CHAPTER 1

INTRODUCTION

The prominent goal of software engineering is building software systems that function as expected and are of high quality with minimum cost. The concept of quality in a software system hinges on various facets, such as how many defects it contains, how easy it is for developers to inadvertently inject faults into the system, and the difficulty involved in modifying and understanding all or parts of the system. Ensuring quality is important as for one thing; it naturally ties with the cost of developing and maintaining the system. Over the past few decades there has been much research on understanding what is meant by quality, and more importantly, how satisfactory quality can be achieved and maintained.

The importance of quality software is no longer an advantage but a necessary factor as software error can have in terms of life, financial loss, or time delays. No doubt that the software quality can make or break a company. Unfortunately, most companies not only fail to deliver a quality product to their customers, but also fail to understand the attributes of a quality product. Traditional software metrics used to evaluate the product characteristics such as size, complexity, performance and quality is switched to rely on some fundamentally different attributes like encapsulation, inheritance and polymorphism, which are inherent in object-orientation. This switching led to the definition of many metrics proposed by various researchers and practitioners to measure the object oriented attributes.
1.1 METRICS AND SOFTWARE QUALITY

Software metrics have become essential in some disciplines of software engineering. In forward engineering they are used to measure software quality and to estimate the cost and effort of software projects. In the field of software evolution, metrics can be used for identifying stable or unstable parts of software systems, as well as identifying where refactoring can be applied or have been applied, and detecting increases or decreases of quality in the structure of evolving software systems. In the field of software re-engineering and reverse engineering, metrics are used for assessing the quality and complexity of software systems, and also to get a basic understanding and provide clues about sensitive parts of software systems.

Software quality needs to be a continuous process. A continuous quality process ensures quality tasks that are not only deployed across every stage of the Software Development Life Cycle (SDLC), but also ingrained into the team's workflow. It can be achieved by taking a policy-based approach that embeds automated policy monitoring "sensors" across the SDLC. Writing code without heed for quality and security and then later trying to identify and remove all of the application's defects is not only resource-intensive, but it's also largely ineffective. To have any chance of exposing all of the defects that may be nested throughout the application, one would need to identify every single path through the application and then rigorously test each and every one. Moreover, any problem found at this point would be difficult to fix, considering that the effort, cost, and time required to fix each bug increases exponentially as the development process progresses. Most important, the bug-finding approach fails to address the root cause of the problem. Building quality and security into an application involves designing and implementing the application according to a policy in order to reduce the risk of defects and security vulnerabilities, then verifying that the policy is implemented and operating correctly. The key to quality is
implementing and enforcing a process that builds quality into the product not searching for better ways to find and fix defects. By implementing the following software verification methods as part of a continuous quality process, Quality Assurance (QA) time can be significantly optimized:

1. **Static analysis:** QA time is largely wasted chasing after simple defects — defects that could easily be prevented by ensuring that developers write code according to the team's policy for establishing code security, reliability, performance, and maintainability. Static analysis is the shortest path to implementing proper, consistent group behavior. With appropriate implementation and training, team members will come to accept the policies and will adopt policy adherence as a natural part of their day-to-day workflow.

2. **Code review during development:** Considerable QA time is also spent trying to identify functional defects — an instance where the application does not do what it's supposed to. The only way to identify such defects is with the human brain. However, having the human brain try to find those defects after development is difficult because so much disparate information must be consumed at once. It's better to conduct peer reviews during development, which is when functional defects are fastest, easiest, and least costly to identify and resolve. Again, by reducing the amount of defects that need to be addressed during QA, you reduce the length and cost of the QA cycle.

3. **Automated regression testing:** The ability to create an automated regression test suite and run it completely automatically is essential for overlapping QA with development. Such a test suite should leverage technologies
including static analysis, unit testing, and protocol testing. Moreover, it should be driven by an automated infrastructure so that the test suite runs on its own each night (after the build) and immediately alerts the team if modifications introduced have an unexpected or negative impact to the existing functionality. With that type of automated regression testing, QA can focus on overseeing the execution and extension of this test suite. Software Quality enables innovation. A business must continuously ensure a key focus on quality in the development and maintenance of software systems upon which the business relies because it is this quality that makes it possible to react, adapt, and deploy quickly. If there is any breakdown, any inefficiency, any lack of quality, and then the organization will fail to reach the market in a timely fashion and the organization will find itself at a competitive disadvantage.

