

**EMP5103 – Reliability, Quality and Safety Engineering**

REPORT

Software Quality and Safety

**PRESENTED TO: PROF. B. DHILLON**

|  |  |
| --- | --- |
| **Student Name** | **Student Number** |
| Ferhan Jamal | 100953487 |

**ABSTRACT**

In software engineering software quality has become a topic of major concern. As software is becoming critically important for the organizations to be competitive in its business, the requirement that the software is highly supportive for the organization in achieving its goals means that the software should have high utility and user quality. It has also been recognized that software maintenance is becoming the main activity in software work. With the growing collection of software in organizations this cost is becoming substantial. The amount of maintenance needed and the effort needed to perform a certain maintenance task is critically dependent on the technical quality of software resulting from the software development process.

TABLE OF CONTENTS

[1. INTRODUCTION 1](#_Toc319621024)

[1.1 SOFTWARE FUNCTIONAL QUALITY 2](#_Toc319621025)

[1.2 SOFTWARE STRUCTURAL QUALITY 2](#_Toc319621026)

[2. SOFTWARE QUALITY MEASUREMENT 2](#_Toc319621027)

[3. SOFTWARE QUALITY 3](#_Toc319621028)

[1.3 RISK MANAGEMENT: 3](#_Toc319621029)

[1.4 COST MANAGEMENT: 4](#_Toc319621030)

[4. ANALYSIS OF SOFTWARE QUALITY ATTRIBUTES 4](#_Toc319621031)

[5. MEASURING RELIABILITY 6](#_Toc319621032)

[6. MEASURING EFFICIENCY 7](#_Toc319621033)

[7. MEASURING SECURITY 7](#_Toc319621034)

[8. MEASURING MAINTAINABILITY 8](#_Toc319621035)

[9. MEASURING SIZE 9](#_Toc319621036)

[10. IDENTIFYING CRITICAL PROGRAMMING ERRORS 10](#_Toc319621037)

[1.5 RELIABILITY 11](#_Toc319621038)

[1.6 EFFICIENCY 11](#_Toc319621039)

[1.7 SECURITY 11](#_Toc319621040)

[1.8 MAINTAINABILITY 11](#_Toc319621041)

[11. SOFTWARE RELIABILITY 11](#_Toc319621042)

[12. SOFTWARE QUALITY ASSURANCE 14](#_Toc319621043)

[1.9 PRACTICES DONE BY COMPANIES 14](#_Toc319621044)

[1.10 IMPLEMENTATION OF QUALITY ASSURANCE 16](#_Toc319621045)

[1.10.1 Identifying if software is vulnerable 16](#_Toc319621046)

[1.10.2 How to protect 16](#_Toc319621047)

[1.11 METRICS 17](#_Toc319621048)

[1.11.1 How to identify if software is vulnerable 17](#_Toc319621049)

[1.11.2 How to protect yourself 17](#_Toc319621050)

[1.12 TESTING ACTIVITIES 18](#_Toc319621051)

[1.12.1 How to identify if software is vulnerable 18](#_Toc319621052)

[1.12.2 How to protect yourself 18](#_Toc319621053)

[13. VERIFICATION AND VALIDATION OF SOFTWARES 18](#_Toc319621054)

[1.13 VERIFICATION OF SOFTWARE 19](#_Toc319621055)

[1.13.1 Terms Involved in Verification 19](#_Toc319621056)

[1.14 VALIDATION OF SOFTWARE 20](#_Toc319621057)

[1.14.1 Terms Used in Validation Process 21](#_Toc319621058)

[14. REVIEWS AND AUDITS OF SOFTWARE 22](#_Toc319621059)

[1.15 STANDARD / PROCEDURE DESCRIPTION 22](#_Toc319621060)

[1.15.1 When to do what 22](#_Toc319621061)

[1.15.2 Subject matter 22](#_Toc319621062)

[1.15.3 Means of initiation 22](#_Toc319621063)

[1.16 REVIEW STATUS REGISTER 23](#_Toc319621064)

[1.17 MANDATORY AND OPTIONAL PRACTICE 23](#_Toc319621065)

[1.17.1 Mandatory 23](#_Toc319621066)

[1.17.2 Optional (recommended) 24](#_Toc319621067)

[1.17.3 Optional 24](#_Toc319621068)

[1.18 PARTIAL OR COMPLETE REVIEWS 24](#_Toc319621069)

[1.19 MANAGEMENT REVIEW PROCESS 25](#_Toc319621070)

[1.20 INITIATION 26](#_Toc319621071)

[1.20.1 Initiating event 26](#_Toc319621072)

[1.20.2 Staffing 26](#_Toc319621073)

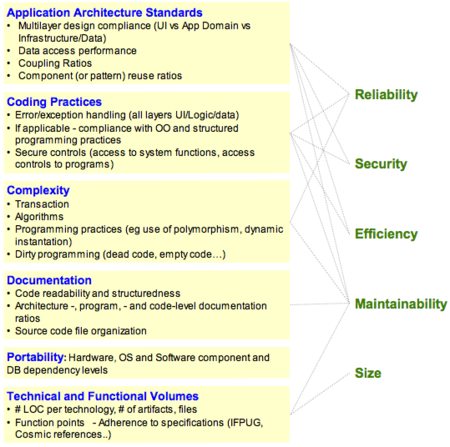
[1.20.3 Notification 27](#_Toc319621074)

[1.21 INPUT MATERIAL 27](#_Toc319621075)

[15. CONCLUSION 29](#_Toc319621076)

# INTRODUCTION

Software quality measurement is about quantifying to what extent software or system possesses desirable characteristics. This can be done via qualitative or quantitative means or a mix of both. In both cases, for each desirable characteristic, there are a set of measurable attributes the existence of which in a piece of software or system tend to be correlated and associated to this characteristic. More precisely, using the [Quality Function Deployment](http://en.wikipedia.org/wiki/Quality_Function_Deployment) approach, these measurable attributes are the "Hows" that needs to be enforced to enable the "whats" in the Software Quality definition above.

