CHAPTER 2

LITERATURE SURVEY

Measurement is the key element of the engineering process. Measures are used to better understand the attribute of a model that is created and to assign the quality of engineering products.

In any project estimation, size, effort, time and cost are considered as important characteristics in the overall planning of the project for achieving the objectives and goals. These parameters are also considered as important characteristics to effectively utilize the resources.

2.1 SOFTWARE MEASURES

In the existing literature, measurement has been defined as 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 (Fenton 1997).

Software measures are quantifiable measures that could be used to measure different characteristics of a software system or the software development process. Metrics, Measurements and Models go together in the process of evaluating the quality of software. The importance of software measurement is increasing continually leading to the development of new
measurement techniques (Aggarwal et al 2005). Within the software engineering context, a measure provides quantitative indications of the extent, amount, dimension and capacity or size of some attributes of a product or process.

2.1.1. Types and Classification of Metrics

There are number of metrics that are found in literature. Metrics are classified in different approaches with different context. In a broader classification, they are classified as elementary and composite metrics, subjective and objective metrics and process, product and project metrics (Fenton et al 1998). Elementary metrics are computed directly from the product artifacts, number of methods in a class and complexity of a method. Composite metrics are computed via mathematical combination of elementary metrics. They are unreliable due to inherent incoherency (Li and Seed 1993).
Figure 2.1 Metrics Classification
Product Metrics are further classified as Internal and External Metrics. Internal metrics are considered as Static and Dynamic Metrics. Static Metrics are divided into Specification, Design and Code Metrics. From Design and Code Metrics procedure and Object Oriented Metrics have been defined. The Object Oriented Metrics are further classified as per the usage like size measurement, Coupling and Inheritance measurement, Cohesion and Encapsulation.

2.1.1.1 Process metrics

Process metrics measure the attributes related to software development life cycle process in order to improve them. The most relevant process attributes are time, effort and cost. All could be estimated by Boehm's Constructive Cost Model (COCOMO) (Boehm et al 2000). Team leaders are most concerned with this type of metrics.

2.1.1.2 Product metrics

Product metrics are those that describe the characteristics of the software development life cycle processes. Outputs such as requirements specification documents, design diagrams, source code listing and end product executable programs are to be measured for quality assessment using the characteristics that are predefined.

For example in requirement specification documents, reliability is assessed using Flaesh-Kncacid Grade level index (University of Memphis, 2004). Similarly the source code size is measured using lines of code (LOC) metrics (Conte et al 1986). In object oriented design diagrams, coupling is
judged using Data Abstraction Coupling (DAC) (Li and Seed 1993). Finally, executable programs too are subject to measurement through the metrics such as Mean Time to Failure (MTTF) and Rate of Failure Occurrence (ROFOC) (Sommerville 1999). Product metrics are subdivided into internal and external metrics.

2.1.1.3 Resource metrics

Resource Metrics are those that describe the available resources characteristics like the number of developers and their skills, hardware reliability and performance. Owners and project managers are most concerned about this type of metrics. Resource metrics are also called as Project metrics.

2.1.2 Internal and External Metrics

Product metrics are sub classified into internal and external metrics. External metrics address the properties visible to the users of a product such as reliability, functionality, performance and usability (Bertrand Meyer 1998).

External metrics are harder to measure when compared to the internal metrics. Since they are of subjective nature, external metrics are not available till the late stage of the software development life cycle.

Static metrics are collected from the static artifacts of the software such as specification, documents, design diagrams and code listing. Lack of Cohesion (LOC), Weighted Methods per Complexity, and Coupling Between Objects (CBO) are some of the examples of static metrics. Dynamic metrics are collected during the run time of the software from the executable form.
The extent of class usage, Dynamic coupling and Lack of Cohesion (Boehm 2000) are examples of dynamic metrics. Static metrics are further divided into specification, design and code metrics.

Specification metrics analyze the product specifications and provide early feedback about developing software product (Demarco 1986). Other specification metrics include counts of weak phrases, lines of text and unique subjects (Hyatt et al 1996).

Design metrics are measured at a bit later stage of the software development process. This can help refining the design in order to avoid unforeseen problems and risks within the final product. Early feedback has been the primary motivation for design metrics. Design metrics include Henry and Kafuras information flow (Henry et al 1981) reuses ratio (Xenos et al 2000) and coupling factor (Abreu et al 1994).