There are several reasons for the high cost of poor quality, depending on the root cause of the problem. For example, poor problem domain analysis and thus poor requirements - leads to unplanned and costly rework. Eleventh-hour validation through testing ruins project predictability, inevitably extending the schedule and thus the budget. And there is the opportunity cost of operational downtime due to reliability or performance problems - the inability of customers to access your system or of the business to do its work. The business benefits of quality are both broad and deep. Not only does quality facilitate innovation by increasing predictability, lowering risk, and reducing rework; it also serves as a differentiator, since it enables a business to set itself apart from its competitors. Most importantly, continuously ensuring quality always cost less than ignoring quality. Quality is free when it is done right.
1.2 QUALITY MODELS

Software quality is the extent to which an industry-defined set of desirable features are incorporated into a product so as to enhance its lifetime performance. Software Quality is defined as conformance of the produced software to stated and implied needs [ISO/IEC 9126-1]. In order to understand and measure quality, scientists often built models of how quality characteristics relate to each other. A quality model is a set of quality characteristics and relationships between them, which provides the basis for evaluating product software quality and specifying its requirements [ISO/IEC 9126-1]. Current quality models such as ISO/IEC 9126 contain numerous metrics and their full usage requires significant evaluation effort per product.

There are a number of quality models in software engineering literature, each one of these quality models consists of a number of quality characteristics (or factors, as called in some models). These quality characteristics could be used to reflect the quality of the software product from the view of that characteristic. Selecting which one of the quality models to use is a real challenge. In this section, we will discuss the contents of the following quality models:

1. McCall’s Quality Model.
2. Boehm’s Quality Model.
3. Dromey's Quality Model.
4. FURPS Quality Model.
5. ISO 9126 Quality Model.

1.2.1 McCall’s Quality Model
McCall’s Quality Model is one of the most known quality models in the software engineering literature. This model originates from the US military and is primarily aimed towards the system developers and the system development process. Using this model, McCall attempts to bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the users’ views and the developers’ priorities. The structure of the McCall’s quality model consists of three major perspectives (types of quality characteristics) for defining and identifying the quality of a software product, and each of these major perspectives consists of a number of quality factors. Each of these quality factors has a set of quality criteria, and each quality criteria could be reflected by one or more metrics. The contents of the three major perspectives are the following:

1.2.1 Product Revision

Product Revision is about the ability of the product to undergo changes, and it includes the following:

- **Maintainability**: the effort required to locate and fix a fault in the program within its operating environment.
- **Flexibility**: The ease of making changes required by changes in the operating environment.
- **Testability**: The ease of testing the program, to ensure that it is error-free and meets its specification.

1.2.1.2 Product Operations

Product Operations is about the characteristics of the product in operation. The quality of the product operations depends on:
- **Correctness**: the extent to which a program fulfils its specification.

- **Reliability**: the system ability not to fail.

- **Efficiency**: it further categorized into execution efficiency and storage efficiency and generally meaning the use of resources, e.g. processor time, storage.

- **Integrity**: the protection of the program from unauthorized access.

- **Usability**: the ease of the use of the software.

### 1.2.1.3 Product Transition

Product Transition is about the adaptability of the product to new environments. It is all about:

- **Portability**: the effort required to transfer a program from one environment to another.

- **Reusability**: the ease of reusing software in a different context.

- **Interoperability**: the effort required to couple the system to another system

McCall’s Quality Model consists of 11 quality factors to describe the external view of the software (from the users’ view), 23 quality criteria to describe the internal view of the software (from the developer’s view) and a set of Metrics which are defined and used to provide a scale and method for measurement. Table 1 presents two of the three major perspectives and their corresponding quality factors and quality criteria.
The main objective of the McCall’s Quality Model is that the quality factors structure should provide a complete software quality picture Kitchenhamet et al. (1996). The actual quality metric is computed by answering “yes” and “no” questions. However, if answering equally amount of “yes” and the first two levels of the [McCall] quality model are presented on the following figure 1.1.

