Systems) for prediction of maintenance efforts. Authors conclude that soft computing techniques can be used for constructing accurate models for prediction of software maintenance efforts and adaptive neuro-fuzzy inference system technique gives the most accurate model.

2.4.3 Specifications of Object Oriented Metrics
First observed attempt to quantify object oriented software by Morries et al. [Morries89] uses informal approach to define the metrics. Formulas to define the metrics are given but neither metrics nor its elements are formally specified. [Abreu93] gives a framework for classification of object oriented metrics and various metrics for each category, but no formal approach has been used to define the metrics. Metrics are specified only theoretically. [CK91] proposes set of metrics that can be used at design time to measure complexity of object oriented software, which is later refined by [CK94]. CK metrics are defined by mathematical formulas which use set theory to define components of metrics. [Abreu94b] proposes a metric suite MOOD which specifies eight metrics using simple mathematics, set theory and formally defined functions. [Li93] proposes a set of metrics theoretically only. [Briand97] uses generic formula to formally define coupling metrics. [Genero00] uses simple mathematical formulas to define metrics for UML class diagram. [Aggarwal07a] formally specify the metrics using simple mathematical formulas.

Recent approaches to formalize object oriented metrics includes:

OCL Based Formalization
The OCL is a part of the UML standard that provides a formal and precise way to define constraints. It is an easy way to specify metrics for UML class diagrams. [Abreu01] suggests OCL as a formal language to define object oriented metrics for UML class
diagrams and presents definition of MOOD2 metrics suite using OCL. [Baroni03] presents Formal Library for Aiding Metrics Extraction (FLAME) which provides formal definition of functions using OCL. Further CK metrics are formalized using functions defined in FLAME and OCL language. [McQuillan06] extends the Baroni's [Baroni03] approach by decoupling metric definition from the underlying metamodel and presents formal definition of CK metrics using UML 2.0 metamodel. Drawback of OCL based formalism is that this approach is applicable to only UML models; it can't be used for other object oriented design models.

**Object Oriented Design Model (ODEM)**

According to [Reißing01] it is very hard to define and understand some metrics using OCL therefore, OCL is not suitable for expressing metrics. Author gives general requirements for object oriented metrics which includes:

1) Precise and unambiguous definition.
2) Based on design artifacts and automatable.
3) Applicable on high level design.

To fulfill these requirements ODEM is proposed for defining object oriented metrics based on UML metamodel. This approach defines an abstraction layer on the top of UML metamodel. Metrics are defined using various formal identifiers for components and relationships of metamodel. CK and Martin's metrics are formalized using this approach.

**XQuery Based Approach**

[El-Wakil05] proposed XQuery language for expressing metrics. This approach treats design document as data and metrics as queries on this data. Metrics are represented in XQuery language that can be executed against design documents which are represented in a format that can be queried. Authors claim that this approach is standard and much wider as compared to other approaches and it has also ability to express complex metrics.

### 2.5 SUMMARY

This chapter provides the theoretical framework of OOSD and its measurement, which sets the base for further study. OOSD is the most commonly used approach for developing high quality software systems. OOSD develops software using classes and
objects and is supported by encapsulation, abstraction, inheritance and polymorphism. Developing software using OOSD approach provides many benefits which includes - Less Error-Prone, Enhanced Reuse, Increased Extensibility, Reduced Maintenance Cost, Understandability, Real-World Modeling, Modularity, Increased Quality, Scalability, Information Hiding and Resilience to Change. The quality of object oriented software highly depends upon the OOD. Various principles are available in literature for developing good OOD. Metrics are used to quantify various attributes of software. Effective measurement can serve fundamental management objectives, such as cost and resource estimation, assess progress and future predictions. Software quality measurement is also possible through metrics. Traditional metrics are not fully suitable for measuring object oriented products, processes and resources because many features of object oriented software are not addressed by traditional metrics. Extensive literature survey is done to understand the existing object oriented metrics in terms of purpose, approach and validations. This survey gives insight view of object oriented metrics which helps for conducting further study.

It is concluded from the survey that complexity and reusability are the major areas which should focused by this study. In addition to this, predicting fault-proneness of classes using object oriented metrics is decided to be focused by this study.