Chapter One
Introduction to Software Quality and Software Quality Assurance
1.1. Software Quality
The question "What is software quality?" evokes many different answers. Quality is a complex concept. It means different things to different people, and it is highly context dependent. The following five viewpoints of quality were analyzed by David Garvin (1980's) to show how software quality is perceived in different ways in different domains.

Viewpoints of Quality
1. Transcendental view
This view associates quality with the "innate excellence" of a product. It emphasizes on the idea that quality is universally recognized and measureable, which indicates high achievements and inflexible standards. This perspective designates the fact that product development always strives for producing the ideal "best" characteristics of a product the user wants and the attempt to achieve the ideal, although the ideal best product may not be produced at the end. This strived for ideal "best" characteristic of a product is the transcendental viewpoint of quality. It envisions quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas of everyday life.
2. User based/User's view
In this view, quality concerns the extent to which a product meets user needs and expectations. This view perceives quality as fitness for purpose. A product is said to have a good quality when users are satisfied in using it. This is to mean that if the product meets the purpose for which it was designed and developed in the first place and users are satisfied in using it, then it has a good quality. Users have different views on product usability depending on their needs. This view of quality indicates a more personalized view of users on product quality in a specific context of operation and functionality of the product. A product is of good quality if it satisfies a large number of users. It is useful to identify the product attributes, which the users consider to be important. It is useful to identify the product attributes, which the users consider to be important. This view may encompass many subject elements, such as *usability*, *reliability*, and *efficiency*. According to this view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"
3. Conformance to requirements (Manufacturing based)
This view has its origin in the manufacturing industry – auto and electronics. The key idea in this view of quality is ensuring that a product satisfies the requirements. Any deviation from the requirements is seen as reducing the quality of the product. The concept of process plays a key role. Products are manufactured "right the first time" so that the cost is reduced:
• Development cost
• Maintenance cost
The quality level of a product is determined by the extent to which the product meets its specifications.
Conformance to requirements leads to uniformity in products. Product quality can be incrementally improved by improving the process. Some argue that such uniformity does not guarantee quality. Product quality can be incrementally improved by improving the process. The CMM and ISO 9001 models are based on the manufacturing view.

4. Product based
In this approach, quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities. If a product is manufactured with good internal properties, then it will have good external properties. One can explore the causal relationship between *internal properties* and *external qualities*. Example: Modularity enables testability.
5. Value based
In this approach, quality is defined in terms of the relations between the value and cost of a product. A product with good quality provides performance at an acceptable price and conformance at an acceptable cost. The central idea of this view underlies in how much a customer is willing to pay for a product with certain level of quality. Quality is meaningless if a product does not make economic sense. The value-based view makes a trade-off between cost and quality. This view also represents the merger of two concepts: excellence and worth. Quality is a measure of excellence, and value is a measure of worth.
The definitions of different perspectives of quality indicate the different viewpoints of users and developers. Existing quality definitions fall into one of the 5 basic perspectives, even though the views of users and product manufacturers would be different. High quality for users is related with high performance and improved features of a product/service that satisfies their needs. Producers/developers on the other hand take a different line of thought; a product to be referred as having high quality, it has to conform to outlined requirement specifications.

Software Quality
There are several definitions given to software quality. Among which, the most comprehensive one is suggested by IEEE. Software quality, based on IEEE (IEEE, 1991) is defined as:
1. The degree to which a system, component, or process meets specified requirements.
2. The degree to which a system, component, or process meets customer or user needs or expectations.
Pressman gives another definition to software quality as follows:
• Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
This definition suggests three requirements for quality assurance that are to be met by the developer:

- Specific functional requirements, which refer mainly to the results of the software system
- The software quality standards mentioned in the contract
- Good software engineering practices, reflecting state-of-the-art professional practices, to be met by the developer even though not explicitly mentioned in the contract

1.2. Software Quality Assurance
Intuitively, when thinking about software, we imagine an accumulation of programming language instructions and statements or development tool instructions that together form a program or software package. The IEEE definition of Software states that software is:
• Computer programs,
• Procedures,
• Documentation
• Data pertaining to the operation of a computer system.