[](http://en.wikipedia.org/wiki/File:SoftwareQualityCharacteristicAttributeRelationship.png)The structure, classification and terminology of attributes and metrics applicable to software quality management had been derived from the [ISO 9126-3](http://en.wikipedia.org/wiki/ISO_9126) and the subsequent [ISO 25000:2005](http://webstore.iec.ch/preview/info_isoiec25000%7Bed1.0%7Den.pdf) quality models. The main focus is on internal structural quality. Subcategories have been created to handle specific areas like business application architecture and technical characteristics such as data access and manipulation and the notion of transactions.

The dependence tree between software quality characteristics and their measurable attributes is represented in the diagram, where each of the 5 characteristics that matter for the user or owner of the business system depends on measurable attributes as under.

* Application Architecture Practices
* Coding Practices
* Application Complexity
* Documentation
* Portability
* Technical & Functional Volume

## SOFTWARE FUNCTIONAL QUALITY

The functional quality of software reflects how well it complies with or conforms to a given design, based on [functional requirements](http://en.wikipedia.org/wiki/Functional_requirements) or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile [product](http://en.wikipedia.org/wiki/Product_(business)); [[1]](http://en.wikipedia.org/wiki/Software_quality#cite_note-0)

## SOFTWARE STRUCTURAL QUALITY

This quality refers to how it meets [non-functional requirements](http://en.wikipedia.org/wiki/Non-functional_requirements) that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, in effect how its architecture adheres to sound principles of [software architecture](http://en.wikipedia.org/wiki/Software_architecture). In contrast, functional quality is typically enforced and measured through [software testing](http://en.wikipedia.org/wiki/Software_testing).

In the past, the structure, classification and terminology of attributes and metrics applicable to software quality management were derived or extracted from the [ISO 9126-3](http://en.wikipedia.org/wiki/ISO_9126) and the subsequent [ISO 25000:2005](http://webstore.iec.ch/preview/info_isoiec25000%7Bed1.0%7Den.pdf) quality models. Based on these models, the software structural quality characteristics have been clearly defined by the Consortium for IT Software Quality ([CISQ](http://en.wikipedia.org/wiki/CISQ)), which is an independent organization founded by the Software Engineering Institute [(SEI)](http://www.sei.cmu.edu/) at [Carnegie Mellon University](http://en.wikipedia.org/wiki/Carnegie_Mellon_University), and the Object Management Group [(OMG)](http://www.omg.org/).

CISQ has defined 5 major desirable characteristics needed for a piece of software to provide [business value](http://en.wikipedia.org/wiki/Business_value): Reliability, Efficiency, Security, Maintainability and Size.

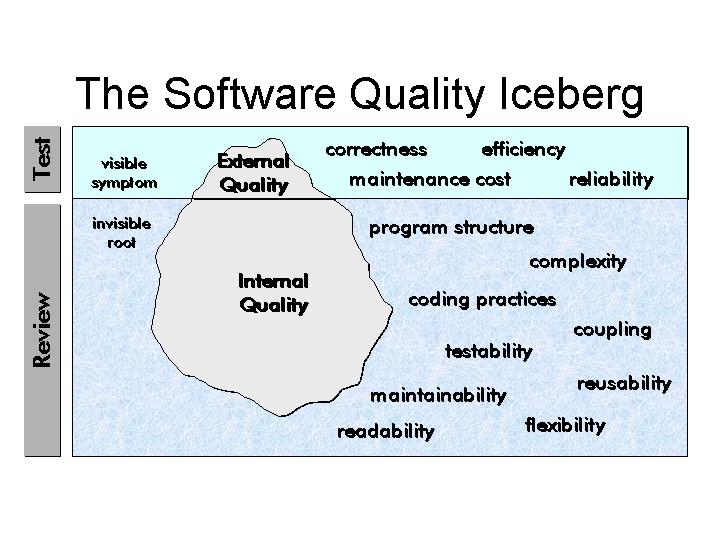
# [SOFTWARE QUALITY MEASUREMENT](http://en.wikipedia.org/wiki/Software_quality#SoftwareQualityMeasurement)

Quality measurement is about quantifying to what extent a software or system rates along each of these five dimensions. An aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. This view of software quality being positioned on a linear continuum has to be supplemented by the analysis of [Critical Programming Errors](http://en.wikipedia.org/wiki/Software_quality#CriticalProgrammingErrors) that under specific circumstances can lead to catastrophic outages or performance degradations that make a given system unsuitable for use regardless of rating based on aggregated measurements.

# SOFTWARE QUALITY

Software engineering has evolved to a level of maturity that makes it not only possible but also necessary to measure quality software for at least two reasons.

## RISK MANAGEMENT:

Software failure has caused more than inconvenience. Software errors have caused human fatalities. The causes have ranged from poorly designed user interfaces to direct [programming errors](http://en.wikipedia.org/wiki/Programming_error). An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper.[[2]](http://en.wikipedia.org/wiki/Software_quality#cite_note-1) This resulted in requirements for the development of some types of software, particularly and historically for [software embedded](http://en.wikipedia.org/wiki/Embedded_software) in medical and other devices that regulate critical infrastructures: "[Engineers who write embedded software] see Java programs stalling for one third of a second to perform garbage collection and update the user interface, and they envision airplanes falling out of the sky.".[[3]](http://en.wikipedia.org/wiki/Software_quality#cite_note-2) In the United States, within the [Federal Aviation Administration (FAA)](http://en.wikipedia.org/wiki/Federal_Aviation_Administration), the [Aircraft Certification Service](http://www.faa.gov/aircraft/air_cert/design_approvals/air_software%7CFAA) provides software programs, policy, guidance and training, focus on software and Complex Electronic Hardware that has an effect on the airborne product (a “product” is an aircraft, an engine, or a propeller)".

## COST MANAGEMENT:

As in any other fields of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs. Industry data demonstrate that poor application structural quality in core [business applications](http://en.wikipedia.org/wiki/Business_application) (such as [Enterprise Resource Planning (ERP)](http://en.wikipedia.org/wiki/Enterprise_resource_planning), [Customer Relationship Management (CRM)](http://en.wikipedia.org/wiki/Customer_relationship_management) or large [transaction processing](http://en.wikipedia.org/wiki/Transaction_processing) systems in financial services) results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations [[4]](http://en.wikipedia.org/wiki/Software_quality#cite_note-3)). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems.

