CHAPTER 1

INTRODUCTION
Quality means doing it right when no one is looking.

~ Henry Ford

INTRODUCTION

In modern society, computers continue to present new innovation routes and various challenges in coping with the rapid advancements in computing and communication technology. Computing has developed from an arcane art, practiced in isolated rooms by few large companies to an activity that has direct effect on the majority of the population. 'Computing and Information Technology' has been suggested as the appropriate field for research, but this term is too broad to enable the acquisition of specific and accurate results. The important unifying aspect of computer system is that it examines the technological aspect of information; i.e. computing governs information processing. As reflected by current technological developments, software innovations, apart from advancements in hardware, have been accorded considerable value. Examining certain aspects of software technology, particularly outlining software processes is important, given that such investigations facilitate the production of quality software. In traditional software development, the expansion of modern software systems has become tedious and complex, resulting in high development costs and unmanageable software quality. The solution lies in directing software development
towards the use of efficient components and cost-effective software development paradigms. Component Based Software Development (CBSD) attempts to fulfil the above mentioned challenges. In CBSD, software is constructed by integrating components rather than by developing software from scratch. The absence of stringent benchmarks for component usage may therefore lead to catastrophic results. Guaranteeing quality in software development through component based approaches necessitates the evaluation of components through software quality models. Nevertheless, most of these models, namely, the Boehm, McCall and International Standards Organization (ISO) 9126 quality models described the quality characteristics of software in a general manner. In constructing a quality model for components, one of the problems encountered is that quality is evaluated only after a component is ready for use. Existing quality models cannot be used to assess component quality because they fail to include important quality sub-characteristics. A comprehensive quality evaluation model should be designed, especially during the development stage.

1.1 Software Quality

The term 'software' was first used by John W. Tukey in 1958. In computer science, software pertains to the processing of data by hardware, programs and other data to obtain information. Software has been described as [72]:

a) a program or a cloud program or a program developed in segments with source code.
b) data used and
c) a set of agents handling complexity, testability, visibility, changeability, conformity, reliability and traceability.

Software has become the foundation of modern technology; it constitutes or controls the products and services that human beings rely on for a wide variety of daily activities, from the crucial to the trivial. Currently, software is insufficiently engineered to thoroughly fulfil its role as a foundational tool [81].

The American Heritage dictionary defines quality as 'a characteristic or attribute of something'. It therefore refers to measurable characteristics.

Peter F. Drucker quoted that the quality of a product or service does not pertain to what a supplier places in the product or service. It revolves around the benefits that customers obtain and what they are willing to pay for such advantages [48].

Juran's defines quality as a product feature that satisfies customer needs, providing customer satisfaction [87].

The American Society for Quality (ASQ) defines quality as the characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Software is a product or service free of deficiencies [189].

In today's world, quality is a core component as business strategies are designed to achieve high-quality software. Similar to the ASQ, the ISO 8402 standard defines quality as the totality of features
Introduction

and characteristics of a product or a service that influences its ability to satisfy stated and implied needs.

Hyatt *et al.* established two criteria that serve as a measure of quality: 'works well enough' and 'available when needed'. The first criterion includes satisfaction with function, performance and interface requirements, along with reliability, maintainability, reusability and correctness parameters. The second criteria, facilitates the avoidance of delays. A delay in a sub-module may cause delays in an entire system [79].

Software characteristics are measured in terms of complexity, cohesion, lines of code and number of function points, amongst other factors. Quality can be measured either for quality of design or for quality of conformance. Quality of design pertains to the characteristics specified for an item by the designer, whilst quality of conformance is the degree to which design specifications are followed during production. With respect to software, quality of design refers to requirements, specifications and system designs, whilst quality of conformance focuses on implementation on the basis of requirements and established goals. Robert Glass believes that user satisfaction is the most significant factor for measuring quality by establishing the formula and is given by equation 1.1 [68]:

\[
\text{User satisfaction} = \text{Compliant Product} + \text{good quality} + \text{delivery within budget & schedule} \tag{1.1}
\]

Software quality is the 'conformance to explicitly stated functional and performance requirements, explicitly documented development
standards and implicit characteristics that are expected of all professionally developed software’ [67].

Developing effective software that substantially influences the global market necessitates ensuring customer satisfaction and repeat business from customers. These goals reflect the quality with which software is produced. Quality is an established benchmark for the following main reasons [172]:

1. It is a must for existence.
2. It provides competitive edge.
3. It reduces costs in the longer term.
4. It increases market scope.
5. It enables customer retention.

Although these requirements indicate that quality contributes to success, it is difficult to define and impossible to measure even as it is easily recognisable.

1.1.1 Software Quality Assurance

British Standard defines quality assurance as ‘All those planned and systematic actions necessary to provide adequate confidence that a product or service will satisfy given requirements for quality’ [33]. Moreover, IEEE defines software quality assurance as:

1. A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
2. A set of activities are developed or manufactured. Contrast with quality control [80].
Waman believes that a successful project gives rise to customer satisfaction and builds a partnership leading to repeated business. It was advocated that meeting customer expectations leads to quality [85]. Software quality assurance is achieved through designing test cases and using them as a control measure to ensure desired quality. The software quality measure varies from low to high, based on the attainment of a certain level, on parameters of - conformance to requirement definition & description and Software Requirement Specification (SRS), conformance to development standards and ability to handle post implementation requirements.

