# ICT-5405

## PROJECT MANAGEMENT AND QUALITY ASSURANCE

## 07 Software Quality Requirements

## Introduction

- All software is an element of a system, whether it be a computer system in which the software may be used on a personal computer, or in an electronic consumer product like a digital camera.
- The needs or requirements of these systems are typically documented either in a request for quote (RFQ) or request for proposal (RFP) document, a statement of work (SOW), a software requirements specification (SRS) document or in a system requirements document (SRD).
- Using these documents, the software developer must extract the information needed to define specifications for both the functional requirements and performance or non-functional requirements required by the client

# Definition

**Functional Requirement**

A requirement that specifies a function that a system or system component must be able to perform.
ISO 24765 [ISO 17a]

**Non-Functional Requirement**

A software requirement that describes not what the software will do but how the software will do it. Synonym: design constraint. ISO 24765 [ISO 17a]

**Performance Requirement**

The measurable criterion that identifies a quality attribute of a function or how well a functional requirement must be accomplished (IEEE Std 1220TM-2005). A performance requirement is always an attribute of a functional requirement.

## Quality Model

- A defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality. ISO 25000 [ISO 14a]

The above definition of a quality model implies that the quality of software can be measured.

## Quality Model

- To prove that the quality requirements were met, a software quality model is used so that the client can:
  – define software quality characteristics that can be evaluated;
  – contrast the different perspectives of the quality model that come into play (i.e., internal and external perspectives);
  – carefully choose a limited number of quality characteristics that will serve as the non-functional requirements for the software (i.e., quality requirements);
  – set a measure and its objectives for each of the quality requirements.

## SOFTWARE QUALITY MODELS

The five quality perspectives described by Garvin (1984) [GAR 84].

1. **Transcendental approach to quality:**

The transcendental view of quality can be explained as follows: “Although I can’t define quality, I know it when I see it.” The main problem with this view is that quality is a personal and individual experience. You would have to use the software to get a general idea of its quality.
Garvin explains that software quality models all offer a sufficient number of quality characteristics for an individual or an organization to identify and evaluate within their context. In other words, the typical model sets out the quality characteristics for this approach, and it only takes time for all users to see it.

# SOFTWARE QUALITY MODELS

## 02. User-based approach:

A second approach studied by Garvin is that quality software performs as expected from the user’s perspective (i.e., fitness for purpose). This perspective implies that software quality is not absolute, but can change with the expectations of each user.

## 03. Manufacturing-based approach:

This view of software quality, in which quality is defined as complying with specifications, is illustrated by many documents on the quality of the development process. Garvin specifies that models allow for defining quality requirements at an appropriate level of specificity when defining the requirements and throughout the life cycle. Therefore, this is a “process-based” view, which assumes that compliance with the process leads to quality software.