However, the distinction between measuring and improving software quality in an embedded system (with emphasis on risk management) and software quality in business software (with emphasis on cost and maintainability management) is becoming somewhat irrelevant. Embedded systems now often include a user interface and their designers are as much concerned with issues affecting usability and user productivity as their counterparts who focus on business applications. The latter are in turn looking at ERP or CRM system as a corporate nervous system whose uptime and performance are vital to the well-being of the enterprise. This convergence is most visible in mobile computing: a user who accesses an ERP application on their Smartphone is depending on the quality of software across all types of software layers.

Both types of software now use multi-layered technology stacks and complex architecture so software quality analysis and measurement have to be managed in a comprehensive and consistent manner, decoupled from the software's ultimate purpose or use. In both cases, engineers and management need to be able to make rational decisions based on measurement and fact-based analysis in adherence to the precept.

# ANALYSIS OF SOFTWARE QUALITY ATTRIBUTES

Many of the existing software measures count structural elements of the application that result from parsing the source code such individual instructions (Park, 1992),[[8]](http://en.wikipedia.org/wiki/Software_quality#cite_note-7)tokens (Halstead, 1977),[[9]](http://en.wikipedia.org/wiki/Software_quality#cite_note-8) control structures (McCabe, 1976), and objects (Chidamber&Kemerer, 1994).[[10]](http://en.wikipedia.org/wiki/Software_quality#cite_note-9)

Software quality measurement is about quantifying to what extent a software or system rate along these dimensions. The analysis can be performed using a qualitative, quantitative approach or a mix of both to provide an aggregate view (using for example weighted averages that reflect relative importance between the factors being measured).

This view of software quality on a linear continuum has to be supplemented by the identification of discrete [Critical Programming Errors](http://en.wikipedia.org/wiki/Software_quality#CriticalProgrammingErrors). These vulnerabilities may not fail a test case, but they are the result of bad practices that under specific circumstances can lead to catastrophic outages, performance degradations, security breaches, corrupted data, and myriad other problems (Nygard, 2007)[[11]](http://en.wikipedia.org/wiki/Software_quality#cite_note-10) that makes a given system de facto unsuitable for use regardless of its rating based on aggregated measurements. A well known example of vulnerability is the Common Weakness Enumeration [[12]](http://en.wikipedia.org/wiki/Software_quality#cite_note-11) a repository of vulnerabilities in the source code that make applications exposed to security breaches.

The measurement of critical application characteristics involves measuring structural attributes of the application's architecture, coding, in-line documentation, as displayed in the picture above. Thus, each characteristic is affected by attributes at numerous levels of abstraction in the application and all of which must be included calculating the characteristic’s measure if it is to be a valuable predictor of quality outcomes that affect the business. The layered approach to calculating characteristic measures displayed in the figure above was first proposed by Boehm and his colleagues at TRW (Boehm, 1978)[[13]](http://en.wikipedia.org/wiki/Software_quality" \l "cite_note-12) and is the approach taken in the ISO 9126 and 25000 series standards. These attributes can be measured from the parsed results of a static analysis of the application source code. Even dynamic characteristics of applications such as reliability and performance efficiency have their causal roots in the static structure of the application.

Structural quality analysis and measurement is performed through the analysis of the [source code](http://en.wikipedia.org/wiki/Source_code), the [architecture](http://en.wikipedia.org/wiki/Software_architecture), [software framework](http://en.wikipedia.org/wiki/Software_framework), [database schema](http://en.wikipedia.org/wiki/Database_schema) in relationship to principles and standards that together define the conceptual and logical architecture of a system. This is distinct from the basic, local, component-level code analysis typically performed by [development tools](http://en.wikipedia.org/wiki/Development_tool) which are mostly concerned with implementation considerations and are crucial during [debugging](http://en.wikipedia.org/wiki/Debugging) and [testing](http://en.wikipedia.org/wiki/Software_testing) activities.

# MEASURING RELIABILITY

The root causes of poor reliability are found in a combination of non- compliance with good architectural and coding practices. This non-compliance can be detected by measuring the static quality attributes of an application. Assessing the static attributes underlying an application’s reliability provides an estimate of the level of business risk and the likelihood of potential application failures and defects the application will experience when placed in operation.

Assessing reliability requires checks of at least the following software engineering best practices and technical attributes:

* Application Architecture Practices
* Coding Practices
* Complexity of algorithms
* Complexity of programming practices
* Compliance with Object-Oriented and Structured Programming best practices (when applicable)
* Component or pattern re-use ratio
* Dirty programming
* Error & Exception handling (for all layers - GUI, Logic & Data)
* Multi-layer design compliance
* Resource bounds management
* Software avoids patterns that will lead to unexpected behaviors
* Software manages data integrity and consistency
* Transaction complexity level

Depending on the application architecture and the third-party components used (such as external libraries or frameworks), custom checks should be defined along the lines drawn by the above list of best practices to ensure a better assessment of the reliability of the delivered software.

# MEASURING EFFICIENCY

As with Reliability, the causes of performance inefficiency are often found in violations of good architectural and coding practice which can be detected by measuring the static quality attributes of an application. These static attributes predict potential operational performance bottlenecks and future scalability problems, especially for applications requiring high execution speed for handling complex algorithms or huge volumes of data.

Assessing performance efficiency requires checking at least the following software engineering best practices and technical attributes:

Application Architecture Practices

Appropriate interactions with expensive and/or remote resources

Data access performance and data management

Memory, network and disk space management

Coding Practices

Compliance with Object-Oriented and Structured Programming best practices (as appropriate)

Compliance with SQL programming best practices

# MEASURING SECURITY

Most security vulnerabilities result from poor coding and architectural practices such as SQL injection or cross-site scripting. These are well documented in lists maintained by CWE, and the SEI/Computer Emergency Center at Carnegie Mellon University.