Code metrics measure the properties of the code written by the program users based on either LOC metric (Cont et al 1986) or Knot metrics (Sheppered and Ince 1993). Design and code metrics are further classified into procedural and object oriented metrics.

### 2.1.3 Object Oriented Metrics

Procedural metrics are used to measure the properties related to software developed in procedural programming language. They are organized around the view of software in which the individual procedure or sub program is the most significant unit. Procedural metrics include lines of code (Cont et al 1986), McCabe's cyclomatic complexity and knot metrics (Sheppered and Ince 1993). On the other hand, object oriented software need additional
metrics to reflect the impact of using object oriented mechanisms such as inheritance, associations, aggregations, polymorphism and message passing. Object oriented metrics include Weighted Methods per Class (WMC), Number of children (NOC) and Response for a class (RFC) (Chidamber and Kermerer 1994).

Object oriented metrics can be further categorised according to the object oriented property they measure. For example there are metrics for measuring size such as number of methods, number of attributes and number of classes. There are metrics for measuring coupling such as direct class coupling, number of dependencies in, and coupling factor. There are metrics for measuring inheritance, cohesion and encapsulation.

Object oriented metrics measure the structural properties of object oriented software. There are two important types of object oriented metrics namely cohesion and coupling metrics that are necessary to improve the design quality. Coupling metrics characterize the static usage of dependencies among the classes in an object oriented system (Briand et al 1999). Cohesion metrics characterize the extent to which the methods and attributes of class belong together (Briand et al 1998). In addition, inheritance also plays an important role in the understandability of object oriented applications. A considerable number of object oriented metrics have been developed by the following researchers (Benlarbi and Melo 1999; Briand et al 1997; Cartwright and Shepperd 2000; Chidamber and Kermerer 1994; Henderson Sellers 1996; Li and Henry 1993; Lorenz and Kidd 1994). The most popular of these, is the metrics suit developed by Chidamber and Kamerer (1994).

The object oriented software engineering community are pre occupied with inheritance and its effect on the quality (Deligiannis and Shepperd 2002).
An experimental investigation found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance. Understanding inheritance program is more difficult based on the questionnaire (Cartwright 1998) and the author attributes this to the subject finding the inheritance program more difficult to understand, based on responses to questionnaire (Cartwright 1998). Hence it is necessary to measure the impact of designing classes, inheritance and methods.

In this thesis, we proposed a new metrics that focus on the ease of valuation with using the concept of inheritance and without considering inheritance. An attempt has been made to define new OO metrics which will be convenient for implementation and less complicated to understand and evaluate the programs/projects.

Apart from this, these metrics when incorporated in the process of estimation will reduce the effort and development time, hence relatively the cost. We also applied these metrics to calculate the object oriented design function points counting procedures. We also propose metrics models, that are suitable to conventional software as well as object oriented software.

2.2 ESTIMATION

2.2.1 Traditional Approaches

Traditional approaches to software metrics fail to address the key objectives of providing quantitative support for management decision making. They cannot be used effectively for quantitative management risk analysis, and hence do not yet address the primary objective of metrics (Norman Fenton and Niel 1999). In order to achieve quality products with reliable costs and
effort estimations, one of the main tasks for planning software project development is size estimation (Costagliola et al 2001). Moreover, process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates (Pressman 2005).

Estimating the development effort for software systems is a long standing problem in software project management. It has generally been noted that the effort is strongly correlated to program size (Oddur Benediksson and Daven Dalcher 2004). For the developer, manager and user of any software product the prediction of project effort requirements is an extremely important activity (Lederer et al 1990). The estimate arrived at frequently will form the basis for contract negotiations, resource and personal allocation over the schedule, for executing the project. It enables the project manager to plan monitor and control the subsequent development process (Londeix 1995).

2.2.2 Cost Estimation Uncertainties

Software cost estimations provides a link between concepts and techniques of economic analysis and software engineering. Because of the requirements for software and hardware and human resources, the cost of the project varies considerably (Jalote 1998). The cost of project depends on the nature and characteristics of the project. At any point of time, the accuracy of estimate depends on the availability of reliable information (Janaki Ram and Raju 2000). There is a great deal of uncertainty during the feasibility study, when the project is being initiated. However if system is specified accurately and fully, uncertainties are reduced and accurate cost estimates can be made (Jalote 1998).
Many authors during seventies analyzed project data using statistical techniques in an attempt to identify the major factors contributing to software development cost. The prototype model was the constructive cost model (COCOMO).