Quality Assurance to the customer is possible only when it is backed by strategy of testing. Quality of software affects customer, developer and stakeholder. Adhering to the development standards ensure to a great extent the achievement of the goals. Moreover, the software quality assurance is concerned with software reliability and software safety.

Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed or maintained [5].

Quality Assurance is defined as formal activities and managerial procedures that are planned and commenced in an attempt to ensure that product and services delivered are at required quality level. Quality Assurance integrates the scope of the project or product, cost and time functions for the project. Quality Assurance stresses on process capability to deliver the right product in first go [107].
1.2 Software Quality Metric

Measurement helps in taking decisions and in judging infinite number of day to day activities. For example, to detect incoming aircrafts radar systems are measured; measurement of blood samples detect specific illness and weather prediction is based on measurement of atmospheric systems; distance measurement enables to predict when the journey will be over; time measurement helps in scheduling. Further, measurement helps in comparison, i.e. to compare the price of items and height of human beings. In good old times, measurement of software was treated as obscure, but now it has become essential. Software developers measure different characteristics of the software to verify whether the requirements are consistent and complete or whether the design is of high quality. The quality factors have been given importance by the software industry in the various quality models. To have a high quality design of the software it becomes essential to measure software quality.

Fenton et al. defined measurement 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’ [55]. According to the definition metric is illustrated as a measure. A metric system is a set of measures that can be combined to form derived measures. Measure has been defined as ‘the act or process of determining extent, dimensions, etc.; especially as determined by a standard’ (Webster’s New World Dictionary). If the
standard is well defined, concrete, then measurement will be meaningful and worth judging.

The definition of a physical quantity is the description of the operational procedure for measuring the quantity. Combining metrics result in derived units such as miles per hour, hours per week, feet per second. Developing precise and invariant standards for measurement is a process of constant refinement. The activities for which metrics has been standardized is easy to manage, however, problems do exist. An activity for which precise metrics do not exist, evaluation is a cumbersome task. Examples of such nature are common in life. It is difficult to measure intelligence of a human being and to measure the color intensity of wall paint and many more similar examples.

Besides that, if measurement devices are available then accuracy of devices is to be identified. At times the data provided from such devices is inaccurate. Moreover, the volume of data, changes the manner in which measurement is performed. A scale is replaced with other devices to measure road length. To analyze the data, common sense and experience of user is applied to use a specific tool of measurement. The measurement of key parameters is an integral part of tracking. Measurements first entail collecting a set of data. The collected data has to be analyzed in totality to draw the appropriate conclusions. Metrics derive information from raw data with a view to help in decision making using appropriate formulae or calculations.
The phrase coined by Galileo Galilei (1564-1642) centuries ago is very much applicable even in today’s world, stating ‘What is not measurable, make measurable’ [50]. It indicates to find measurable attributes in every thing, to make things more understandable and controllable. Measuring and producing metrics to determine the progress of testing is very vital.

Judging the quality of software is based upon a set of quality attributes like usability, maintainability, reliability, efficiency. The composite measures however, at times can be incorrect either because of the individual measures or because of the weights assigned to the attributes. Also because of the difference in opinion it is difficult to get consensus on which factors the quality is to be judged. New methods of measurement need to be adopted, to make measurement simpler and more accurate. A metric to count lines of code or adding a report to measure effort etc., help in better evaluation of software. For example, metrics can be used to know the software testing completion date. For which certain values are required such as total testing cases, how many tests are performed each day, how many days are required to fix defects (which will be encountered), how many defects will be found. Testing completion date is predicted on the basis of metrics. It also projects that testing and metrics are not to be used only in times of trouble. It is equally important to measure good software as well as bad ones. Tom DeMarco made an important statement that ‘You cannot control what you cannot measure’ [45]. The word ‘Control’ can be easily associated with cost of each process,
productivity of software team, goodness of the code, satisfaction of user with the software, important areas, requirements are met, fault detection, goals achieved and future scope. Besides ‘control’, measurement is also used for ‘understanding’ what is happening during development and maintenance. Also it helps in better understanding of relationships among activities and the entities they affect. Moreover, measurement is also used for ‘improvement’ of processes and system [55].

Mordechai et al. have mentioned that ‘Metrics refer to the collection and use of data which reflect productivity and quality of the products produced’ [22]. Initially, the lines of code were treated as a yardstick to measure the amount of efforts placed in software. However, the parameter ‘Lines of code’ may not give correct results due to the change in methodology and complexity of language being used, but for maintenance work or for software written using same identical language, lines of code might be a good metric. In the modern era software is coded, implemented using different categories of languages. Thus, metrics need to be derived from measurements using appropriate calculations. The same set of measurements can produce different set of metrics, to benefit different people. For a metric system to work efficiently, the following conditions must be followed:

a) Right parameters must be measured.

b) Right analysis must be performed.

c) Result must be presented in an appropriate manner.
The entire metrics process involves several steps as presented in Figure 1.1:

![Metrics Process Diagram]

**FIGURE 1.1: METRICS PROCESS**

The first step involving what measurements are crucial and collecting corresponding data is the key to the entire process. Relevance of data, depth of data and the easiness in procuring of accurate data should be taken into account. The metrics program would be performed in iteration, producing more refined measurements and results [59].