Assessing security requires at least checking the following software engineering best practices and technical attributes:

* Application Architecture Practices
* Multi-layer design compliance
* Security best practices (Input Validation, SQL Injection, Cross-Site Scripting, etc.
* Programming Practices (code level)
* Error & Exception handling
* Security best practices (system functions access, access control to programs)

# MEASURING MAINTAINABILITY

Maintainability includes concepts of modularity, understandability, changeability, testability, reusability, and transferability from one development team to another. These do not take the form of critical issues at the code level. Rather, poor maintainability is typically the result of thousands of minor violations with best practices in documentation, complexity avoidance strategy, and basic programming practices that make the difference between clean and easy-to-read code vs. unorganized and difficult-to-read code.[[14]](http://en.wikipedia.org/wiki/Software_quality#cite_note-13)

Assessing maintainability requires checking the following software engineering best practices and technical attributes:

* Application Architecture Practices
* Architecture, Programs and Code documentation embedded in source code
* Code readability
* Complexity level of transactions
* Complexity of algorithms
* Complexity of programming practices
* Compliance with Object-Oriented and Structured Programming best practices (when applicable)
* Component or pattern re-use ratio
* Controlled level of dynamic coding
* Coupling ratio
* Dirty programming
* Documentation
* Hardware, OS, middleware, software components and database independence
* Multi-layer design compliance
* Portability
* Programming Practices (code level)
* Reduced duplicated code and functions
* Source code file organization cleanliness

# MEASURING SIZE

Measuring software size requires that the whole source code be correctly gathered, including database structure scripts, data manipulation source code, component headers, configuration files etc. There are essentially two types of software sizes to be measured, the technical size (footprint) and the functional size:

* There are several software technical sizing methods. The most common technical sizing method is number of Lines Of Code (#LOC) per technology, number of files, functions, classes, tables, etc., from which backfiring Function Points can be computed;
* The most common for measuring functional size is Function Point Analysis. Function Point Analysis measures the size of the software deliverable from a user’s perspective. Function Point sizing is done based on user requirements and provides an accurate representation of both size for the developer/estimator and value (functionality to be delivered) and reflects the business functionality being delivered to the customer. The method includes the identification and weighting of user recognizable inputs, outputs and data stores. The size value is then available for use in conjunction with numerous measures to quantify and to evaluate software delivery and performance (Development Cost per Function Point; Delivered Defects per Function Point; Function Points per Staff Month.).

The [Function Point Analysis](http://en.wikipedia.org/wiki/Function_Point_Analysis) sizing standard is supported by the International Function Point Users Group (IFPUG) (www.ifpug.org). It can be applied early in the software development life-cycle and it is not dependent on lines of code like the somewhat inaccurate Backfiring method. The method is technology agnostic and can be used for comparative analysis across organizations and across industries.

Since the inception of Function Point Analysis, several variations have evolved and the family of functional sizing techniques has broadened to include such sizing measures as COSMIC, NESMA, Use Case Points, FP Lite, Early and Quick FPs, and most recently Story Points. However, Function Points has a history of statistical accuracy, and has been used as a common unit of work measurement in numerous application development management (ADM) or outsourcing engagements, serving as the ‘currency’ by which services are delivered and performance is measured.

One common limitation to the Function Point methodology is that it is a manual process and therefore it can be labor intensive and costly in large scale initiatives such as application development or outsourcing engagements. This negative aspect of applying the methodology may be what motivated industry IT leaders to form the Consortium for IT Software Quality (www.it-cisq.org) focused on introducing a computable metrics standard for automating the measuring of software size while the IFPUG www.ifpug.org keep promoting a manual approach as most of its activity rely on FP counters certifications.

# IDENTIFYING CRITICAL PROGRAMMING ERRORS

Critical Programming Errors are specific architectural and/or coding bad practices that result in the highest, immediate or long term, business disruption risk.

These are quite often technology-related and depend heavily on the context, business objectives and risks. Some may consider respect for naming conventions while others – those preparing the ground for a knowledge transfer for example – will consider it as absolutely critical.

Critical Programming Errors can also be classified per CISQ Characteristics. Basic example below:

## RELIABILITY

* Avoid software patterns that will lead to unexpected behavior (Uninitialized variable, null pointers, etc.)
* Methods, procedures and functions doing Insert, Update, Delete, Create Table or Select must include error management
* Multi-thread functions should be made thread safe, for instance services or struts action classes must not have instance/non-final static fields

## EFFICIENCY

* Ensure centralization of client requests (incoming and data) to reduce network traffic
* Avoid SQL queries that don’t use an index against large tables in a loop

## SECURITY

* Avoid fields in servlet classes that are not final static
* Avoid data access without including error management
* Check control return codes and implement error handling mechanisms
* Ensure input validation to avoid cross-site scripting flaws or SQL injections flaws

## MAINTAINABILITY

* Deep inheritance trees and nesting should be avoided to improve comprehensibility
* Modules should be loosely coupled (fan-out, intermediaries, ) to avoid propagation of modifications
* Enforce homogeneous naming conventions

# SOFTWARE RELIABILITY

Hardware reliability is a generally understood and accepted concept with a long history. Early during the much shorter history of software reliability it became apparent to researchers that a division (often perceived to be large) exists between hardware reliability and software reliability. Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However, software reliability is different from hardware reliability in the sense that software does not wear out, burn out, or deteriorate; i.e., its reliability does not decrease with time. Moreover, software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease because of abrupt changes of its operational usage or incorrect modifications to the software. Software is also continuously modified throughout its life cycle. The malleability of software makes it inevitable for us to consider variable failure rates.

At first these differences raised the question of whether reliability theory can be applied to software at all. It was discovered that the distinction between hardware and software is somewhat artificial. Both may be defined in the same way, so that hardware and software component reliabilities can be combined to get system reliability. Traditionally, hardware reliability focused on physical phenomena because failures resulting from these factors are much more likely to occur than design-related failures. It was possible to keep hardware design failures low because hardware was generally less complex logically than software. Besides, hardware design failures had to be kept low because of the large expense involved in retrofitting of manufactured items in the field. However, when hardware tests show that reliability is not within specified design limits because of problems or faults in the original design, a sequence of engineering changes may be necessary to improve reliability. Thus hardware reliability can and has been modeled like software reliability when the failures are the result of design faults, such as the highly visible Pentium floating-point division fault that resulted in massive callbacks in 1994 (Wolfe, [1994](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref108)).

Perhaps the first hardware reliability model that can also be used as a model of reliability for software was developed in 1956 by Northrop Aircraft (Weiss, [1956](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref107)). This model considers complex systems where engineering changes are made to improve system reliability. It was used to determine the level of system reliability, how rapidly reliability was improving, and the expected reliability at the end of the projected development program. Two other early hardware reliability models along similar lines consider the problem of estimating reliability of a system undergoing development testing and changes to correct design deficiencies (Corcoran, et al., [1964](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref12); Barlow and Scheuer, [1966](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref6)).