A short coming of the COCOMO and other models is that the independent variables were often resulted in measurement, such as the size, in terms of LOC. Such values are already measured but only after the project has been completed. It is very difficult to predict the values of such variables before the start of project. This means that many of models, although based on statistical analysis of actual results, data were hard to determine before the project team has analyzed the requirements and had prepared fairly detailed design.

Allan Albercht and John Gaffen of IBM (Albrecht 1983) developed function point analysis (FPA) to estimate the size and development effort for management information systems. Maluke H. Halstead defined software size in terms of number of operations and operands defined in the programme and proposed relations to estimate the development time and effort (Halstead 1977).

To obtain this size information before the start of a project was nearly impossible because a good understanding of the detailed design was not available until later. Subsequent work by Conte et al (1986) has shown that Halstead relations are based on limited data and this model is no longer used for estimation process (Colman et al 1994).
2.2.3 Cost Estimation Models

The Ada programming language was introduced in 1983 to reduce the cost of developing large systems. Certain features of Ada has significant impact on development and maintenance cost of software hence, Barry Boehm and Walker Royce defined a revised model called Ada COCOMO (Boehm et al 1987). This model also addressed the fact that systems were being built mentally in an effort to handle the inevitable changes in requirements.

Putnam's slim equation has suggested that effort is proportional to size, in which the effort is equal to the cube of source lines of code divided by the fourth power of development time.

Another work by Jenson asserted that development effort is proportional to the square of the size divided by the square of the development time (Jensen 1983).

Jensen's equations are reduced to those equations that are close to COCOMO's embedded mode, but the effort of various cost drivers are handled quite differently. Charles Symons (1988) proposed another revision of FPA that reduces the subjectivity in dealing with files to make the file size independent of the system implementation, and is called as mark II functions points (Symons 1988) in which the computations are based on logical transactions. In this model, each processing activity carried out by the system is analyzed in terms of the number of data items, input and output referenced. These are counted and weighted to compute the function points.
The new version of COCOMO called COCOMO - II was developed by Boehm et al (2000) which addressed various types of development processes mentioned earlier. COCOMO - II explicitly handles the availability of additional information in later stages of projects, the non linear costs of reusing software components and the efforts of several factors on the diseconomies of scale. Whatever may be good estimation techniques that have been followed in a project, it is most likely that we would have over looked some factors that will affect the accuracy of estimates (Timothy et al 2004). Hence it is necessary to estimate the cost of a project using more than one technique.

A key factor in selecting a cost estimation model is the accuracy of its estimates. Despite the large body of experience with estimation model, the accuracy of the existing cost estimation models were not satisfactorily designed (Hareton Leung et al 1999). Moreover, Software cost estimation depends on the nature and characteristics of project (Jalote et al 1998). The main disadvantage using lines of code as a unit of measure for software size is the lack of universally accepted definition and language dependency (Leviten 1986; Low and Jeffery 1990; Matson et al 1994).

In this thesis work, we propose a new metric for the enhancement of the estimation technique for accurate calculations of effort, development and cost. To implement this, the well accepted COCOMO has been adopted. The incorporation this new metric into the COCOMO model has drastically improved the estimation of effort, schedule and cost of the project.

2.3 FUNCTION POINTS

Function points were introduced by Albrecht in 1979 (Albrecht 1979). Function points are intended to measure the functionality of software systems as observed by the user, independent of the technology being used. Function
points can be distilled early in the software development process from requirement and design specifications. The notion of computing function points is a part of larger process called Function Point Analysis (FPA) (Dreger 1989).

International Function Point User Group (IFPUG, 1999) considered the class as logical files and methods as transaction functions at the analysis as well as design phase from the user's perception. This counting procedure lacks the ability to measure the inheritance and communication among the objects. However, it is possible to consider each class as a logical file and methods sent across the application boundary as transactional functions (Whitmire 1992). Schoonevelt (1995) has treated class as logical file and services delivered as transaction functions. Caldiera (1998), in his paper, considered classes as logical file and methods as transactional functions. The weight of each logical file depends on the Data Element Type (DET). It has the data inherited from a base class which is not considered to estimate the weight of logical file. Communication among objects is considered as Data Element Type and Record Element Type (RET) to a logical file. Transaction functions are weighted according to their signatures where as (Janakiram et al 2000) in their paper considered total data in the basic class as well as derived class for measuring complexity. The inherited total data is counted to estimate the complexity of the derived class, and as method signatures are available in the design phase, the complexity of communicating objects is considered for a particular method. In this thesis, we attempt to use two reuse metrics that can be used with both conventional and OO systems.