**1.2.1 Classification of Metrics**

A number of software metrics exist to measure quality significantly. A complete advantage of the component-based approach will be achieved when not only the functional parts are reused, but also when this approach leads to easier and more accurately predicting the system quality, for which the metric should evaluate a precise value, must be valid, reliable, practically computable, intuitive and posses some form of internal validity. Non-size metrics should be
system size independent and expressed in some consistent unit system. Metrics should be obtainable early in the life-cycle, down-scaleable and language independent.

Classification of metrics is based on what they measure and what area they focus on. Desikan and Gopalaswamy classified metrics as product metrics and process metrics [46]. Product metrics is further classified into project, progress and productivity metrics. Project metrics includes a set of metrics to indicate project planning and execution. Progress metrics deals with the quality of a project. The concept lies on the basis of defects found by testing team and rectified by development team. Defects are categorized to improve quality and testing. Productivity combines several measurements and parameters with effort placed on the software. It helps in estimating the release date, the team’s performance, the number of defects, the cost, etc.

Process metrics are collected across all projects and over long duration and are used for making strategic decisions. The various metrics derived in process metrics are as follows: Errors uncovered before release of the software, Defects delivered to and reported by the end users, Work products delivered, Human effort expended, Calendar time expended, Conformance to the schedule, Time and effort to complete each generic activity. The process metrics lead to long term process improvement.

Metrics play a vital role in evaluation and improvement of software quality.
1.3 Software Quality Models

Software quality model can be thought as a schema for explaining the view about quality. The objectives of the software quality model are the evaluation of high-level quality characteristics with greater accuracy to help maintainers in assessing programs and predicting the quality. The software quality models help in building better quality software by proving the relationship between the internal and external quality attributes as well as enables in identifying the need for actions for quality improvement in the early development stage of the software. Prominently some well known quality models are McCall Model (1977) [6], Boehm Model (1978) [6], ISO 9126 (1992) [6] [134] and Gillies Relational Model [67]. These models are discussed as under.

1.3.1 McCall’s Quality Model

Jim McCall developed McCall Quality Model for the US Air Force, to bridge the gap between the users and developers by mapping the user view with the developer’s priority. The model is a hierarchical model introduced in 1977 based on many quality attributes. The model distinguishes between two levels of quality attributes. Higher level quality attributes are known as quality factors. Such factors are directly measurable and are external. The second level of quality attributes are called quality criteria and are either subjectively or objectively measurable. The quality factors and criteria are shown in Figure 1.2. McCall identified three main product quality perspectives:
Product Operation, Product Transition and Product Revision, for characterising the quality attributes of a software product [6].

**a) Product Operation**

Product operation identifies quality factors which influence the extent to which the software fulfils its specifications. In other words, factors that are related to the operation of a product are: Correctness, Efficiency, Reliability, Integrity and Usability.

![McCALL QUALITY MODEL](image)

**FIGURE 1.2: McCALL QUALITY MODEL [6]**
b) Product Transition

Product Transition identifies quality factors which influence the ability to adapt the software to new environment. In other words, factors that are related to the transfer of a product from one technology to another are: Portability, Reusability and Interoperability.

c) Product Revision

Product Revision identifies quality factors which influence the ability to change the software product. The factors required for testing and maintenance are: Maintainability, Flexibility and Testability.

In total McCall identified eleven quality factors grouped into three perspectives. For each such quality factors one or more quality criteria were defined as a way of measurement.

McCall model emphasizes that ‘Quality is the totality of features and characteristics of a product or a service that bears on its ability to satisfy the given needs.’ American National Standards Institute (ANSI) standard expressed quality in terms of customer expectations. Further, a high quality product is one which has associated with it a number of quality factors, described in the requirement specifications; they could be cultured, in which they are normally associated with the artefact through familiarity of use and through the shared experiences of users; or they could be quality factors which the developer regards as important but are not considered by the customer. As McCall model has been used in United States for very large projects in the military, space and public domain. The main aim of McCall quality model is to make quality measurable. The eleven quality factors
measure external and internal quality of software. Internal quality is
the quality of the product during its development. External quality is
the quality of the finished product. External quality is visible to the
end users while internal quality is concerned with technical issues of
the software. The internal quality factors are: Efficiency, Maintainability, Testability, Flexibility, Interface facility, Reusability
and Portability. The external quality factors are: Integrity, Reliability, Usability and Accuracy.

1.3.2 Boehm Quality Model

Boehm, in 1978, introduced a software quality model similar to
McCall model. Boehm model is distinctive from McCall model due to
its three levels for quality attributes. The Boehm quality model is
shown in Figure 1.3.

----

FIGURE 1.3: BOEHM QUALITY MODEL [6]
The three levels of quality attributes are: Primary uses, Intermediate constructs and Primitive constructs. The attribute of Primary uses is a new feature with respect to McCall model, whereas Intermediate constructs are similar to McCall's quality factors and Primitive constructs are similar to quality criteria of McCall criteria. On the same lines as McCall model, Boehm Model too has characteristics, each of which contributes to overall quality. Boehm model also includes characteristics of hardware performance which otherwise are missing in McCall's model. Boehm model emphasizes on the ease of using the system and the ability, to satisfy the needs of the users [6]. According to Pfleeger Boehm model asserts that the quality software is software that satisfies the needs of the users and programmers involved in it. It reflects an understanding of quality where the software [130]:

- does what the user wants it to do,
- uses computer resources correctly and efficiently,
- is easy for the user to learn and use
- is well designed, well coded, easily tested and maintained.