It is now also generally accepted that the software failure process is random. This randomness is introduced in many ways. The location of design faults within the software is random because the overall system design is extremely complex. The programmers who introduce the design faults are human, and human failure behavior is so complex that it can best be modeled using a random process. Also, the occurrence of failures is dependent on the operational profile, which is defined by input states. It is usually not known which input state will occur next, and sometimes an input state may occur unexpectedly. These events make it impossible to predict where a fault is located or when it will be evoked to cause a failure in a large software system.

In an attempt to unify hardware and software for an overall system reliability, software reliability theory has generally been developed in a way that is compatible with hardware reliability theory, so that system reliability figures may be computed using standard hardware combinatorial techniques (Shooman, [1990](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref95); Lloyd and Lipow, [1984](http://www.mrw.interscience.wiley.com/ese/articles/sof329/bibliography.html#ref58)). However, unlike hardware faults that are mostly physical faults, software faults are design faults that are harder to visualize, classify, detect, and correct. As a result, software reliability is a much more difficult measure to obtain and analyze than hardware reliability. Usually hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of systems complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address non stationary phenomena such as reliability growth or reliability decrease experienced in software. This makes software reliability a challenging problem that requires an employment of several methods to attack. We now describe the theory behind software reliability.

# SOFTWARE QUALITY ASSURANCE

The software quality assurance goal is to confirm the confidentiality and integrity of private user data is protected as the data is handled, stored, and transmitted. The QA testing should also confirm the application cannot be hacked, broken, commandeered, overloaded, or blocked by denial of service attacks, within acceptable risk levels. This implies that the acceptable risk levels and threat modeling scenarios are established up front, so the developers and QA engineers know what to expect and what to work towards.

## PRACTICES DONE BY COMPANIES

* Leverage the available resources like the OWASP Top Ten list, CLASP, or the policy compliance frameworks and the threat modeling processes. These processes will help identify design parameters, establish measurable goals, and ensure that security testing proceeds in a systematic, thorough, and quantified fashion.
* Effective software quality assurance involves three complementary factors: Process, Metrics, and Automation.
* Plan to test and quantify application security behavior during the QA process, just like any other system functionality.
* Include the following considerations in your test plans:
* The policy compliance framework requirements
* Overviews of security testing methods, tools, training, and resource allocations
* The operating budget and schedule considerations
* Select a preferred vulnerability scoring system (CVSS, OVAL, etc.) and a management/tracking system (Bugzilla, a third-party vulnerability management package or service, etc.)
* Establish and collect useful metrics that will facilitate decision making (for example, the count of open defects by severity and category, the arrival count over time, the close rate, total testing coverage, etc.)
* Identify the testing activities which will be automation candidates and discuss how it will be done.
* Have a set of QA entry criteria, which identifies the items necessary to begin testing:
* Policy compliance validation requirements
* The applicable threat modeling scenarios
* The testing schedule, resource list, and budget
* The metric and vulnerability scoring system selections
* An organizationally meaningful certification, which shows the QA team participated in design reviews and was satisfied with the security parameters of the system.
* The completed test plans
* The QA exit criteria should include proof of application security integrity and readiness including:
* A summary report with charts, which summarize the collected metrics.
* A security testing report, which describes how well the application performed, compared to the policy compliance requirements and threat modeling scenarios, and its readiness compared to the established security baselines.
* No outstanding high-severity security defects (for example, a simple list showing that all severity 1 security bugs have been resolved and verified).
* An assessment which uses metrics to show that application security meets or exceeds established baselines, and that all security-related design goals have been met (that is, proof that the job is well done).

## IMPLEMENTATION OF QUALITY ASSURANCE

Utilize the test planning, test results, and metrics data to quantify the application security meets or exceeds the policy compliance and risk assessment goals.

### Identifying if software is vulnerable

* The presence of a working process results in an operating culture having certain distinguishing characteristics. Make sure you see some or all of these operating in yours.
* For example,
* The development team members are getting routinely updated on secure coding practices.
* Design reviews incorporate and encourage security considerations.
* The QA process includes planning and testing time for security assessments, instead of covering them as an afterthought or in an ad-hoc fashion.
* Security-related bugs are specifically tracked and have an established escalation policy.

### How to protect

* Make “Security” an operating word in the engineering team’s vocabulary. Encourage training opportunities, discussions, coding examples, and on-going interest.
* Select and employ a vulnerability scoring system, such as CVSS, OVAL, or the like. Or at least make sure that security related defects have some special tracking method or tag.
* Make sure the question of “Got security?” comes up during design reviews.
* Establish a working escalation procedure for security-related defects.

## METRICS

The QA group will identify, select, and employ the meaningful metrics to provide the baseline measurement of application security. This baseline will serve as a comparison point for future assessments, too.

### How to identify if software is vulnerable

A good system of metrics provides a basis for the following:

* Summary charts, showing the security-related bug counts over time, their open and closure rates, and the progress towards policy compliance and risk assessment goals.
* The numbers necessary to answer management’s questions about “How secure is the application?” or “Is risk increasing or decreasing over time?”
* A known security defect density (that is, the average number of security bugs per unit of code is being monitored and the rate is going in the right direction: Down!)

### How to protect yourself

* Establish a working set of metrics. For example, count the number of high, medium, and low severity security bugs as a start. Follow with rate assessments, which will answer questions like, “How fast are security-related bugs being discovered in QA testing?”, “How severe are the bugs that are being detected?”, and “How complete is the testing coverage for the areas prioritized by our policy compliance or risk assessment goals?”
* Track that all security related tests have been checked (a simple spreadsheet will do).
* Automate the calculation and charting of the metrics as possible, so accurate information is available on-demand, even in a dashboard summary fashion.
* Make sure all high-priority security bugs are fixed and regression-checked, prior to software release.

## TESTING ACTIVITIES

### How to identify if software is vulnerable

Not every QA team will employ all of the following testing activities, but the more you employ strategically, the better your security assurance will be:

* Cross-site scripting and SQL injection tests have been run.
* An assessment of how well the application handles user input, including special or multisystem characters, excessively long strings, null inputs, or invalid values has been done.
* Cookie or credentials manipulation testing has been performed.
* Denials of Service scenarios have been checked. It is understood how the application will perform in the presence of connection, login, or transaction flooding.