Software quality assurance always includes, in addition to code quality, the quality of the procedures, the documentation and the necessary software data. The combination of all four components is needed to assure the quality of the development process as well as the following long years of maintenance service for the following reasons:

• **Computer programs** (the "code") are needed because, obviously, they activate the computer to perform the required applications.

• **Procedures** are required, to define the order and schedule in which the programs are performed, the method employed, and the person responsible for performing the activities that are necessary for applying the software.

• Various types of **documentation** are needed for developers, users and maintenance personnel. The development documentation (the requirements report, design reports, program descriptions, etc.) allows efficient cooperation and coordination among development team members and efficient reviews and inspections of the design and programming products.

The user's documentation (the "user's manual", etc.) provides a description of the available applications and the appropriate method for their use.

• The maintenance documentation (the "programmer's software manual", etc.) provides the maintenance team with all the required information about the code, and the structure and tasks of each software module. This information is used when trying to locate causes of software failures ("bugs") or to change or add to existing software

• **Data** including parameters, codes and name lists that adapt the software to the needs of the specific user are necessary for operating the software. Another type of essential data is the standard test data, used to determine that no undesirable changes in the code or software data have occurred, and what kind of software malfunctioning can be expected.

1.2.1. Software Quality Assurance (SQA)

Software quality assurance is a **systematic, planned set of actions** necessary to provide adequate confidence that the software *development process* or the *maintenance process* of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary limits.

The purpose of SQA should not be limited to the development process. Instead, it should be extended to cover the long years of service subsequent to product delivery. Adding issues directly related to the software product introduces quality issues that integrate software maintenance functions into the overall conception of SQA. SQA actions are not limited to the technical aspects of the functional requirements, but should include also activities that deal with scheduling and the budget. The reasoning behind this expansion in scope is the close relationship between timetable or budget failure and the meeting of functional technical requirements.

The objectives of Software Quality Assurance

The main objective of quality assurance is to minimize the cost of guaranteeing quality by a variety of activities performed throughout the development and manufacturing processes/stages. These activities prevent the causes of errors, and detect and correct them early in the development process. As a result, quality assurance activities substantially reduce the rate of products that do not qualify for delivery and, at the same time, reduce the costs of guaranteeing quality in most cases.

1. Software development (process-oriented):

a. Assuring an acceptable level of confidence that the software will conform to functional technical requirements.

b. Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.

c. Initiating and managing of activities for the improvement and greater efficiency of software development and SQA activities. This means improving the prospects that the functional and managerial requirements will be achieved while reducing the costs of carrying out the software development and SQA activities.

2. Software maintenance (product-oriented):

a. Assuring with an acceptable level of confidence that the software maintenance activities will conform to the functional technical requirements.

b. Assuring with an acceptable level of confidence that the software maintenance activities will conform to managerial scheduling and budgetary requirements.

c. Initiating and managing activities to improve and increase the efficiency of software maintenance and SQA activities. This involves improving the prospects of achieving functional and managerial requirements while reducing costs.

1.2.2. Software Errors, Faults and Failures

Software errors are sections of the code that are partially or totally incorrect as a result of a grammatical, logical or other mistake made by a systems analyst, a programmer, or another member of the software development team.

Software faults are software errors that cause the incorrect functioning of the software during a specific application.

Software faults become **software failures** only when they are “activated”, that is, when a user tries to apply the specific software section that is faulty. Thus, the root of any software failure is a software error. A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.

The origin of software failures lies in a software error made by a programmer. An error can be a grammatical error in one or more of the code lines, or a logical error in carrying out one or more of the client’s requirements. However, not all software errors become software faults. In other words, in some cases, the software error can cause improper functioning of the soft- ware in general or in a specific application. In many other cases, erroneous code lines will not affect the functionality of the software as a whole; in a part of these cases, the fault will be corrected or “neutralized” by subsequent code lines.

We are interested mainly in the software failures that disrupt our use of the software. This requires us to examine the relationship between software faults and software failures. Do all software faults end with software failures? Not necessarily: a software fault becomes a software failure only when it is “activated”, that is when the software user tries to apply the specific, faulty application. In many situations, a software fault is never activated due to the user’s lack of interest in the specific application or to the fact that the combination of conditions necessary to activate the fault never occurs.