Boehm quality model represents the hierarchical structure of the characteristics and each of which contributes to the quality model.

1.3.3 ISO 9126 Quality Model

The ISO 9126 model developed, in 1992, a hierarchical model, proposes six major attributes namely functionality, reliability, usability, efficiency, maintainability and portability. It is believed that these attributes are comprehensive and any other factor can be
associated with these factors. The major differentiation between ISO 9126 model and the group comprising of McCall and Boehm model is that ISO 9126 hierarchy is strict. Each characteristic is related only to one attribute. Moreover, outer attributes are related to user view of the software rather than to the developer view. Six characteristics have a total 20 attributes [6]. ISO 9126 quality model is shown in Figure 1.4.

ISO 9126 model has six major characteristics contributing to quality. These characteristics are: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability [134].

Moreover, these six factors are comprehensive in itself, which means that each such factor actually addresses more than one attribute. The ISO 9126 model, is worth full for indirect measurement and is excellent in checking the quality of a system by checklist.
1.3.4 Gillies Relational Model

Gillies [67] presented a model comprising of sixteen pairs of quality attributes. Gillies demonstrated that the relationships were not commutative, i.e. attribute A may reinforce attribute B, but attribute B may not reinforce attribute A. The model is project independent. Hierarchical models are further improved by relational models because of the use of more quality attributes. The relationships are binary and stereotyped. There is however, no weightage assigned to any of the quality attribute to judge its importance. The reason behind this query is that the importance of the quality attributes changes with the passage of time and the work associated.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>R</th>
<th>E</th>
<th>I</th>
<th>S</th>
<th>U</th>
<th>F</th>
<th>E</th>
<th>I</th>
<th>P</th>
<th>U</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>U</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>0</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Efficiency</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>O</td>
</tr>
<tr>
<td>Integrity</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>O</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>+</td>
<td>-</td>
<td>O</td>
</tr>
<tr>
<td>Security</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>O</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Understandability</td>
<td>0</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Flexibility</td>
<td>0</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Ease of interfacing</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>O</td>
<td>0</td>
<td>O</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Portability</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>-</td>
<td>0</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>User consultation</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>-</td>
<td>0</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Accuracy</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>0</td>
<td>O</td>
<td>0</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Timeliness</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>-</td>
</tr>
<tr>
<td>Time to use</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>-</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>-</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>-</td>
</tr>
<tr>
<td>Appeal</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>User flexibility</td>
<td>O</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
<td>-</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Cost/benefit</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>User friendliness</td>
<td>+</td>
<td>+</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>+</td>
<td>0</td>
<td>O</td>
<td>+</td>
<td>O</td>
<td>+</td>
</tr>
</tbody>
</table>

where
- O no relationship or that the relationship is heavily context-dependent
- + direct relationship between the metrics
- - inverse relationship between the metrics

**FIGURE 1.5: GILLIES RELATIONAL MODEL [181]**

For example, safety to human being while using the software is an important attribute in nuclear power project, however, the same
Introduction

attribute is almost ignored in word processor project. Gillies Relational
quality model is depicted in Figure 1.5.

1.4 Object-Oriented Metric Suites

Accurate measurement is a prerequisite for all engineering
disciplines and software engineering is not an exception. The
expectations of quality of the software application is high when it
yields the expected results, it is stable, adaptable and leads to
reduction in maintenance costs. If a change has been introduced in a
component, the impact of the change on the whole application has to
be determined by the developer to assess the stability of the
application. Consequently, there is a need to measure quality and
assess the component’s impact on the overall system. Metrics are
needed to measure several types of quality issues. Metrics are also
needed to study the characteristics of a given software system under
different scenarios [7] [23] [110]. To facilitate the software quality
assessment various software quality metrics have been developed.
Moreover, the metrics are designed to meet a particular set of criteria.
The various criteria that have to be met are listed below [116]:

1. The metrics determination should be formally defined.
2. Non-size metrics should be system size independent.
3. Metrics should be dimensionless or exist in some consistent
   unit system.
4. Metrics should be obtainable early in the life-cycle.
5. Metrics should be down-scalable.
6. Metrics should be easily computable.

7. Metrics should be language independent.

8. Metrics should be simple and easy to interpret.

However, the selection of an extensive set of metrics will depend on the project's characteristics and context in which they are used.

A number of quality metric suites in the industry exist namely, Chidamber and Kemerer (CK) suite, Metrics for Object-Oriented Design (MOOD) suite, Li suite and Mark Lorenz & Jeff Kidd suite [103]. Many other suites are proposed by many researchers but these had drawbacks. Lamb and Abounader model for object-oriented design metrics [99] lacked in mapping of elements provided by the model and was unable to extract valuable information. Briand, Daly and Wiist model for unified framework for coupling [30] lacked a specific process for defining software measures [55]. Reißing model for object-oriented design measurement [138] had ambiguity in terms of mapping from the front end to the formal model and it was not part of the model. Such suites were never tested and were not standardised.