### How to protect yourself

* Run user agent injection tests (cross-site scripting, SQL query injections, data manipulation checks).
* Check how the application handles user input that is ill-formed, too short or too long, or that contains special or multibyte characters.
* Check how sensitive the application is to cookie manipulation or session tampering.
* Verify the application’s behavior under load. For example, what happens if 1,000 users login simultaneously? Or if a flood of TCP/IP connections are established, but no SYNs are received?

# VERIFICATION AND VALIDATION OF SOFTWARES

A perfect software product is built when every step is taken with full consideration that 'a right product is developed in a right manner'. 'Software verification & validation' is one such model, which helps the system designers and test engineers to confirm that a right product is built in the right way throughout the development process and improve the quality of the software product.

Verification & Validation Model' makes it sure that, certain rules are followed at the time of development of a software product and also makes it sure that the product that is developed fulfills the required specifications. This reduces the risk associated with any software project up to a certain level by helping in detection and correction of errors and mistakes, which are unknowingly done during the development process.

## VERIFICATION OF SOFTWARE

The standard definition of verification goes like this: "Are we building the product RIGHT?" i.e., Verification is a process that makes it sure that the software product is developed in the right way. The software should confirm to its predefined specifications. As the product development goes through different stages, an analysis is done to ensure that all required specifications are met.

Methods and techniques used in the verification and validation shall be designed carefully, the planning of which starts right from the beginning of the development process. The verification part of 'Verification and Validation Model' comes before validation, which incorporates software inspections, reviews, audits, walkthroughs, buddy checks, etc. in each phase of verification (every phase of verification is a phase of the testing life cycle).During the verification, the work product (the ready part of the software being developed and various documentations) is reviewed/examined personally by one or more persons in order to find and point out the defects in it. This process helps in prevention of potential bugs, which may cause failure of the project.

### Terms Involved in Verification

Inspection  
Inspection involves a team of about 3-6 people, led by a leader, which formally reviews the documents and work product during various phases of the product development life cycle. The work product and related documents are presented in front of the inspection team, the member of which carries different interpretations of the presentation. The bugs that are detected during the inspection are communicated to the next level in order to take care of them.

Walkthroughs  
Walkthrough can be considered same as inspection without formal preparation (of any presentation or documentations). During the walkthrough meeting, the presenter/author introduces the material to all the participants in order to make them familiar with it. Even when the walkthroughs can help in finding potential bugs, they are used for knowledge sharing or communication purpose.

#### Buddy Checks

This is the simplest type of review activity used to find out bugs in a work product during the verification. In buddy check, one person goes through the documents prepared by another person in order to find out if that person has made mistake(s) i.e., to find out bugs which the author couldn't find previously. The activities involved in verification process are: requirement specification verification, functional design verification, internal/system design verification and code verification (these phases can also be subdivided further). Each activity makes it sure that the product is developed in the right way and every requirement, every specification, design code etc., is verified!

## VALIDATION OF SOFTWARE

Validation is a process of finding out if the product being built is right? That is, whatever software product is being developed, it should do what the user expects it to do. The software product should functionally do what it is supposed to, it should satisfy all the functional requirements set by the user. Validation is done during or at the end of the development process in order to determine whether the product satisfies specified requirements. Validation and verification processes go hand in hand, but visibly validation process starts after verification process ends (after coding of the product ends). Each verification activity (such as requirement specification verification, functional design verification, etc.) has its corresponding validation activity (such as functional validation/testing, code validation/testing, system/integration validation, etc.).All types of testing methods are basically carried out during the validation process. Test plan, test suits and test cases are developed, which are used during the various phases of validation process. The phases involved in validation process are: Code Validation/Testing, Integration Validation/Integration Testing, Functional Validation/Functional Testing, and System/User Acceptance Testing/Validation.

### Terms Used in Validation Process

#### Code Validation/Testing

Developers as well as testers do the code validation. Unit Code Validation or Unit Testing is a type of testing, which the developers conduct in order to find out any bug in the code unit/module developed by them. Code testing, other than Unit Testing, can be done by testers or developers.

#### Integration Validation/Testing

Integration testing is carried out in order to find out if different (two or more) units/modules co-ordinate properly. This test helps in finding out if there is any defect in the interface between different modules.

#### Functional Validation/Testing

This type of testing is carried out in order to find if the system meets the functional requirements. In this type of testing, the system is validated for its functional behavior. Functional testing does not deal with internal coding of the project, instead, it checks if the system behaves as per the expectations.

#### User Acceptance Testing or System Validation

In this type of testing, the developed product is handed over to the user/paid testers in order to test it in a real-time scenario. The product is validated to find out if it works according to the system specifications and satisfies all the user requirements. As the user/paid testers use the software, it may happen that bugs that are yet undiscovered, come up, which are communicated to the developers to be fixed. This helps in improvement of the final product.

# REVIEWS AND AUDITS OF SOFTWARE

## STANDARD / PROCEDURE DESCRIPTION

This section will give a detailed outline of the procedures for review/audit type i.e. management and technical reviews, inspections, walkthroughs and audits.

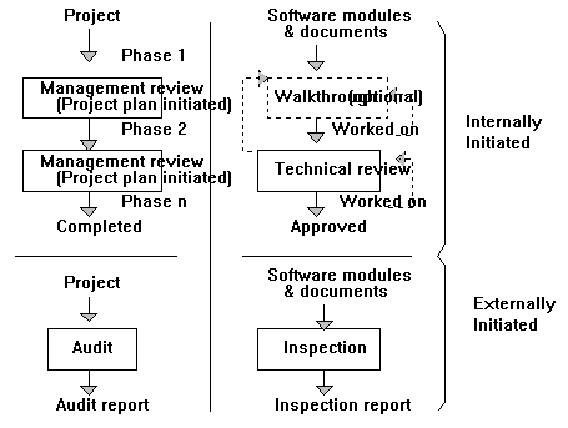
### When to do what

Figure 1 below illustrates the way in which the various reviews and audits are used and how they differ from each other. The processes have two main distinctions, as follows.

### Subject matter

Management reviews and audits evaluate the project as a whole, whereas walkthroughs, technical reviews and inspections are applied to individual documents or software modules.