## SOFTWARE QUALITY MODELS

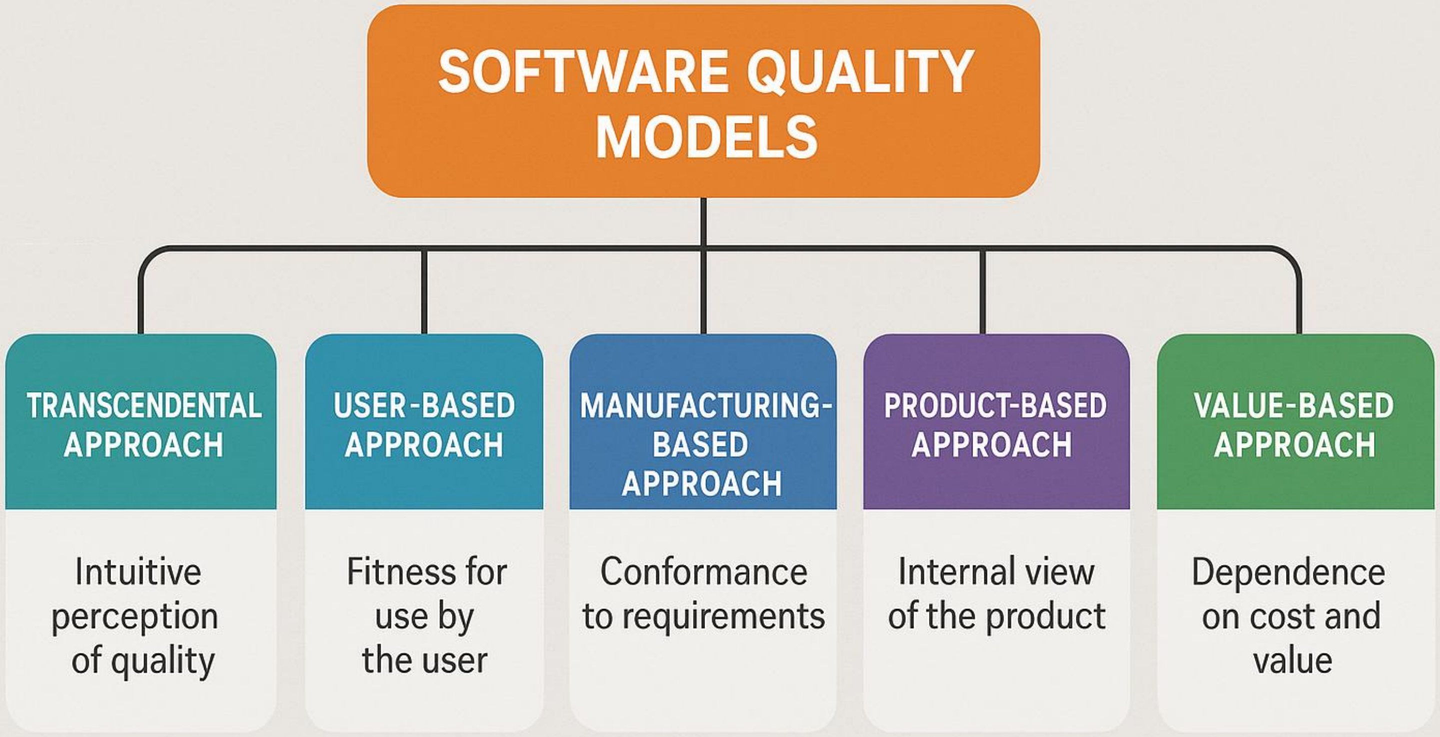
### 04. Product-based approach:

The product-based quality perspective involves an internal view of the product. The software engineer focuses on the internal properties of the software components, for example, the quality of its architecture. These internal properties correspond to source code characteristics and require advanced testing techniques. **Garvin explains** that if the client is willing to pay, then this perspective is possible with the current models. He describes the case of NASA, who was willing to pay an extra thousand dollars per line of code to ensure that the software aboard the space shuttle met high quality standards

## SOFTWARE QUALITY MODELS

**05. Value-based approach:**

this perspective focuses on the elimination of all activities that do not add value, for example the drafting of certain documents as described by Crosby (1979) [CRO 79]. *In the software domain, the concept of "value" is synonymous with productivity, increased profitability, and competitiveness.* It results in the need to model the development process and to measure all kinds of quality factors. These quality models can be used to measure these concepts, but really only for insiders and mature organizations.

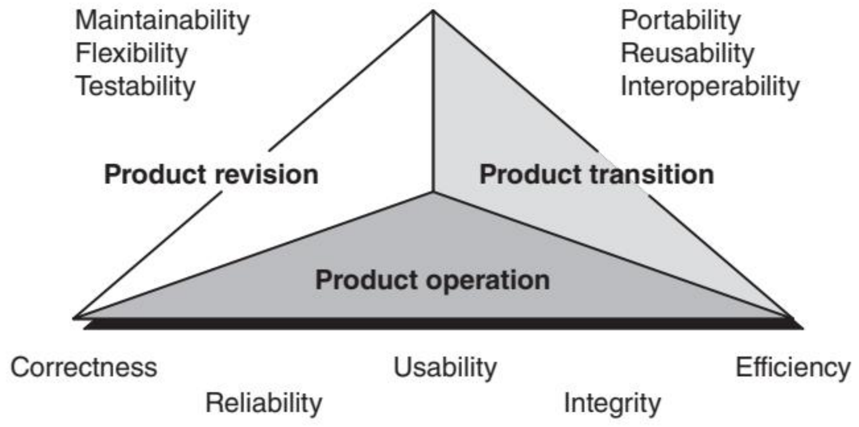

SOFTWARE QUALITY MODELS
TRANSCENDENTAL APPROACH
Intuitive perception of quality
USER-BASED APPROACH
Fitness for use by the user
MANUFACTURING-BASED APPROACH
Conformance to requirements
PRODUCT-BASED APPROACH
Internal view of the product
VALUE-BASED APPROACH
Dependence on cost and value

GARVIN'S FIVE QUALITY PERSPECTIVES

# Initial Model Proposed by McCall

McCall and his colleagues have been attributed with the original idea for a software quality model [MCC 77]. This model was developed in the 1970s for the United States Air Force and was designed for use by software designers and engineers. **It proposes three perspectives for the user and primarily promotes a product-based view of the software product:**
– **Operation:** during its use;
– **Revision:** during changes made to it over the years;
– **Transition:** for its conversion to other environments when the time comes to migrate to a new technology.