Kearney criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [90]. Vessey and Weber commented on the general lack of theoretical rigor in the structural programming literature [171]. Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties and fails to display the normal predictable behaviour [133] [175]. Morris proposed object-oriented metrics that are firmly rooted in theory and relevant to practitioners in
organizations but were never tested. The object-oriented programming has gained popularity tremendously over the recent times [125] [139]. The object-oriented approach includes classes, inheritance, encapsulation, polymorphism and message passing.

1.4.1 CK Suite

The CK suite was first proposed by Chidamber and Kemerer in 1991 specifically for object-oriented software [38]. Initially Chidamber and Kemerer proposed object-oriented design complexity metrics using Bunge’s ontology [35]. They extended the work of Wand and Weber to use the object-oriented concepts of coupling, inheritance, cohesion etc. CK Suite [37] deals with measurement of a process in order to improve the process. In recent times the important software development approach as been object-oriented development. Also the urge to measure software has gained importance. Suites which existed prior to CK Suite had some drawbacks namely (1) they lacked in theoretical basis, (2) they lacked in measurement, (3) they were not generalized (4) were too labor intensive to collect data and measure etc. CK Suit overcame these limitations. However, the CK Suite of metrics did not considered complexity that arises from other object-oriented design factors such as encapsulation and polymorphism [162]. To evaluate quality on the criteria of polymorphism and encapsulation a suite MOOD exists.

CK suite adopts the four steps involved in the Object-Oriented Design process as given by Booch [27]: A) Identification of classes (and objects) B) Identify the semantics of classes (and objects) C) Identify
relationship between classes (and objects) D) implementation of classes (and objects) by defining methods and constructing views. Class design is as per the Object-Oriented paradigm for CK suite.

Weyuker [175] developed a list of desiderata for software metrics and evaluated a number of existing software metrics based on the properties evolved. The desired as a necessity list include notions of monotonicity, interaction, non-coarseness, non-uniqueness and permutation.

Fenton [55] criticised Weyuker’s properties with a view point that the properties are not predicted on a single consistent view of complexity. CK suite is designed to measure complexity of the design of classes; it can measure prior to program execution as it is a static metrics. The metrics as evaluated by CK suite are: Weighted Methods per Class (WMC), Depth Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object (CBO), Lack of Cohesion per Method (LCOM) and Response For a Class (RFC).

1.4.2 MOOD Suite

Fernando Brito e Abreu proposed the Metrics for Object Oriented Design (MOOD) metrics set. The goal of the Abreu set was to improve the Object-Oriented Design process to achieve better maintainability. Each metric in the MOOD quantifies distinct feature of Object orientation such as encapsulation, inheritance, polymorphism and message passing [56] [2]. The various metrics set are namely Attribute Hiding Factor (AHF). Method Hiding Factor (MHF), Attribute
Introduction

Inheritance Factor (AIF), Method Inheritance Factor (MIF), Polymorphism Factor (PF or POF) and Coupling Factor (CF).

1.4.3 Lorenz & Kidd Suite

Mark Lorenz and Jeff Kidd introduced Lorenz & Kidd quality model suite in 1994. The Lorenz & Kidd [111] suite is divided into two categories namely project and design. Project metrics are used to predict project needs such as staffing size, application size and scheduling. They also measure dynamic changes that have taken place in the state of the project such as how much has been done and how much is still left. These metrics are more global and less specific than design metrics. There are ten metrics identified in project metrics. Design metrics are the measurement of the static state of the project design at a particular point in time. The Design metrics has a number of metrics categorized as Method size, Method internals, Class size, Class inheritance, Method inheritance, Class internals and Class externals. These metrics are more localised and restricted in nature and they look at the quality of the way the system is being developed. Within each of these categories the metrics are further subdivided into logical related groups. Within Design metrics more than thirty metrics are identified in the model.

1.4.4 LI Suite

Wei Li and Sallie Henry [102] [98] proposed a metric suite in 1993, to measure several internal quality attributes such as coupling, complexity and size. According to the author, the attribute is related to a specific concept in object-oriented programming. Six metrics are
proposed in the suite. The metrics defined in the Li suite are: Number of Ancestor Classes (NAC), Number of Descendent Classes (NDC), Number of Local Methods metric (NLM), Class Method Complexity (CMC), Coupling Through Abstract data type (CTA) and Coupling Through Message passing (CTM).

1.5 Component Based System Development (CBSD)

The role of computers in day-to-day life has considerably expanded. Expectations from software have grown over time, prompting the development of complex applications to satisfy ever-growing demands. Highly complex software is expanded in terms of size; i.e., lines of code, which are difficult to control, resulting in high development costs, low productivity, unmanageable software quality and high risks in shifting to new technology [132]. Under these circumstances, one of the most promising software development approaches is CBSD. In 1986, Brad Cox defined the concept of software components. A component is an integrated aggregate of one or more units. A unit is a component, a component with subroutines it calls and executes is a component, etc. By this (recursive) definition, a component can be anything from a unit to an entire system [20].