### Means of initiation

Audits and inspections are normally initiated by parties external to the project team, such as the quality manager or the steering committee. They are done by people selected by the initiating party who are independent of the project team, though the project team would still assist with information. Walkthroughs and technical reviews, and to a lesser extent management reviews are common project activities initiated by the project team during the normal course of a project. Walkthroughs are optional reviews which are performed prior to completion of the document or software module.

## REVIEW STATUS REGISTER

A review status register shall be created to record the progress of reviews. The register shall record at least the following details for each review performed.

* Review report document number (in accordance with Std-03).
* Main reviewed item document number (in accordance ISSD Std-03).
* Review start dates.
* Review end date.
* Number of decisions.
* Number of action points.
* Number of major issues/defects.
* Review effort.
* Size of reviewed material.
* Exit decision (pass, provisional pass or fail).
* Date action points were completed.
* Date all issues/defects were corrected.
* Review outcomes finalized date.
* Status (planned, active or complete).

## MANDATORY AND OPTIONAL PRACTICE

### Mandatory

* In the conduct of QMS projects the following activities are mandatory.
* Technical review of all documents prior to approval
* Management reviews of projects at the completion of each project phase.

### Optional (recommended)

* In the conduct of QMS projects the following activities are optional but recommended.
* Walkthroughs of documents during their development as a means of improving document quality.
* Technical reviews of selected software modules to ensure standard and specification compliance.

### Optional

* In the conduct of QMS projects the following activities are optional only.
* Walkthroughs of software modules to identify areas for improvement.
* Audit(s) of the project to establish compliance with QMS requirements.
* Inspections of documents and software modules to verify compliance with QMS standards.
* Technical reviews of all software modules.

## PARTIAL OR COMPLETE REVIEWS

Reviews are designated in the objectives as being either partial or complete reviews. Partial reviews examine only those elements which have changed. Partial reviews can only be done after a complete review.

When a document, software item or other project item is developed, it undergoes a review (technical review) to validate its content, and compliance with the applicable standards. (For documents, approval can only occur after the review and any follow-up actions have been resolved.)

This review must be a full review of the item. That is it must review the entire content of the item. If the item is later modified in some way another review must be performed. This review may elect to cover only the changed portion of the material. Such reviews are referred to as partial reviews.

The following applies to partial reviews.

* They may be used if the material to be reviewed has previously been subject to a complete review
* They shall not be used if the material has undergone major changes
* The sections to be reviewed must be stated in the review objectives
* The review meeting must agree that a partial review is appropriate and that the changes do not affect the remainder of the document.

## MANAGEMENT REVIEW PROCESS

A management review is the formal evaluation of a project by a designated review team to determine whether the project is proceeding according to the project and quality plans.

During the review meeting the whole review team examines plans, or the progress against applicable plans, standards and/or guidelines. Each problem area is recorded. When critical data and information cannot be provided, an additional meeting shall be scheduled to complete the management review process.

The results of the review are published in a report which is filed in accordance with Std-03.

While management reviews can be done at any time, they are normally done at the end of each phase of the project.

The objectives of the management review process are as follows.

* Ensure activities move forward according to plan, based on an evaluation of the product development status.
* Change project direction or to identify the need for alternative planning.
* Maintain overall control of the project through the adequate allocation of resources.

## INITIATION

### Initiating event

Management reviews are scheduled activities; their timing is determined by the project and quality plans. Unscheduled management reviews may be called by the project manager, project steering committee or quality manager.

### Staffing

The review leader for scheduled management reviews is nominated by the project manager, unless specified otherwise in the project or quality plan. The review leader for unscheduled management reviews is determined by the party initiating the review in consultation with the project manager.

A management review shall consist of at least three people. The roles and responsibilities of these people are given below.

|  |  |
| --- | --- |
| **Role** | **Responsibility** |
| Review leader | * Issue the review notification. * Produce and distribute the review material in advance. * Chair the review meeting to ensure its objectives are met. * Issue the review report. * Prepare for the review by studying the distributed material. * Participate in the review to ensure its objectives are met. |
| Recorder | * Record the action points and decisions during the review. * Reflect any changes to the review objectives and/or agenda in the review report. * Prepare for the review by studying the distributed material. * Participate in the review to ensure its objectives are met. |
| Reviewer | * Prepare for the review by studying the distributed material. * Participate in the review to ensure its objectives are met |

### Notification

Participants in a management review shall be given at least five working days notice of the review. The review notification shall include the following information.

* Project name and number.
* Date, time and venue of the review meeting.
* Type of review - (i.e. management, technical, inspection or walkthrough).
* List if input material.
* Statement of review objectives.
* List of issues ensuring objectives cover all issues to be discussed.
* Review team - who will participate and in what role (i.e. moderator/leader, author, recorder etc.).
* Agenda.
* Date, time, venue and expected duration of the preliminary briefing (if required).

Where a management review determines that subsequent meetings are required to complete the review date & time decided by the meeting (notification is not required).

## INPUT MATERIAL

The input material shall be made available to the participants by the review leader at least two working days prior to the review.

The minimum input to the management review is as follows.

* Statement of objectives (in review notification).
* Current project plan, schedule and cost data.
* Reports (i.e. managerial review reports, technical review reports, audit reports) from other reviews/audits which have already been done (project documents are filed - Project Document Filing.)
* Reports of resources assigned to the project.
* Data on the documents and software modules completed, or in progress.
* Additional information as required by the review leader.

# CONCLUSION

The quality of software is assessed by a number of variables. These variables can be divided into external and internal quality criteria. External quality is what a user experiences when running the software in its operational mode. Internal quality refers to aspects that are code-dependent, and that are not visible to the end-user. External quality is critical to the user, while internal quality is meaningful to the developer only.

Some quality criteria are objective, and can be measured accordingly. Some quality criteria are subjective, and are therefore captured with more arbitrary measurements.

Internal quality (code characteristics) is a concern to the developer only, while all the external quality aspects (coming from using the software) are critical to the end user. However the developer has also interests in performances (speed, space, network usage) and determinism, because they make testing the software easier. Developers treat ease-of-use, back-compatibility, security, and power consumption as requirements.

It is important to consider how difficult it is to measure each of these criteria. It can be difficult because there is no simple variable to look at, or because the measurement process is costly, or because it requires a complex infrastructure. For instance, speed has an objective measurement that is easy to measure. Power consumption has a simple measurement (how many µW the application consumes), but it is complex to measure while security of software is difficult and costly to estimate.