Figure: The three perspectives and 11 quality factors of McCall et al. (1977)



# McCall's Quality Factors

| Quality Factor | Definition |
|---|---|
| **Correctness** | Extent to which a program satisfies its specifications and fulfills the user's mission objectives. |
| **Reliability** | Extent to which a program can be expected to perform its intended function with required precision. |
| **Efficiency** | Amount of computing resources and code required by a program to perform a function. |
| **Integrity** | Extent to which access to software or data by unauthorized persons can be controlled. |
| **Usability** | Effort required to learn, operate, prepare input, and interpret output of a program. |
| **Maintainability** | Effort required to locate and fix a defect in an operational program. |
| **Testability** | Effort required to test a program to ensure that it performs its intended functions. |
| **Flexibility** | Effort required to modify an operational program. |
| **Portability** | Effort required to transfer a program from one hardware and/or software environment to another. |
| **Reusability** | Extent to which parts of a software system can be reused in other applications. |
| **Interoperability** | Effort required to couple one system with another. |

# External Quality and Internal Quality

## What Constitutes the External Quality and Internal Quality of Software?

From the **external point of view**, the focus is on presenting characteristics that are important to people who do not know the technical details. For example, a user is interested in the speed in which the maintainer can make a requested change to the software because the effort dedicated to this task has an impact on cost and waiting time. The user neither knows nor cares about technical details of the software, so his perspective is external.

From an **internal point of view**, the focus is on measuring the attributes of maintainability, which will influence: (1) the effort to identify where to make the change, (2) changes to the current structures (attempting to reduce the impact of a change), (3) testing the change, and (4) its release into production. If software is not well documented and is poorly structured, it will have low maintainability (internal quality). This low internal quality will affect the time required for making the change, which is the external characteristic that interests the user. Therefore, we can see here that *external quality is not a directly observable measure*, but it is derived from internal quality. Internal quality however, can be directly measured in the software.

# Requirements

- Requirements are generally grouped into three categories:
- 1) **Functional Requirements**: These describe the characteristics of a system or processes that the system must execute. This category includes business requirements and functional requirements for the user.
- 2) **Non-Functional (Quality) Requirements:** These describe the properties that the system must have, for example, requirements translated into quality characteristics and sub-characteristics such as security, confidentiality, integrity, availability, performance, and accessibility.
- 3) **Constraints:** Limitations in development such as infrastructure on which the system must run or the programming language that must be used to implement the system.