An individual software component is a software package or a module that comprises of a set of related functions. In earlier times, software was built from scratch, but in present-day society, components are procured and assembled as software systems. Components can produce or consume events and are event driven. In the early 1990s Apple initiated the first commercial implementation of
CBSD with its Macintosh ‘Publish and Subscribe’ technology [72]. Since then, CBSD has been extensively practiced worldwide. It is an approach in which loosely coupled independent components are assembled into systems, a practice that provides short-term and long-term benefits for the software itself. Component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [77].

Component may be is a class or a cluster of classes which are tightly coupled, but it is always a conceptually atomic unit. For example, it might be the set of classes that provide the functionality of binary tree. This would include an iterator class and a link class in addition to the tree class. The aim is that the coupling among internals of this component is more intense than the coupling of the component with any external objects [186].

CBSD has emerged as the key technology for efficiently producing evolvable and high-quality software. In a bottom-up approach, components are developed and/or are available, and then combined to generate solutions for complex and distributed applications for a wide variety of platforms as depicted in Figure 1.6. The goal of this approach is to reduce development costs and efforts. CBSD is confronted with the issue of interoperability because component-based software systems are reusable heterogeneous components built by different developers, at different times with different applications conceptualised at the time of development. The adaptability of a
component to new environments, in terms of system software, hardware configuration and specific user needs, is a critical concern. Nonetheless, the various advantages of the CBSD approach outweigh its shortcomings and provide rewards, including increased productivity, reduced time-to-market, effective management of complexity, improvements to reusability and maintenance.

![Component-Based Software Development Diagram](image)

**FIGURE 1.6: COMPONENT-BASED SOFTWARE DEVELOPMENT**

The component-based approach relieves the pressure experienced by developers in software development. Traditionally, software development focused on one system at a time and the system was evolved using waterfall model. This approach led to pressure in terms of deadlines, budgets and quality requirements. Conversely, CBSD presents a methodology for assembling components that are already ready to use and are prepared for integration. CBSD requires a mature and a systematic focus on component aspects in software development [81]. Building systems from components and building components for different systems decreases the duration of life cycle of software. The components need to be analyzed, designed, produced and deployed. In the whole process quality has to be monitored.
closely. The process has to be repeated for all components. The entire activity is short in comparison to traditional software development. The success of software development in CBSD depends very much on the success of components and their integration with other components.

Software products are expected to meet high functional and quality standards. Component based applications have parallel two separate life cycles, one for the component and other for the entire system. Components have their own requirement specifications, are developed in budget constraints and have to adhere to schedule. The system in a whole too follows requirement specifications, face financial limits and meet deadlines, but the domain of the limitations is different from that of the component e.g. the requirement for the system would include the entire software. The components on the other side are limited to small set of features. However, for the system there is an element of risk that a component may not completely satisfy the requirement of certain software or that it may have features not known to application developer. Modification in the environment, software or component during integration of components may lead to a risk that the change introduced will reduce the reliability of the system due to different unforeseen mismatches. Components are generally result oriented with little or no explanation of how the result was obtained and quality evaluation is difficult in terms of assigning and determining quality attributes to the component. Quality attribute is the characteristics of software in the form of security, safety,
performance, dependability, maintainability, accuracy, usability and many more. Unlike traditional systems, individual components can be used by many assemblers for diversified purposes in many applications. This complicates the testing process as the developer is forced to unit test each component without knowing exactly how it will be used or who will use it [165]. After the component is developed it is made available to market for sale. The customer may not purchase it due to poor quality of component or may not be able to evaluate the quality and hence lose the opportunity to use a good component [161].

Most of the software quality judging attributes are non quantifiable hence making the task of quality evaluation difficult. As an emerging paradigm CBSD requires a new quality framework. Conventional measures of critical indicator may not necessarily work well in the new paradigm [177]. Beside these challenges the advantages of CBSD including effective management of complexity, reduced-time-to market, increased productivity, a greater degree of consistency and a wider range of usability make the CBSD more and more lucrative [32].

Component based development re – establishes the idea of re-use and introduces new elements. In component based development, software’s are built by assembling components already developed and prepared for integration. Component is any set of code written, which can be isolated to address a single problem. For all future software developments, it is determined whether the same problem exists or at least a problem which is closest to the problem earlier encountered exists. Now, the solution in the form of set of code, which formed as
the solution to the earlier problem is termed as component. The same
cOMPONENT may at times solve a set of problems and yet it is referred
as a component. But components should be capable enough to solve a
problem in isolation so that the same component is used for solving
similar or same problem in some other software [32].

General examples of concrete components include Interface,
computATIONAL, memory manager, controller components and web
services. Interface components transform the representation used by
one system component into the representation used by another
component. An example of an interface component is a human
interface component that takes some system model and displays it for
the human operator. Computational components perform a
computation of some sort. Examples of computational components are
(mathematical) functions and filters. A memory component maintains
a collection of persistent, structured data, to be shared by a number
of other components. Examples are databases, a file system or a data
base table. A manager component contains a state and a number of
associated operations. When invoked, these operations use or update
the state and this state is retained between successive invocations of
the manager’s operations. Abstract date types and servers are
examples of manager components. A controller component governs
the time sequence of other events. A top-level control module and a
scheduler are examples of controller components. A web service is a
reusable software component that semantically encapsulates discrete
functionality and is distributed and programmatically accessible over
standard internet protocols. Specifically, a web service is a stack of emerging standards that describe a service-oriented, component based application architecture [96].