Examples of software failures

1. Ariane Rocket goes boom (1996)

Cost: $500 million

Disaster: Ariane 5, Europe's newest unmanned rocket, was intentionally destroyed seconds after launch on its maiden flight. Also destroyed was its cargo of four scientific satellites that were designed for studying how the Earth's magnetic field interacts with solar winds.

Cause: Shutdown occurred when the guidance computer tried to convert the sideways rocket velocity from 64-bits to a 16-bit format. The number was too big, and an overflow error resulted. When the guidance system shut down, control passed to an identical redundant unit, which also failed because it was running the same algorithm.

2. British Passports to Nowhere (1999)

Cost: £12.6 million, mass inconvenience

Disaster: The U.K. Passport Agency implemented a new Siemens computer system, which failed to issue passports on time for a half million British citizens. The Agency had to pay millions in compensation, staff overtime and umbrellas for people queuing in the rain for passports.

Cause: The Passport Agency rolled out its new computer system without adequately testing it or training its staff. At the same time, a law change required all children under 16 traveling abroad to obtain a passport, resulting in a huge spike in passport demand that overwhelmed the buggy new computer system.

3. Y2K (1999)

Cost: $500 billion

Disaster: One man's disaster is another man's fortune, as demonstrated by the infamous Y2K bug. Businesses spent billions on programmers to fix a glitch in legacy software. While no significant computer failures occurred, preparation for the Y2K bug had a significant cost and time impact on all industries that use computer technology.

Cause: To save computer storage space, legacy software often stored the year for dates as two digit numbers, such as "99″ for 1999. The software also interpreted "00″ to mean 1900 rather than 2000, so when the year 2000 came along, bugs would result

4. Cancer Treatment to Die For (2000)

Cost: Eight people dead, 20 critically injured

Disaster: Radiation therapy software by a company named "Multi-data Systems International" miscalculated the proper dosage, exposing patients to harmful and in some cases fatal levels of radiation. The physicians, who were legally required to double-check the software's calculations, were charged for murder.

Cause: The software calculated radiation dosage based on the order in which data was entered, sometimes delivering a double dose of radiation.

1.2.3. Common Causes of Software Errors

As software errors are the cause of poor software quality, it is important to investigate the causes of these errors in order to prevent them. A software error can be "code error", a "procedure error", a "documentation error", or a "software data error". It should be emphasized that the causes of all these errors are human, made by systems analysts, programmers, software testers, documentation experts, managers and sometimes clients and their representatives. Even in rare cases where software errors may be caused by the development environment (interpreters, wizards, etc.), it is reasonable to claim that it is human error that caused the failure of the development environment tool. The causes of software errors can be further classified as follows according to the stages of the software development process in which they occur.

1) **Faulty definition of requirements**

The faulty definition of requirements, usually prepared by the client, is one of the main causes of software errors. The most common errors of this type are:

• Erroneous definition of requirements.

• Absence of very important requirements.

• Incomplete definition of requirements.

For instance, one of the requirements of a municipality's local tax software system refers to discounts granted to various segments of the population: senior citizens, parents of large families, and so forth. Unfortunately, a discount granted to students was not included in the requirements document.

• Inclusion of unnecessary requirements, functions that are not expected to be needed in the near future.

2**) Client–developer communication failures**

Misunderstandings resulting from defective client–developer communication are additional causes for the errors that prevail in the early stages of the development process:

• Misunderstanding of the client's instructions as stated in the requirement document.

• Misunderstanding of the client's requirements changes presented to the developer in written form during the development period.

• Misunderstanding of the client's requirements changes presented orally to the developer during the development period.

• Misunderstanding of the client's responses to the design problems presented by the developer.

• Lack of attention to client messages referring to requirements changes and to client responses to questions asked by the developer on the part of the developer.

3) **Deliberate deviations from software requirements**

In several circumstances, developers may deliberately deviate from the documented requirements, actions that often cause software errors. The errors in these cases are byproducts of the changes. The most common situations of deliberate deviation are:

• The developer reuses software modules taken from an earlier project without sufficient analysis of the changes and adaptations needed to correctly fulfill all the new requirements.