# The First Standardized Model: IEEE 1061

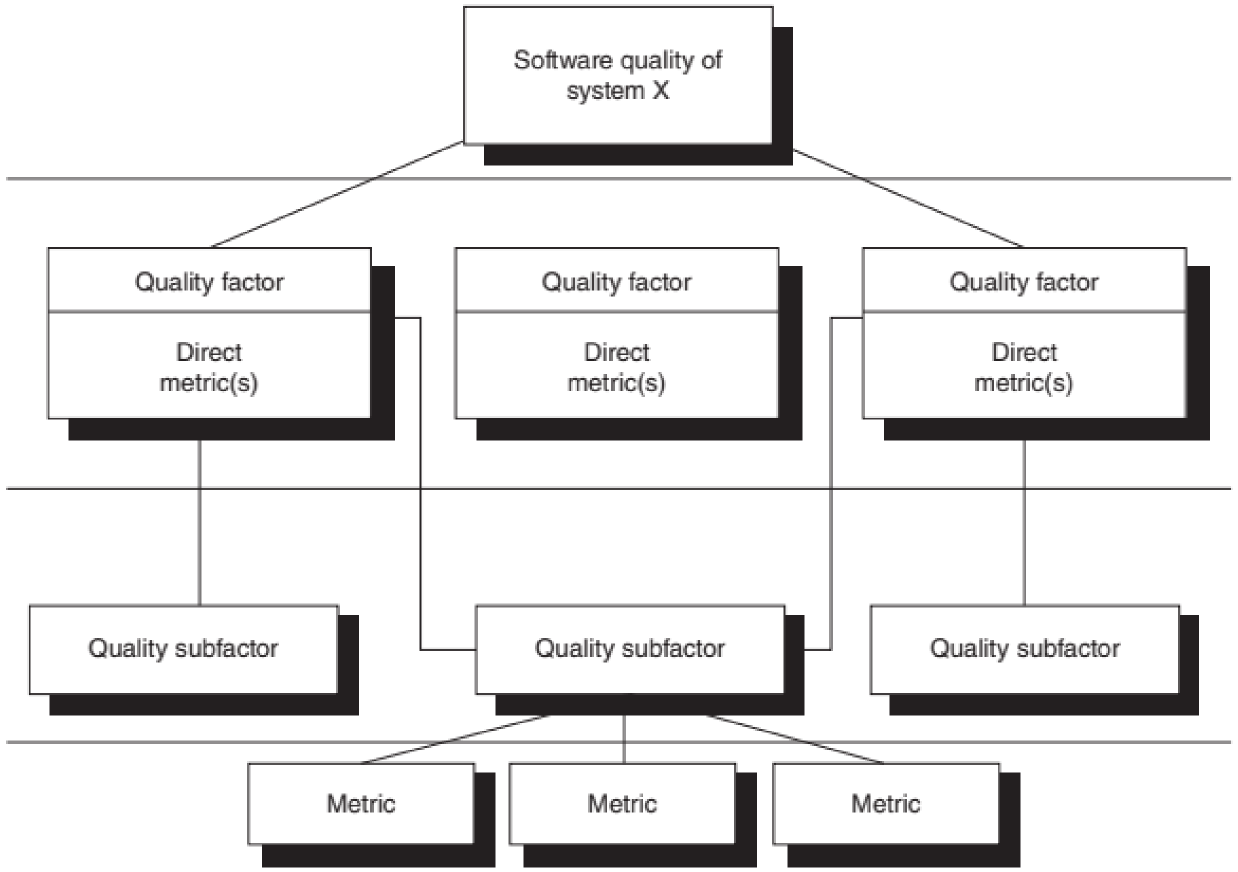


**Figure 3.3** Framework for measuring software quality as per the IEEE 1061 [IEE 98b].

## IEEE 1061

- The **IEEE 1061** standard, that is, the Standard for a **Software Quality Metrics Methodology** provides a framework for *measuring software quality* that allows for the establishment and identification of software quality measures based on quality requirements in order to ***implement, analyze, and validate*** software processes and products.
- This standard claims to adapt to all business models, types of software, and all of the stages of the software life cycle.

# Software Quality Metrics

- **Product metrics:** Describes the characteristics of the product such as size, complexity, design features, performance, and quality level.
- **Process metrics:** These characteristics can be used to improve the development and maintenance activities of the software.
- **Project metrics:** This metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

- **Software quality metrics** are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.
  1. Product quality metrics
  2. In-process quality metrics
  3. Maintenance quality metrics

# Product quality metrics

## 1.Mean Time to Failure

It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

## 2.Defect Density

It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

## 3.Customer Problems

Measures the problems that customers encounter when using product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented with the defect problems.
The problems metric is usually expressed in terms of **Problems per User-Month (PUM)**.

**PUM** = Total Problems customers reported (true defect) for a time period / Total number of license months of the software during the period

Where, Number of license-month of the software = Number of install license of the software × Number of months in the calculation period

# Product quality metrics

## 4.Customer Satisfaction

Customer satisfaction is often measured by customer survey data through the five-point scale –

1. Very satisfied
2. Satisfied
3. Neutral
4. Dissatisfied
5. Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example –

- Percent of completely satisfied customers
- Percent of satisfied customers
- Percent of dis-satisfied customers
- Percent of non-satisfied customers
- Usually, this percent satisfaction is used.