Components are evolved today to meet the growing complex needs of the users. Even then there are a few hindrances which exist for components. The major factor leading to the usage of components is reusability, but due to lack of certain following basic requirements of reusability, at times it has not become the driving force.

1. Classify requirements: The requirements to be classified based on market or business or application or interface or support or core requirements.
2. Match repository for reusable code such that the requirements do not match suitably with the available code.
3. Development of code should be a standardized process which at times is not practised.
4. The reuse methodology should use system development life cycle to have better results.

Moreover, the review mechanisms of reusable code requires considerations while identifying common set of code for identifying commonalities of related systems. Such considerations could be as follows:

5. Shareable set of code should be centralized managed.
6. Design of reusable code to be reviewed.
7. Information about reusable code be made available to all.
8. Related systems be reviewed to search common problems.
9. Always a search be attempted to find an existing code which matches with the requirements. Even if such a match is not found even then find one which can be modified to satisfy the requirements.

10. Some mechanism be evolved to manage all the available reusable code [96].

Further more, reuse requires some modification of the object being reused and reuse must be integrated into specific software development. It is difficult to evaluate as to what is reusable and what isn’t. Moreover, the changes to be made are not formalized [17].

1.6 Need of the Study

The software quality has been given importance for the past few decades by different researchers. The studies related to software quality have been standardized in the form of internationally recognized models such as, McCall quality model-1977 developed by Jim McCall for the US air force [6]. McCall identified 11 quality factors in all grouped into 3 perspectives. For each such quality factor one or more quality criteria are defined as a way of measurement. The McCall’s criteria of quality are Usability, Integrity, Efficiency, Correctness, Reliability, Maintainability, Flexibility, Testability, Portability, Reusability and Interoperability [67].

An overall quality assessment can be made about a software product by evaluating the criteria for each factor. Quality criteria can be termed as second level quality attitudes or internal attributes. Similarly another internationally acclaimed model-Boehm Software
Quality Model-1978 is now a standard model was introduced by Boehm. However, distinctive characteristics of Boehm Model, from McCall Model are its three levels for quality attributes namely, Primary uses, Intermediate constructs and Primitive constructs. The attribute of Primary uses, is a new feature with respect to McCall model, whereas Intermediate constructs are similar to McCall’s quality factors and Primitive constructs are similar to quality criteria of McCall model’s criteria. On the same lines as McCall model, Boehm Model too has characteristics, each of which contributes to overall quality. Boehm model also includes characteristics of hardware performance which otherwise is missing in McCall’s model. Boehm model gave weightage to the ease of using the system and the ability to satisfy the needs of the users [6].

A recent development, in 1992, gave birth to ISO 9126 model. A hierarchical model, propose six major attributes namely Functionality, Reliability, Usability, Efficiency, Maintainability and Portability [6]. The model proposes that each factor is associated only to one attribute.

Yet another model, Gillies Relational Model, composes of sixteen pairs of quality attributes.

Each of these models, have similar quality factors. Some of the factors are common and are prevalent in all the mentioned models such as: Usability, Efficiency, Reliability, Maintainability and Portability. The remaining factors are indirectly integrated within the other factors.
Extensive work and study has been performed in development of these models by various eminent researchers and have been able to identify quality factors, but these factors have not been quantified. Related studies have either provided definitions and classifications of the quality characteristics based on the models or have evaluated quality for the Commercially Of The Shelf (COTS) components.

Further, it is realized that to provide quality product, numerous software quality models exist which have been tested on the existing software. In the past, studies [24] [146] [137] have been conducted with the scope limited to COTS to measure quality. Another set of related work [69] [141] [183] [92] were oriented towards development of software models and did not apply metrics to quality factors.

In times to come, CBSD shall be more practiced, used in development and management of software due to the obvious advantage of; a bigger complex software becomes easy to manage, reduces time, increases productivity, improves quality, a greater degree of consistency is achieved and a wider range of usability is attained. Hence, component based systems before actually put to use needs to prove their quality that whether these systems can fit in the frame work decided by the developers or not. For this reason, quality of components would gain more weightage in the success of software. Thereby, the quality should be evaluated and quantified during the development of component so that if the component is below standard limit of acceptance, it may be improved rather than quality being measured after the development of component. Quality measurement
of the ready-to-deliver product may lead to non-acceptance of component and will not provide an opportunity to enhance the quality of component by the developers. The need to evaluate quality of a component during its development stage is a necessity to improve the component. An opportunity must exist to check, test and improve the quality of the component during development stage.

There exists a gap in research in component based system particularly in the quality assurance from the point of view of developer. Keeping in view these gaps and its importance the following study was undertaken

"COMPONENT BASED SYSTEMS:
A QUALITY ASSURANCE FRAMEWORK".

1.7 Objectives of the Study

The broad objective of the study is to provide a conceptual and practical framework for the measurement of software quality on the basis of various software quality factors like efficiency, maintainability, reliability, portability and usability among mapping of metrics. The specific objectives of the study are:

a) To design a framework by constructing a structure wherein the attributes of the structure are defined.

b) To identify the various component quality metrics and to create the framework of these metrics.

c) The identified metrics in the form of quality assurance framework will be applied on to the component(s) to validate the identified metrics.
d) To map the developer's perspective of the various quality factors and sub-factors to the identified metrics.

e) To propose a quality assurance model for Component Based Software Development.