Table 1.1 The contents of McCall’s quality model - product revision and product operations

<table>
<thead>
<tr>
<th>Major perspectives</th>
<th>Quality factor</th>
<th>Quality criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Revision</td>
<td>Maintainability</td>
<td>Simplicity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conciseness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Self descriptiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modularity</td>
</tr>
<tr>
<td></td>
<td>Flexibility</td>
<td>Selfdescriptiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expandability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generality</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>Simplicity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instrumentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selfdescriptiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modularity</td>
</tr>
<tr>
<td>Product operations</td>
<td>Correctness</td>
<td>Traceability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Completeness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Consistency</td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>Execution efficiency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Storage efficiency</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>Consistency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accuracy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error tolerance</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>Access control</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access audit</td>
</tr>
<tr>
<td></td>
<td>Usability</td>
<td>Operability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Training</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Communicativeness</td>
</tr>
</tbody>
</table>
The main idea behind McCall’s model is assessment of relationships among external quality factors and product quality criteria. The external quality is related to the product and is measured by the customers, while the internal quality is quality experienced during the development and it is measured by the programmers.

Figure 1.1 McCall quality model hierarchical representations
1.2.2 **Boehm’s Quality Model**

Boehm introduced his quality model to automatically and quantitatively evaluate the quality of software. This model attempts to qualitatively define the quality of software by a predefined set of attributes and metrics. It consists of high-level characteristics, intermediate-level characteristics and lowest-level (primitive) characteristics which contribute to the overall quality level (see Figure 1.2).

In this model, the high-level characteristics represent basic high-level requirements of actual use to which evaluation of software quality could be put. In its high-level, there are three characteristics.

- **As-is utility**: to address how well, easily, reliably and efficiently can as-is utility use the software product?
- **Maintainability**: to address how easy is it to understand, modify and retest the software product?
- **Portability**: to address if can I still use the software product when the environment has been changed?

Figure 1.2 shows the contents of the Boehm’s quality model in the three levels, high-level, intermediate-level and lowest-level characteristics. In addition, it is noted that there is a number of the lowest-level characteristics which can be related to more than one intermediate-level characteristics, for example, the ‘Self Contentedness’ primitive characteristic could be related to the ‘reliability’ and ‘portability’ primitive characteristics. The primitive characteristics can be used to provide the foundation for defining quality metrics.
Boehm defined the ‘metric’ as “a measure of extent or degree to which a product possesses and exhibits a certain (quality) characteristic”. In
the intermediate level characteristic, there are seven quality characteristics that together represent the qualities expected from a software system.

- **Portability**: The software can be operated easily and well on computer configurations other than its current one.
- **Reliability**: The software can be expected to perform its intended functions satisfactorily.
- **Efficiency**: The software fulfills its purpose without waste of resources.
- **Usability**: The software is reliable, efficient and human-engineered.
- **Testability**: The software facilitates the establishment of verification criteria and supports evaluation of its performance.
- **Understandability**: The software purpose is clear to the inspector.
- **Flexibility**: The software facilitates the incorporation of changes, once the nature of the desired change has been determined.

### 1.2.3 Dromey’s Quality Model

This quality model has been presented by Dromey et al (1995, 1996). It is a product based quality model that recognizes that quality evaluation differs for each product and that a more dynamic idea for modeling the process is needed to be wide enough to apply for different systems. Basically this model consists of four software product properties and for each property there is a number of quality attributes. In addition, figure 3 shows the contents of the Dromey's quality model.
1.2.4 FURPS Quality Model

The FURPS model was originally presented by Robert Grady, then it has been extended by IBM Rational Software into FURPS+, where the ‘+’ indicates such requirements as design constraints, implementation requirements, interface requirements and physical requirements Jacobson et al (1999).