References

1. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-0) Pressman, Scott (2005), Software Engineering: A Practitioner's Approach (Sixth, International ed.), McGraw-Hill Education [Pressman](http://en.wikipedia.org/wiki/Software_quality#CITEREFPressman), p. 388
2. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-1) [Medical Devices: The Therac-25\*](http://sunnyday.mit.edu/papers/therac.pdf), Nancy Leveson, Uninversity of Washington
3. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-2) [Embedded Software](http://ptolemy.eecs.berkeley.edu/publications/papers/02/embsoft/embsoftwre.pdf), Edward A. Lee, To appear in Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002, Revised from UCB ERL Memorandum M01/26 University of California, Berkeley, CA 94720, USA, November 1, 2001
4. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-3) [Improving Quality Through Better Requirements (Slideshow)](http://www.ralphyoung.net/conferences/ImprovingQuality2.ppt), Dr. Ralph R. Young, 24/01/2004, Northrop Grumman Information Technology
5. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-4) DeMarco, T., *Management Can Make Quality (Im)possible*, Cutter IT Summit, Boston, April 1999
6. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-5) Crosby, P., *Quality is Free*, McGraw-Hill, 1979
7. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-6) [McConnell, Steve (1993), Code Complete (First ed.), Microsoft Press](http://en.wikipedia.org/w/index.php?title=Software_quality&diff=436729813&oldid=434635568#CITEREFMcConnell1993)
8. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-7) Park, R.E. (1992). Software Size Measurement: A Framework for Counting Source Statements. (CMU/SEI-92-TR-020). Software Engineering Institute, Carnegie Mellon University
9. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-8) Halstead, M.E. (1977). Elements of Software Science. Elsevier North-Holland.
10. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-9) Chidamber, S. & C. Kemerer. C. (1994). A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20 (6), 476-493
11. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-10) Nygard, M.T. (2007). Release It! Design and Deploy Production Ready Software. The Pragmatic Programmers.
12. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-11) Martin, R. (2001). Managing vulnerabilities in networked systems. IEEE Computer.
13. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-12) Boehm, B., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., & Merritt, M.J. (1978). Characteristics of Software Quality. North-Holland.
14. [**^**](http://en.wikipedia.org/wiki/Software_quality#cite_ref-13) [IfSQ Level-2 A Foundation-Level Standard for Computer Program Source Code](http://www.ifsq.org/resources/level-2/booklet.pdf), Second Edition August 2008, Graham Bolton, Stuart Johnston, IfSQ, Institute for Software Quality.
15. Dhillon B. S., “*Reliability in Computer System Design*”, Intellect Books, 1987
16. Schick, G.J. and Wolverton, R.W., “*An analysis of Competing Software Reliability Models*”, IEEE Transactions on Software Engineering, Vol. 4, 1978 pp 104-120
17. Haugk, G., Tsiang, S.H. and Zimmerman, L., “*System testing of the No. 1 Electronic Switching System*”, Bell System Tech. Journal, Vol. 43, 1964, pp 2575-2592
18. Floyd, R.W., “*Assigning Meanings to Programs*”, Mathematical Aspects of Computer Science, Vol. XIX, 1967 pp 19-32
19. Hudson, G.R., “*Programming Errors as a Birth-and-Death Process*”, Report No. SP-3011, System Development Corporation, December 1967
20. Naur, P., “*Software Engineering Concepts and Techniques*”, Petrocelli/Charter Books, New York, 1976
21. Barlow, R. and Scheuer, E.M., “*Reliability Growth During a Development Testing Program*”, Technometrics, Vol. 8, 1966 pp 53-60
22. London, R.L., “*Providing Programs Correct: Some Techniques and Examples*”, BIT Vol. 10 1969, pp 168-182
23. Suuter, J.L., “*Reliability in Computer Programs*”, Mechanical Engineering, Vol. 91, 1969, pp 24-27
24. Xie, M., “*Software Reliability Modelling*”, World Scientific, 1991

Pham, H., “*Software reliability*”, Springer, 2000

1. Musa, J.D., Iannino, A., and Okumoto, K., “*Software Reliability*”, McGraw-Hill, New

York, 1987

1. Sukert, A.N., “*An investigation of software reliability models*”, Proceedings Annual Reliability Maintainability Symposium., 478-484, 1977.
2. Dhillon B. S., “*Design reliability: fundamentals and applications*”, CRC Press, 1999
3. Mills, H.D., “*On the Statistical Validation of Computer Programs*”, Report No. 72-6015, 1972 IBM Federal Systems Division, Gaithersburg, MD.
4. Musa, J.D., “*A theory of software reliability and its applications*”, IEEE Trans. Software Eng., Vol. 1, 312-327, 1975.
5. Shooman, M.L., “*Software reliability measurement and models*”, Proceedings Annual Reliability Maintainability Symposium, 485-491, 1975.
6. Jelinski, Z. and Moranda, P.B., “*Software reliability research*”, Proceedings of the

Statistical Methods for the Evaluation of Computer System Performance, Academic

Press, 1972, pp. 465-484.

1. Nelson, E., “*Estimating software reliability from test data*”, Microelectronics and Reliability, 17, 67-75, 1978.
2. Ramamoorthy, C.V. and Bastani, F.B., “*Software reliability: status and perspectives*”,

IEEE Transactions Software Engineering, Vol. 8, 354-371, 1982.

1. Dunn, R. and Ullman, R., “*Quality Assurance for Computer Software*”, McGraw-Hill, New York, 1982.
2. Duane, J.T., “*Learning curve approach to reliability monitoring*”, IEEE Transactions Aerospace, Vol. 2, 563-566, 1964.
3. Lyu, M.R., “*Handbook of Software Reliability Engineering*”, McGraw-Hill, New York, 1996.
4. Rome Air Development Center, “*Methodology for Software Reliability Prediction and Assessment*”, Report No. RLTR-92-52, Volumes I and II, Rome Air Development Center, Griffiss Air Force Base, Rome, NY, 1992.
5. Lakey, P.B. and Neufelder, A.M., “*System and Software Reliability Assurance Notebook*”, Rome Laboratory
6. Lyu, M.R., “*Software Reliability Engineering: A Roadmap*”, Future of Software Engineering, 2007