In case of Object Oriented Function Point (OOFP) counting procedure (IFPUG 1999) the complexity of a class as a factor to estimate the functionality of system has not been considered in the literature. It is difficult to design OO system in such a way that all the derived classes use all the
inherited data. An OODFP counting procedure was adopted from the traditional function point counting procedure (IFPUG 1999). The functionality of the OO software is decided according to the data processed by functions and communications among the objects. In this thesis, we enhance the COCOMO model to suit the object oriented design.

The traditional function point counting procedure is not suitable for measuring the functionality of object oriented systems because the raw functionality of software, communication between objects and lack of inheritance (Jalote 1998). For estimating the size of OO software, the criteria is based on the adaptation of classical function point method to object oriented software to map the concept of functions to methods in OO software. Most elements in the object models are related to application domains so the perspective is still from that of the user. At the OO design phase, the object model reflects the implementation choices which include the aspect of the system that are not specified in the required documents. The count of object oriented function point (OOFP) includes the functionality and measurement perspective of the designer. To map the function point in OO software, logical files are mapped on to the class and transactions are mapped on to their methods. A logical file in the function point approach is a collection of related user-identifiable data. A class in an object model encapsulates a collection of data items and objects that are instances of class in OO world corresponding to records of a logical file in data processing applications.

2.3.1 Function Point Analysis

Function points measure five characteristics of a software system in performing FPA, in which the five characteristics are weighted based on their value to the software customer (Albrecht 1979). The weights for each
characteristic are based on complexity estimation (simple, average or complex). The sum of the weighted function point count for each characteristics is termed Unadjusted Function Point Count (UFPC) (Abran and Robillard 1996).

The function point delivered is unrelated to the language or tools used to develop a software project (Albrecht and Gaffney 1983). Function point approach has features that overcome the major problems with using lines of code as measure of system size. Function points are independent of language, tools or methodologies used for implementation. They have not taken into consideration the programming languages, data management systems processing hardware, or any other processing technology. Function points estimated from requirement specifications or design specifications, made it possible to estimate development effort in the early phases of development life cycle (Low and Jeffery 1990). Service function points are based on the user external view of the system, technical users of software systems have better understanding of what function point measure (Kemerer 1987). Function point analysis appears to have advantages over lines of code as a measure of software size for use in estimating software development cost (Matsonetal 1994).

Accurate estimation size is vital, Unfortunately it is proved to be difficult, especially in the early stage of the development when the estimates are of greater importance (Caldiera 1998). Hence it is necessary to compute function points at an earlier stage in order to improve the estimation at a later stage.

Estimating different characteristics such as effort, size, cost etc, of software during different phases of software development is required to
manage the resources effectively. Function point measure can be used as input to estimate these characteristics of software. The function point procedure cannot be used to measure the functionality of an object orient systems (Janaki Ram; Raju 2000).

In this thesis, in order to improvise the object oriented design function point measurement, a new model has been proposed. This model includes the attribute reuse factor by inheritance which is proposed in this work. The incorporation of this factor has improved the object oriented design function points counting procedure. This helps to provide a better estimation.

2.4 SOFTWARE COMPLEXITY METRICS

Software reuse is the process of using existing software artifacts or components instead of building them from scratch.

Reuse process involves

- Selection of reusable artifacts or components
- Its adoption for the purpose of application
- Its integration into software product under development.

Software reuse has become a widely accepted way to reduce costs, shorten cycle times and improve quality. Figure 2.2 reusability model for code.
2.4.1 Complexity Versus Reuse

Reuse influences the software complexity in three different aspects. They are understandability, testability and maintainability. Reused code increases understandability because the code part is familiar to the software developers in terms of their functionality. Moreover, reused code is already tested in some other application and it is verified to check whether it works according to its requirements. So reuse decreases testing effort and increases test ability. Finally, reused code reduces maintenance effort because developers have experience in the change, faults and enhancements that occurs in the code. So reuse reduces maintenance cost, time, effort and increases maintainability.
2.4.2 Average Method Complexity

More complex methods are likely to be more difficult to maintain. Higher method complexity leads to a lower degree of overall application comprehensibility. Moreover, higher method complexity will reduce reliability and increase the cost of testing.

Hence, it is necessary to reduce the complexity of the software using the Cyclomatic Complexity as a base (McCabe 1976).