• Due to time or budget pressures, the developer decides to omit part of the required functions in an attempt to cope with these pressures.

• Developer-initiated, unapproved improvements to the software, introduced without the client's approval, frequently disregard requirements that seem minor to the developer. Such "minor" changes may, eventually, cause software errors.

4) **Logical design errors**

Software errors can enter the system when the professionals who design the system (systems architects, software engineers, analysts, etc) formulate the software requirements.

Typical errors include:

• Algorithm errors

• Process definitions that contain sequencing errors.

For example, the software requirements for a company's debt-collection system define the debt-collection process as follows.

- ✓ "Once a client does not pay his debts, even after receiving three successive notification letters, the details are to be reported to the sales department manager who will decide whether to proceed to the next stage, referral of the client to the legal department". The systems analyst defined the process incorrectly by stating, "after sending three successive letters followed by no receipt of payment, the firm would include the name of the client on a list of clients to be handled by the legal department". The logical error was caused by the analyst's incorrect omission of the sales department phase within the debt collection process.

5) Coding errors

A broad range of reasons cause programmers to make coding errors. These include misunderstanding the design documentation, linguistic errors in the programming languages, errors in the application of CASE and other development tools, errors in data selection, and so forth.

6) **Non-compliance with documentation and coding instructions**

Almost every development unit has its own documentation and coding standards that define the content, order and format of the documents, and the code created by team members. To support this requirement, the team develops and publicizes its templates and coding instructions. Members of the development team or unit are required to comply with these requirements. Even if the quality of the "non-complying" software is acceptable, future handling of this software (by the development and/or maintenance teams) is expected to increase the rate of errors:

• Team members who need to coordinate their own codes with code modules developed by "non-complying" team members can be expected to encounter more than the usual number of difficulties when trying to understand the software developed by the other team members.

• Individuals replacing the "non-complying" team member (who has retired or been promoted) will find it difficult to fully understand his or her work. • The design review team will find it more difficult to review a design prepared by a noncomplying team.

• The test team will find it more difficult to test the module; consequently, their efficiency is expected to be lower, leaving more errors undetected.

7) Shortcomings of the testing process

Shortcomings of the testing process affect the error rate by leaving a greater number of errors undetected or uncorrected. These shortcomings result from the following causes:

• Incomplete test plans leave untreated portions of the software or the application functions and states of the system.

• Failures to document and report detected errors and faults.

- ✓ Failure to promptly correct detected software faults as a result of inappropriate indications of the reasons for the fault.
- ✓ Incomplete correction of detected errors due to negligence or time pressures.

8) Procedure errors

Procedures direct the user with respect to the activities required at each step of the process. They are of special importance in complex software systems where the processing is conducted in several steps, each of which may feed a variety of types of data and allow for examination of the intermediate results.

9) Documentation errors

The documentation errors that trouble the development and maintenance teams are errors in the design documents and in the documentation integrated into the body of the software. These errors can cause additional errors in further stages of development and during maintenance.

Another type of documentation error, one that affects mainly the users, is an error in the user manuals and in the “help” displays incorporated in the software. Typical errors of this type are:

• Omission of software functions

• Errors in the explanations and instructions given to users, resulting in “dead ends” or incorrect applications

• Listing of non-existing software functions, that is, functions planned in the early stages of development but later dropped, and functions that were active in previous versions of the software but cancelled in the current version

1.3. Measuring Quality

The five viewpoints of quality help us in understanding different aspects of the quality concept. On the other hand, measurement allows us to have a quantitative view of the quality concept.

Reasons for developing a quantitative view of quality

• Measurement allows us to establish baselines for qualities

• Developers must know the minimum level of quality they must deliver for a product to be acceptable

• Organizations make continuous improvements in their process models and an improvement has a cost associated with it. Organizations need to know how much improvement in quality is achieved at a certain cost incurred due to process improvement. This causal relationship is useful in making management decisions concerning process improvement.

• The present level of quality of a product needs to be evaluated so the need for improvements can be investigated

1.4. Software Quality Models and Standards

Different software quality models have been proposed to define quality and its related attributes. The most influential ones are the McCall’s model, ISO 9126 and the CMM.