In this quality model, the FURPS stands for the following five characteristics:

- **Functionality**: it may include feature sets, capabilities, and security.
- **Usability**: it may include human factors, aesthetics, consistency in the user interface, online and context sensitive help, wizards and agents, user documentation, and training materials.
- **Reliability**: it may include frequency and severity of failure, recoverability, predictability, accuracy, and mean time between failures (MTBF).

- **Performance**: it imposes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage.

- **Supportability**: it may include testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, installability, and localizability.

### 1.2.5 ISO 9126 Quality Model

In 1991, the ISO published its first international consensus on the terminology for the quality characteristics for software product evaluation; this standard was called as Software Product Evaluation - Quality Characteristics and Guidelines for Their Use (ISO 9126). From 2001 to 2004, the ISO published an expanded version, containing both the ISO quality models and inventories of proposed measures for these models. The current version of the ISO 9126 series now consists of one International Standard (IS) and three Technical Reports (TRs):


1. Internal and external quality model.

2. Quality in use model.

Figure 1.4 ISO 9126 Quality Model – Internal and External Attributes

Figure 1.5 ISO 9126 Quality Model – Internal and External Attributes

The first part of the two-part quality model determines six characteristics in which they are subdivided into twenty-seven sub-characteristics for internal and external quality, as in Figure 1.4. These sub-characteristics are a result of internal software attributes and are noticeable externally when the software is used as a part of a computer system. The
second part of the two-part model indicates four qualities in use characteristics, as in Figure 1.5.

1.3 SOFTWARE METRICS AND COMPLEXITY

Software metrics evaluate different aspects of the complexity of a software product. Software complexity was originally defined as a “measurement of the resources that must be expended in developing, testing, debugging, maintenance, user training, operation, and correction of software products”. Complexity has been characterized in terms of seven different levels, the correlation and interdependence of which will determine the overall level of complexity in a software product Gonzalez et al (1995). The levels are as follows:

- Control Structure
- Module Coupling
- Algorithm
- Code
- Nesting
- Module Cohesion
- Data Structure

However, most metrics measure only one software complexity factor. These foundations of complexity will determine the internal quality of a product. Internal quality measures are those which are performed in terms of the software product itself and are measurable both during and after the creation of the software product. They have however, no inherent, practical meaning within themselves. To give them meaning they must be characterized in terms of the product's external quality.
External quality measures are evaluated with respect to how a product relates to its environment and are deemed to be inherently meaningful; such examples would be the maintainability or testability of a product. It should be noted that good internal quality is a requirement for good external quality. Figure 1.6 illustrates the software quality model which depicts the relationship between these measures. Much research has contributed models and measures of both internal software quality attributes and external attributes of a design.

Although the relationships between these attributes is for the most part intuitive, e.g., more complex code will require greater effort to maintain, the precise functional form of those relationships can be less clear and is the subject of intense practical and research concern Darcy et al (2005). Empirical validation aims at demonstrating the usefulness of a measure in practice and is, therefore, a crucial activity to establish the overall validity of a measure Basili et al (1996). Therefore it is the belief of the author that a well-designed empirical study serves to clarify and strengthen the observed relationships.

The earliest software measure, which was proposed in the late 1960s, is the Source Lines of Code (SLOC) metric, which is still used today. It is used to measure the amount of code in a software program. It is typically used to estimate the amount of effort that will be required to develop a program, as well as to estimate productivity or effort once the software is produced. Two major types of SLOC measures exist: physical SLOC and logical SLOC. Exact definitions of these measures vary.

The most common definition of physical SLOC is a count of “non-blank, non-comment lines” in the text of the program's source code. Logical SLOC measures attempt to measure the number of “statements”, however their specific definitions are tied to specific computer languages. Therefore, it is much easier to create tools that measure physical SLOC, and physical
SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions.