2.4.3 Methods Per Class

From the standpoint of code reuse, a large number of methods per object class are desirable because subclasses tend to inherit a larger number of methods from super classes. It was seen that extensibility will suffer if the number of methods per object class gets too large. Moreover, a large number of methods per object class are likely to complicate testing due to the increased object size and complexity. In order to provide an optimal number of methods per class, we prove the following relation as a design guideline.

\[
\text{Average No. of Methods per class} = \frac{\text{Total No. of Methods}}{\text{Total No. of Object Classes}} \quad (2.1)
\]

2.4.4 Object oriented design

In OO systems, the measurement process impacts on the measure of lines of code produced by the developer. The major steps involved in Object Oriented design are
Identification of classes (Object).
Identification of semantics of classes.
Identification of relationship between classes and implementation of classes.

Among all these steps, class design has highest priority in Object Oriented Design as it deals with functional requirement of the system. It must occur before system design and program design, so that the coding can be performed efficiently. Keeping this in mind, we propose new metrics that are specifically designed to measure the complexity in the design of classes in order to provide importance to classes in the design. Moreover, code reuse can be achieved effectively by including a large number of methods per class because subclasses tend to inherit a large numbers of methods from superclass. However, a large number of methods per object class are likely to complicate testing due to the increased object size and complexity. In order to tackle this problem in this work, we propose a design criteria in which the average number of methods per object class is computed by dividing the total no. of methods with total no. of object classes. Moreover, we propose additional OO metrics such as the NOC (Number of Children), DIT (Depth of Inheritance Tree), Reuse ratio and Specialization ratio metrics all of which have an impact on design.

International Function Point User Group (IFPUG, 1999) considered the class as logical files and methods as transaction functions at the analysis as well as design phase from the user's perception. This counting procedure lacks the ability to measure the inheritance and communication among the objects. However, it is possible to considered each class as a logical file and methods sent across the application boundary as transactional functions (Whitmire 1992). Schoonevelt (1995) has treated class as logical file and services
delivered as transaction functions. Caldiera (1998), in his paper, considered classes as logical file and methods as transactional functions. The weight of each logical file depends on the Data Element Type (DET). It has the data inherited from a base class which is not considered to estimate the weight of logical file. Communication among objects is considered as Data Element Type and Record Element Type (RET) to a logical file. Transaction functions are weighted according to their signatures where as (Janakiram et al 2000) in their paper considered total data in the basic class as well as derived class for measuring complexity. The inherited total data is counted to estimate the complexity of the derived class, and as method signatures are available in the design phase, the complexity of communicating objects is considered for a particular method. In this thesis, we attempt to use two reuse metrics that can be used with both conventional and OO systems.

In case of Object Oriented Function Point (OOFP) counting procedure (IFPUG 1999) the complexity of a class as a factor to estimate the functionality of system has not been considered in the literature. It is difficult to design OO system in such a way that all the derived classes use all the inherited data. An OODFP counting procedure was adopted from the traditional function point counting procedure (IFPUG 1999). The functionality of the OO software is decided according to the data processed by functions and communications among the objects. In this thesis, we enhance the COCOMO model to suit the object oriented design.

If software fails, it can have catastrophic consequences such as economic damage or loss of human life. Therefore it is important to assess the quality of software before it is put into use (Jacquelive et al 2006). The traditional approaches to both quality and resource predictions have remained fundamentally unchanged since early 1980's. The assumption is that some
measure of size / complexity is the key driver and that variations which are
determined by regression approaches are explained by other measurable
variables. However, these methods can not be used effectively for quantitative
management and risk analysis and hence do not yet address the primary
objective of metrics. In case of quality predictor (Basili et al 1984 ; Gaffners
1984; Lipow 1982; Shen et al 1985) the emphasis has been determining
regression based models of the form

\[ F (\text{complexity metric}) = \text{defect density} \]  \hspace{1cm} (2.2)

where defect density is defects per KLOC.

Design of object oriented systems is little bit involved due to the
newness of the technology, and lack of formal metrics to aid designers and
managers in managing complexity in OOD. Object oriented designs are
relatively richer in information and therefore, metrics if properly defined, can
take advantage of that information available at any early stage in the life cycle.
Most of the prior research does not exploit this additional information
(Rajendra et al 2003).

Large number of practitioners have experienced poor quality with large
systems and as a consequence the need for quality control for software became
very strong. This quality control should focus on two aspects; firstly on the
development of new standard for programming and secondly, on the
development of measures to monitor the complexity of code being produced
so as to provide bench marks and to permit poor quality sections to be
modified or rewritten.