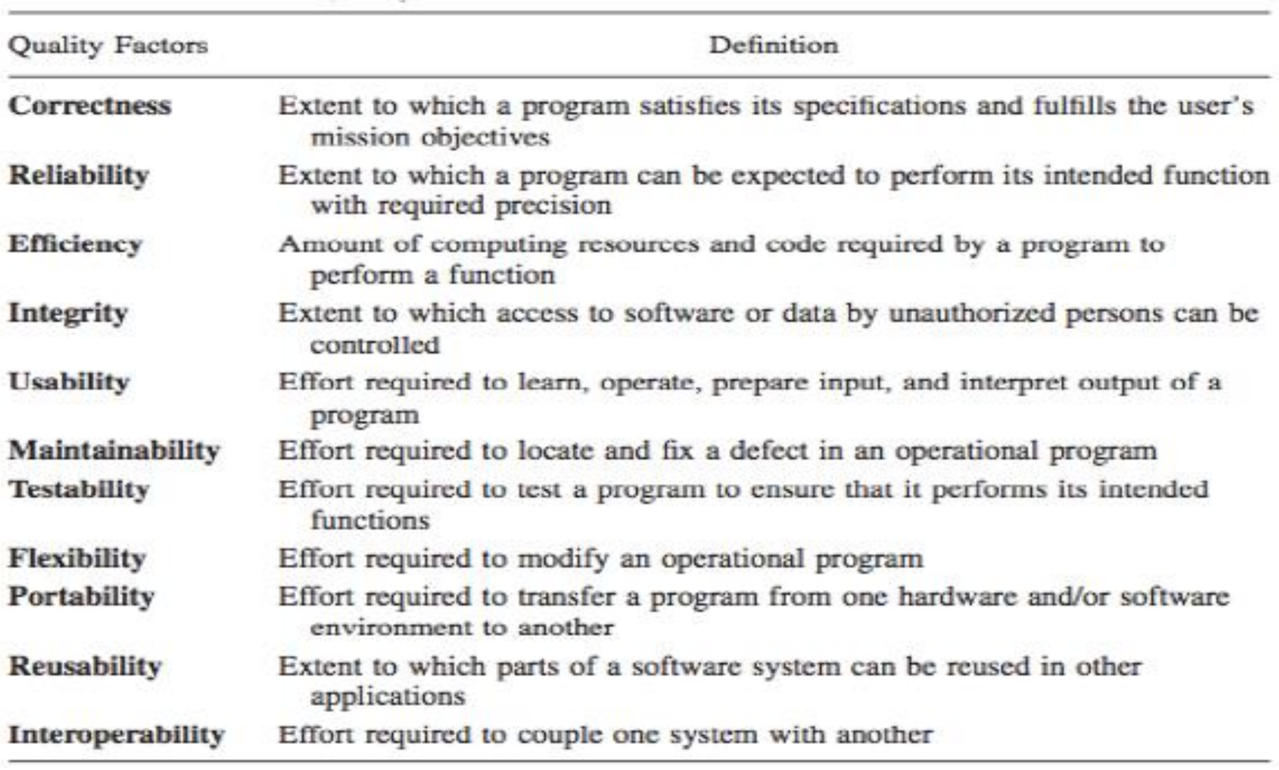
1.4.1. MCcall’s Quality Factors and Criteria

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to the mid-1970s. McCall, Richards, and Walters (1970) were the first to study the concept of software quality in terms of ***quality factors*** and ***quality criteria***.

1.4.1.1. Quality Factor

A *quality factor* represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *portability*, and *reusability*. A full list of the quality factors will be given in a later part of this section. As the examples show, quality factors are external attributes of a software system. Customers, software developers, and quality assurance engineers are interested in different quality factors to a different extent. For example, customers may want efficient and reliable software with less concern for portability. The developers strive to meet customer needs by making their system efficient and reliable, at the same time making the product portable and reusable to reduce the cost of software development.