## Mean time between failures ( MTBF)

- It is defined in the requirements specifications as being the ability of a software product to maintain a specified level of service when it is used under specific conditions. Such as mean time between failures (or MTBF).
- "**Mean Time Between Failures**" is literally the average time elapsed from one failure to the next. Usually people think of it as the average time that something works until it fails and needs to be repaired (again).

$$MTBF = \frac{Total\ Working\ Time\ -\ Total\ Breakdown\ Time}{Number\ of\ Breakdowns}$$

# MTBF Calculation Example

Let's imagine that an asset which is expected to work for 24 hours a day has three outages. One lasts for an hour, another for 2 hours and the final breakdown lasts 30 minutes.

total working time = 24 hours
total breakdown time = 3.5 hours (1 + 2 + 0.5)
number of breakdowns = 3

📌 Then, as per the MTBF formula, the Mean Time Between Failures calculation will be:

$$MTBF = \frac{24 - 3.5}{3} = \frac{24 - 3.5}{3} = {\sim}6.83$$

## Exercise:

- **Calculate the MTBF for software that must be available during work hours, that is, 37.5 hours a week, and cannot fail more than 15 minutes per week.**

the software must be available for 37.5 hours per week, and it can fail for 15 minutes per week.
Therefore, the total operational time is 37.5 - 0.25 = 37.25 hours per week, and the number of failures is 1 per week.

Using the MTBF formula, we get:
MTBF=Total operational time/Number of failures
=37.25/1=37.25 hours per week

# Avalaibility

- $A = \dfrac{\textit{hours available}}{(\textit{hours available + hours unavailable})}$

- For example, the work team, when preparing the system specifications, indicates that the MTBF should be 95% to be acceptable (during service hours). If the software must be made available during work hours, that is, 37.5 hours per week, it should therefore not be down for more than 2 hours a week: $\dfrac{37.5}{(37.5 + 2)} = 0.949\%$.

## Suppose for any system MTTF (mean-time-to-failure) = 200 and MTTR (mean-time-to-repair) = 100 time units. Find the mean-lime-between-failure (MTBF), and availability of the product.

The Mean Time Between Failures (MTBF) can be calculated using the formula:
***MTBF=MTTF+MTTR***

*Given that MTTF (Mean Time To Failure) is 200 time units and MTTR (Mean Time To Repair) is 100 time units:*

***MTBF=200+100=300***
*So, the MTBF for the system is 300 time units.*

*The availability (A) of a system is calculated using the formula:*
*A=* **MTBF / (MTBF + MTTR)**
*Substitute the given values:*
*A= 300/300+100 =300/400=3/4*
*So, the availability of the product is 3/4 or approximately 75%.*

## Problems per User-Month (PUM)

- Let total problems that customers reported (true defect and non-defect oriented problems) for a time period = 873.
- Then the number of install license of the software = 120 and number of months in the calculation period = 1.5 months.
- Find the Problems per User-Month (PUM)

Here's the formula for PUM:

PUM = Total problems reported / Total number of user-months

First, we need to calculate the total number of user-months:

Number of user-months = Number of install licenses * Number of months

Number of user-months = 120 licenses * 1.5 months = 180 user-months

Now, we can plug in the values to find the PUM:

PUM = 873 problems / 180 user-months PUM

= 4.85 problems per user-month

Therefore, the Problems per User-Month (PUM) for the given data is 4.85.

# In-process Quality Metrics

## 1. Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. **Higher defect rates** found during testing is an indicator that the software has experienced higher error injection during its development process, unless the **higher testing defect rate** is due to an extraordinary testing effort.
This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

## In-process Quality Metrics

### 2. Defect arrival pattern during machine testing

- The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.

- The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.

- The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

# In-process Quality Metrics

## 3. Phase-based defect removal pattern

- This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

- Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

- With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.

# In-process Quality Metrics

## Defect removal effectiveness:

- This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

$$DRE = \frac{Defect\ removed\ during\ a\ development\ phase}{Defects\ latent\ in\ the\ product} \times 100\%$$

Suppose total defects latent in the system is 770 and among them 543 defects are removed during development phase. Find the Defect removal effectiveness (DRE).

# Maintenance Quality Metrics

- **Fix backlog and backlog management index**

  Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

  Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

- $$BMI = \frac{No.\ of\ problems\ closed\ during\ the\ month}{No.\ of\ problems\ arrived\ during\ the\ month} \times 100\%$$

- **If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.**

## Maintenance Quality Metrics

- **Fix response time and fix responsiveness**
  - The fix response time metric is usually calculated as the mean time of all problems from open to close.
  - Short fix response time leads to customer satisfaction.