In this thesis a new software metric model has been proposed where
software complexity metrics for object oriented systems are identified. This
model reveals that there is a proportional relationship between number of classes and total lines in an object oriented program. Another contribution of this research work is on providing a metric suit that can be used for both procedure oriented and object oriented systems. In comparison with CK metrics (Chidambar et al 1994), the metrics proposed in this thesis is of more generic in nature and is easy to compute. Moreover, from the experimental results carried out with small, medium and large projects, it has been inferred that the metric proposed in this thesis improves the software quality of all projects irrespective of their complexities.

2.5 REUSABILITY METRICS FOR USER CENTERED DESIGN

Reuse measures, determine the amount of finished product was copied and modified. Productivity is measured as a size of output divided by effort expended, and incorporates reuse (Kim Johnson 1999).

2.5.1 Reuse Economic Models

Reuse economic models are categorised as follows. Cost avoidance model, that seek to show the financial benefit of reuse in terms of money, the reuser did not have to spend. The second category returns on Investment models, shows reuse benefits in any terms such as effort as cost by giving the net cost or benefit of reuse. The third category cost benefit models attempts to itemize all the cost of reuse and balance them against benefits.

The software reusability is considered as a combined form of designs for reusability and reusable design with understandability, learnability and operability of the product to the final user or user group (Pressman 2001).
As per Metrics for Usability Standards In Compact (MUSIC) technology, the physical architecture which may be a corporate building or with a single room with all computing faculties not only play an important role in determining the reusability of the product but also the nature of computing resource (Tandler 2001). Context aware computing like user identity based services and location aware appliances like GPS base services are to be considered in determining the software reusability (Carroll 2002).

User performance measurement is carried out using a number of sub measures and there are task effectiveness, efficiency, productive period and relative user efficiency (Chang 2006).

In the existing systems they proposed only a few reusability metrics namely reuse case, reuse group, reuse purpose and reuse time. On the other hand, in this thesis, we have enhanced these reuse metrics by proposing the design attributes such as reuse interface, reuse context, work space and reusing people as well as devices. More over, we proposed a matrix model for enhancing the reusability. Finally we have established the relationships between reuse and the other computable factors.

2.6 COMPARISON OF RELATED WORKS AND THIS THESIS CONTRIBUTION

There are number of cost estimation models available in the literature. However, the proposed work is different in many ways. First, it provides a new metric called method reuse factor for improving the cost, schedule and effort estimation. These metrics are convenient for implantation and less complicated to understand and easy to evaluate the programs. Finally reduce the effort and development and cost.

For the development of any product, process is also very important. In continuation of estimations, in the literature many counting procedures were
included. However, in the proposed new model for counting the object oriented design function points there is considerable improvement which has enhanced the counting procedure and improved the percentage of error calculation.

By making an empirical evaluation, a new complexity metric model has been proposed, which provides a way to control complexity, error rate and maintenance of object oriented programs.

Finally, though lot of metrics are available with designer / developer centered as users are more important stakeholders in the systems, a user centered design and enhanced software metrics have been proposed.

We propose the metrics that focus on ease of valuation with using the concept of inheritance and with out considering inheritance

**Disadvantages**

Most of the Conventional metrics that are available are not useful to measure OO systems

Large number of practitioners have experienced poor quality with large systems and there fore have a need for quality control throw measure to monitor complexity of the code.

Lack of formal metrics to aid designers and managers in managing complexity in OOD

Existing system propose only reusability metrics like reuse case , reuse group , reuse purpose and time.

Metrics are available with designer and developer centered but not the user centered.
Few metrics are available for OO software. These are not sufficient to address the software quality expected in many complex applications.

Most existing metrics focus on measuring quality of procedure oriented software.

Most of the cases faults occur in areas of software where function points are greater than the optimal level.

**Advantages with New Metrics**

New metrics will be convenient for implementation and less complicated to understand and evaluate the program.

When incorporated in the process of estimation will reduce the effort and development time and cost.

Suitable for conventional as well as OO software.

Incorporation of this factor improve the OODFP counting procedure which helps to provide better estimates.

Metrics proposed improves software quality of all projects irrespective of their complexity, for medium, large as well as small projects.

They are generic in nature and easy to compute.

There is an enhancement of reused metrics by proposing the design attributes such as reuse interface, reuse context, work space and reusing people as well as devices.

Matrix model is used to enhance the reusability and also establish the relationship[s between reuse and other computable factor.