Figure 1.6 The software quality model shows how different measures of internal quality can characterize the overall quality of a software product

There are a number of drawbacks of using a crude measure such as LOC as a surrogate measure for different notions of program size such as effort, functionality and complexity. The need for more discriminating measures became especially urgent with the increasing diversity of
programming languages, as LOC in an assembly language is not comparable in effort, functionality, or complexity to an LOC in a high-level language.

There was an increase in the number of different complexity metrics defined. Some of the more prevalent ones were Halstead's software science metrics, which made an attempt to capture notions of size and complexity beyond simply counting lines of code. Although the work has had a lasting impact they are principally regarded as an example of confused and inadequate measurements.

McCabe defined a measure known as Cyclomatic Complexity. It may be considered as a broad measure of soundness and confidence for a program. It measures the number of linearly-independent paths through a program module and it is intended to be independent of language and language format.

Function points, are a measure of the size of computer applications and the projects that build them. The size is measured from a functional, or user, point of view. It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application. The original metric has been augmented and refined to cover more than the original emphasis on business-related data processing. However as object-oriented techniques became more prevalent there was an increasing need for metrics that could correctly evaluate their properties.

## 1.4 SOFTWARE METRICS AND REUSABILITY

Reuse based software development is one of the most promising technologies in software development. The software development that utilizes the software reuse gets the benefits of reduction in the development as well as testing cost and time. It is possible to create customized systems economically
by building only the parts that are application-specific. Unnecessary reinvention of technology is thereby avoided. A reusable software component can be anything such as program code, design specifications, plans, documentation, expertise and experience, and any information used to create software and software documentation. A classical, but largely unrealized, goal of software engineering is software component technologies. Such technologies are envisioned to exploit large-scale reuse, and elevate the granularity of programming to the subsystem level. The major steps in the Software Reuse Procedure includes,

- **Component extraction**- Extracting the component with high reuse potential and high quality from existing software systems.

- **Component standardization**- Packaging a reusable software component in a standard format.

- **Component classification**– Classifying the different reusable components into libraries based on their attributes.

- **Component retrieval**– Reusing the suitable component out of the library based on the attributes needed.

- **Component adaptation**– Adapting the reuse component to the new application.

The key step is to extract the software components with high reuse potential and high quality i.e. Component Extraction, as this step may affect the steps that follow directly. The most serious problem with the Component Extraction is an overly high degree of dependency throughout the whole software product. It is therefore difficult to reuse selected packages or subsystems in another software system without making many changes. If we pick up one package for reusing, we get almost the whole system because of these direct or indirect dependencies. The recompilation of large parts of the
system (eventually the whole system itself) for only minor modifications in any of the subsystems indicates the effect of dependency between the different subsystems. Therefore, it is hard to extract a single subsystem as a reusable component out of the whole system. This dependency is called Coupling. Measuring the coupling will give a chance to predict the reusability of a module or to predict the change needed to make it reusable.

Object oriented technology uses objects and not algorithms as its fundamental building blocks, the approach to software metrics for object oriented programs must be different from the standard metrics set. Metrics, such as lines of code and cyclomatic complexity, have become accepted as standard for traditional functional/procedural programs and were used to evaluate object-oriented environments at the beginning of the object-oriented design revolution. However, traditional metrics for procedural approaches are not adequate for evaluating object-oriented software, primarily because they are not designed to measure basic elements like classes, objects, polymorphism, and message-passing.

Even when adjusted to syntactically analyze object-oriented software they can only capture a small part of such software and thus provide a weak quality indication. Since this time there have been many proposed object-oriented metrics in the literature. The question now is, which object-oriented metrics should a project use? As the quality of object-oriented software, like other software, is a complex concept there can be no single, simple measure of software quality acceptable to everyone. To assess or improve software quality in you must define the aspects of quality in which you are interested, and then decide how you are going to measure them. By defining quality in a measurable way, you make it easier for other people to understand your viewpoint and relate your notions to their own Kitchemham et al (1996). Some of the seminal methods of evaluating an
object-oriented design are through the use of measures for coupling and cohesion.