1.8 Scope of the Study

Traditional approach is fading away and is giving way to the gaining popularity of CBSD and object-oriented approach for development of software systems. Object-oriented CBSD have been widely accepted because of many attributes like reusability, polymorphism, time-saving, better abstraction, easy maintenance, cost effectiveness and simplicity in the software code. Software's are being developed in the present paradigm using object-oriented features. Object-oriented OSS components developed in Java are used for the purpose of evaluation of quality. Three components with each comprising of three versions were chosen. Software quality factors have formed a part of software quality models. Keeping in view the constraints and the objectives of the study the research is confined to McCall quality model, Boehm quality model, ISO 9126 quality model and Gillies Relational model from wherein the quality factors for evaluation are selected. Further, the selected quality factors were five to keep the research in manageable reach. Several quality models have highlighted various quality factors which are crucial in the analysis of quality of a component. Models though apart in their creation but have certain quality factors which are common. The present study is confined to the following common, vital, critical
Introduction

factors i.e. Usability, Efficiency, Reliability, Maintainability and Portability with the developer's perspective. These factors have a widespread impact on the evaluation of quality. Along with studying the quality factors, the sub-factors with respect to each factor are identified so as to assign the quality metrics.

Evaluation of software quality requires software metrics. The metrics are applied on the components from the object-oriented domain. The metrics are selected from the Chidamber and Kemerer (CK) suite and Metrics for Object Oriented Design (MOOD) suite.

1.9 Research Methodology

The collected data for the purpose to meet the objectives is both secondary as well as based on results obtained on applying identified metrics on the obtained component(s) through open source. Data was collected through case studies, experimental setup, reports, documented manuals, research articles and literature. To achieve the research objectives a detailed literature survey on the various software quality models to identify quality factors was performed. Further, the quality sub-factors were identified. The software metrics suite were studied namely Chidamber and Kemerer, Metrics for Object Oriented Design, Li and Lorenz and Kidd to identify the most appropriate set of metrics to evaluate quality for CBSD in object-oriented domain. The software quality metrics suites, applied on the open source software components by the use of software tools, were ascertained. The components selected for the experiment were Apache ivy, Heritrix and JfreeChart with three versions of each. The software tools applied on
the components were Chidamber and Kemerer Java Metrics and Essential Metrics. By means of statistical software tool, PSPP, the results were interpreted and analysed to map the metrics onto the quality sub-factors to propose the software quality assurance framework. Thereby, quality of the software component is assessed during the development stage of the component.

1.10 Chapter Plan

The thesis has been organised into seven chapters as follows:

Chapter 1 Introduction: Various terms related to software quality and CBSD are given in this chapter. The conceptual details about software quality assurance, its measurement, classification of metrics, software quality models, object-oriented metric suites and CBSD are presented. Further, the need and scope of the study is explored. The research objectives are elaborated. The study is based on five objectives. Finally, the research methodology to meet the objectives is put forth.

Chapter 2 Literature Review: The chapter comprehensively discussed the literature related to software quality, software quality models, software quality metrics, object-oriented metric suites and CBSD. Besides the definitions, studies were also presented.

Chapter 3 Research Methodology: In this chapter sequence of steps to be followed to meet the objectives are discussed. Methods used to conduct theoretical as well as empirical research are elaborated. The selection criteria for choosing the software tools and research objects
are discussed. Finally, the statistical tool used for analysis is explained.

Chapter 4 Identifying Software Quality Factors, Sub-Factors and Metrics for CBSD: This chapter deals with a detailed description of the software quality factors and sub-factors for each identified factor. The factors constituting the study are efficiency, maintainability, portability, reliability and usability. The chapter also comprises of software quality metrics and identification of metrics for the research. It was concluded that the metrics which measure quality of the component are depth of inheritance tree, weighted methods per class, number of children, coupling between objects, response for a class, method hiding factor, attribute hiding factor, method inheritance factor, attribute inheritance factor and polymorphism factor.

Chapter 5 Application of Software Tools on Research Objects: An Analysis: This chapter presents an account of the software tools applied in the research. Chidamber and Kemerer Java Metrics (CKJM) and Essential Metrics (EM) software tools selected for the study are explained in their working and scope. An outline of the three research objects chosen for the experiment is given. Apache ivy, Heritrix and JFreeChart along with the versions of each are briefly explained. For the statistical computation the statistical package was chosen from the OSS domain. Details of the statistical tool PSPP are provided. The results obtained from the experiment are placed next. Descriptive statistics for the three versions of research objects is given along with
correlation results and analysis of the software metrics with respect to CKJM and EM software tools.

**Chapter 6 Mapping Quality Factors With Metrics And Quality Assurance Framework:** In this chapter a discussion about mapping of software metrics with each quality factor and sub-factor is given in detail. Based on the relationship of metrics with quality sub-factors the software quality assurance framework is developed.

**Chapter 7 Conclusions and Future Scope:** Finally, the conclusions and future scope of the study is presented.

Conclusively, the **Bibliography** is presented.