**Percent delinquent fixes =**

$$\frac{\textit{No. fixes that exceeded the response time criteria of severity level}}{\textit{No. of fixes delivered in a specified time}} \times 100\%$$

- **Fix quality**

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

# Organizing for Quality Assurance

- **Managers**

  Top management executives, especially the executive directly in charge of software quality assurance

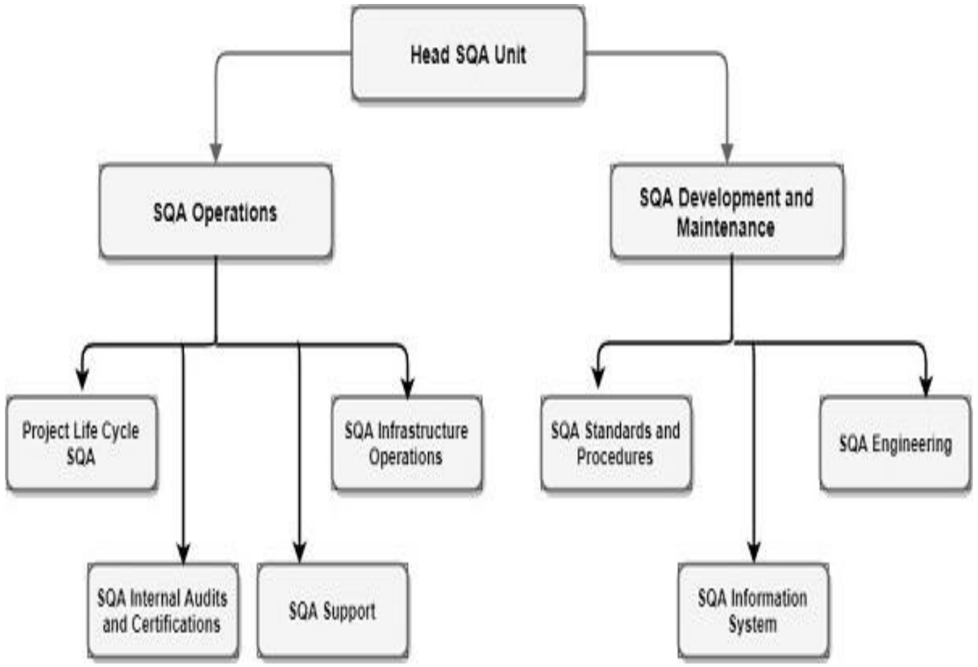
  Software development and maintenance department managers

  Software testing department managers

  Project managers and team leaders of development and maintenance projects

  Leaders of software testing teams
- **Testers**

  Members of software testing teams


Head SQA Unit
SQA Operations
SQA Development and Maintenance
Project Life Cycle SQA
SQA Infrastructure Operations
SQA Standards and Procedures
SQA Engineering
SQA Internal Audits and Certifications
SQA Support
SQA Information System

## Quality Factors of the ISO 25000 Standard

- Eight quality attributes for software,
  1. Functional Suitability
  2. Performance Efficiency
  3. Compatibility
  4. Usability
  5. Reliability
  6. Security
  7. Maintainability
  8. Portability

# Good software requirements

## Good software will have the following characteristics:

- **Necessary:** They must be based on necessary elements, that is, important elements in the system that other system components cannot provide.
- **Unambiguous:** They must be clear enough to be interpreted in only one way.
- **Concise:** They must be stated in a language that is precise, brief, and easy to read, which communicates the essence of what is required.
- **Coherent:** They must not contradict the requirements described upstream or downstream. Moreover, they must use consistent terminology throughout all requirements statements.
- **Complete:** They must all be stated fully in one location and in a manner that does not oblige the reader to refer to other texts to understand what the requirement means.
- **Accessible:** They must be realistic regarding their implementation in terms of available finances, available resources, and within the time available.
- **Verifiable:** They must allow for the determination of whether they are met or not based on four possible methods—inspection, analysis, demonstration, or tests.

# Factors

**Factors that Foster Software Quality**

1) Good understanding of non-functional quality.
2) Good process for defining, following up, and communicating quality requirements.
3) Evaluating quality throughout the life cycle of the software.
4) Establishing software criticality before starting a project.
5) Using the benefits of software traceability.

**Factors that may Adversely Affect Software Quality**

1) Not taking quality requirements into account.
2) Not taking software criticality into account.
3) Making excuses to not be concerned with quality

## Describe the steps proposed by the IEEE 1061 standard to determine a good definition of non-functional requirements

1. Start by identifying the list of non-functional (quality) requirements for the software as of the beginning of the specification's elicitation.
2. To define these requirements, contractual stipulations, standards, and company history must be taken into account.
3. Set a priority for these requirements and try not to resolve any conflicting quality requirements at this time.
4. Make sure that all participants can share their opinion when collecting information.