A large number of metrics have been proposed to measure object-oriented design quality. Design metrics can be classified into two categories; static and runtime/dynamic. Static metrics measure what may happen when a program is executed and are said to quantify different aspects of the complexity of the source code. Run-time metrics measure what actually happens when a program is executed. They evaluate the source code's run-time characteristics and behavior as well as its complexity.

Despite the rich body of research and practice in developing design quality metrics, there has been less emphasis on run-time metrics for object-oriented designs mainly due to the fact that a run-time code analysis is more expensive and complex to perform. However, due to polymorphism, dynamic binding, and the common presence of unused (dead) code in software, static coupling and cohesion measures do not perfectly reflect the actual situation taking place amongst classes at run-time.

The complex dynamic behavior of many real-time applications motivates a shift in interest from traditional static metrics to run-time metrics. In this work, we investigate whether useful information on design quality can be provided by run-time measures of coupling and cohesion over and above that which is given by simple static measures. This will determine if it is worthwhile to continue the investigation into run-time coupling and cohesion metrics and their relationship with the external quality.

1.5 FACTORS INFLUENCING SOFTWARE METRICS

This section discusses factors which affect software metrics, including coverage and object-level behavior. The relationship with software testing is also discussed.
1.5.1 Coverage

When relating static and run-time measures, it is important to have a thorough understanding of the degree to which the analyzed source code corresponds to the code that is actually executed. In this thesis, this relationship is studied using instruction coverage measures with regard to the influence of coverage on the relationship between static and dynamic metrics. It is proposed that coverage results have a significant influence on the relationship and thus should always be a measured, recorded factor in any such comparison.

1.5.2 Metrics and Object Behavior

Of late little work has been done on the analysis of code at the object-level that is the use of metrics to identify specific object behaviors. We identify this behavior through the use of run-time object-level coupling metrics. Run-time object-level coupling quantifies the level of dependencies between objects in a system whereas run-time class-level coupling quantifies the level of dependencies between the classes that implement the methods or variables of the caller object and the receiver object Arisholm et al (2004). The class of the object sending or receiving a message may be different from the class implementing the corresponding method due to the impact of inheritance. We also investigate the ability of run-time cohesion measures to predict such behavior.

1.5.3 Metrics and Software Testing

Testing is one of the most effort-intensive activities during software development. Much research is directed toward developing new and improved fault detection mechanisms. A number of papers have investigated the relationships between static design metrics and the detection of faults in
object-oriented software Basili et al (1996), Briand et al (2002). However, to date no work has been conducted on the correlation of run-time coupling metrics and fault detection. In this thesis, we investigate whether measures for run-time coupling are good predictors of fault-proneness, an important software quality attribute.

The key to ensuring the product satisfies expected standards is measurement. Measurement provides a means to objectively assess the properties of the product and evaluate whether it meets its desired qualities. Thus, measurement in the software realm must be done with the same purpose of quantifying the aspects of software that affect its overall quality.

There is an increasing awareness on the importance of software measurement within the software engineering community, as well as the necessity of respecting the scientific basis of measurement. However there is little evidence for the latter as there is a tendency for researchers and practitioners to apply software metrics without a full awareness of what they mean. Coupling, which is the measure of the interdependence between parts of a software system (e.g. classes), is one important property for which many metrics have been defined. While it is widely agreed that there is a relationship between high coupling and poor maintainability, current empirical evidence toward this is insufficient to promote a full understanding of this relationship. Part of this is due to the lack of coverage of all forms of connections that comprise coupling. However, there is specific, indirect, form of coupling that manifests between two seemingly unrelated parts of the system through hidden connections. In other words, there is a relationship between indirect coupling and maintainability.

Designing and developing a good quality software product requires efficient measures to accurately monitor the internal software quality attributes, such as coupling, cohesion, size and complexity, throughout the
course of a software development life cycle. Over the years, software metrics have been widely and successfully used to measure such internal quality attributes for object oriented software systems. Such internal quality attributes in turn give a measure of the external quality attributes like maintainability, modifiability, understandability, reusability, and testability etc Chidamber et al (1994), Briand et al (1996), Briand et.al.(2000).