The software quality assurance team is more interested in the testability of a system so that some other factors, such as correctness, reliability, and efficiency, can be easily verified through testing. The 11 quality factors in the McCall’s model are discussed briefly in the table below:



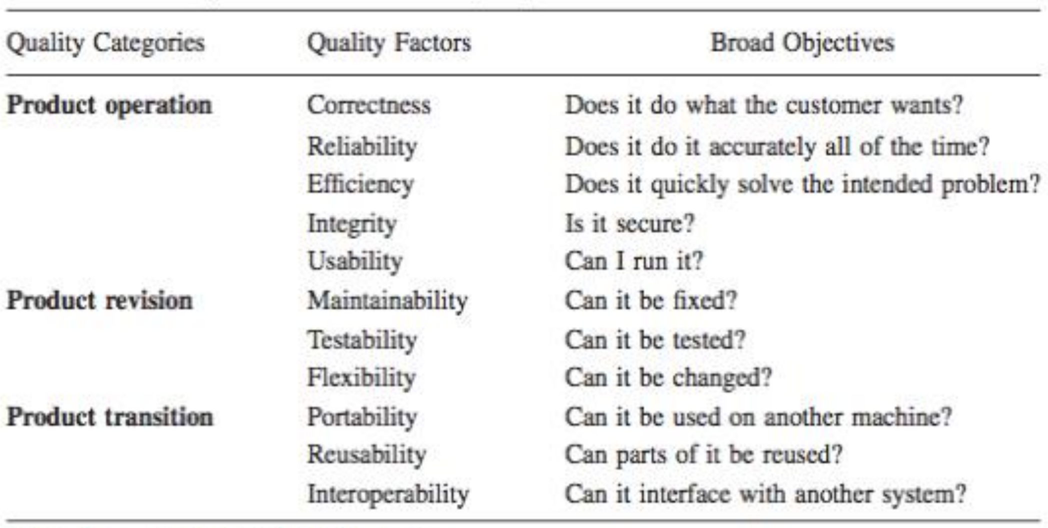
Quality Factors	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	Amount of computing resources and code required 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

Table 1: McCall's Quality Factors

The **11** quality factors defined in Table 1 above have been grouped into three broad categories named:

• Product operation
• Product revision
• Product transition

These three categories relate more to post development activities expectations and less to in development activities. McCall’s quality factors emphasize more on the quality levels of a product delivered by an organization and the quality levels of a delivered product relevant to product maintenance. The elements of each of the three broad categories are identified and further explained in the following Table 2.



Quality Categories	Quality Factors	Broad Objectives
Product operation	Correctness	Does it do what the customer wants?
	Reliability	Does it do it accurately all of the time?
	Efficiency	Does it quickly solve the intended problem?
	Integrity	Is it secure?
	Usability	Can I run it?
Product revision	Maintainability	Can it be fixed?
	Testability	Can it be tested?
	Flexibility	Can it be changed?
Product transition	Portability	Can it be used on another machine?
	Reusability	Can parts of it be reused?
	Interoperability	Can it interface with another system?

Table 2: Categories of McCall's Quality factors

1.4.1.2. Quality Criteria

A *quality criterion* is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. Highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system. Similarly, traceability of a user requirement allows developers to accurately map the requirement to a subset of the modules, thereby increasing the correctness of the system. Some quality criteria relate to products and some to personnel. For example, modularity is a product-related quality criterion, whereas training concerns development and software quality assurance employees. The relationship between quality factors and quality criteria is shown in Figure 1.

An arrow from a quality criterion to a quality factor means that the quality criterion has a positive impact on the quality factor. For example, *traceability* has a positive impact on *correctness*. Similarly, the quality criterion *simplicity* positively impacts *reliability*, *usability*, and t*estability*. Though it is desirable to improve all the quality factors, doing so may not be possible. This is because, in general, quality factors are not completely independent. Thus, we note two characteristics of the relationship as follows:

• If an effort is made to improve one quality factor, another quality factor may be degraded. For example, if an effort is made to make a software product testable, the efficiency of the software is likely to go down. An effort to make code portable is likely to reduce its efficiency. In fact, attempts to improve integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability will reduce the efficiency of a software system.

• Some quality factors positively impact others. For example, an effort to enhance the correctness of a system will increase its reliability. Similarly, an effort to enhance the testability of a system will improve its maintainability.

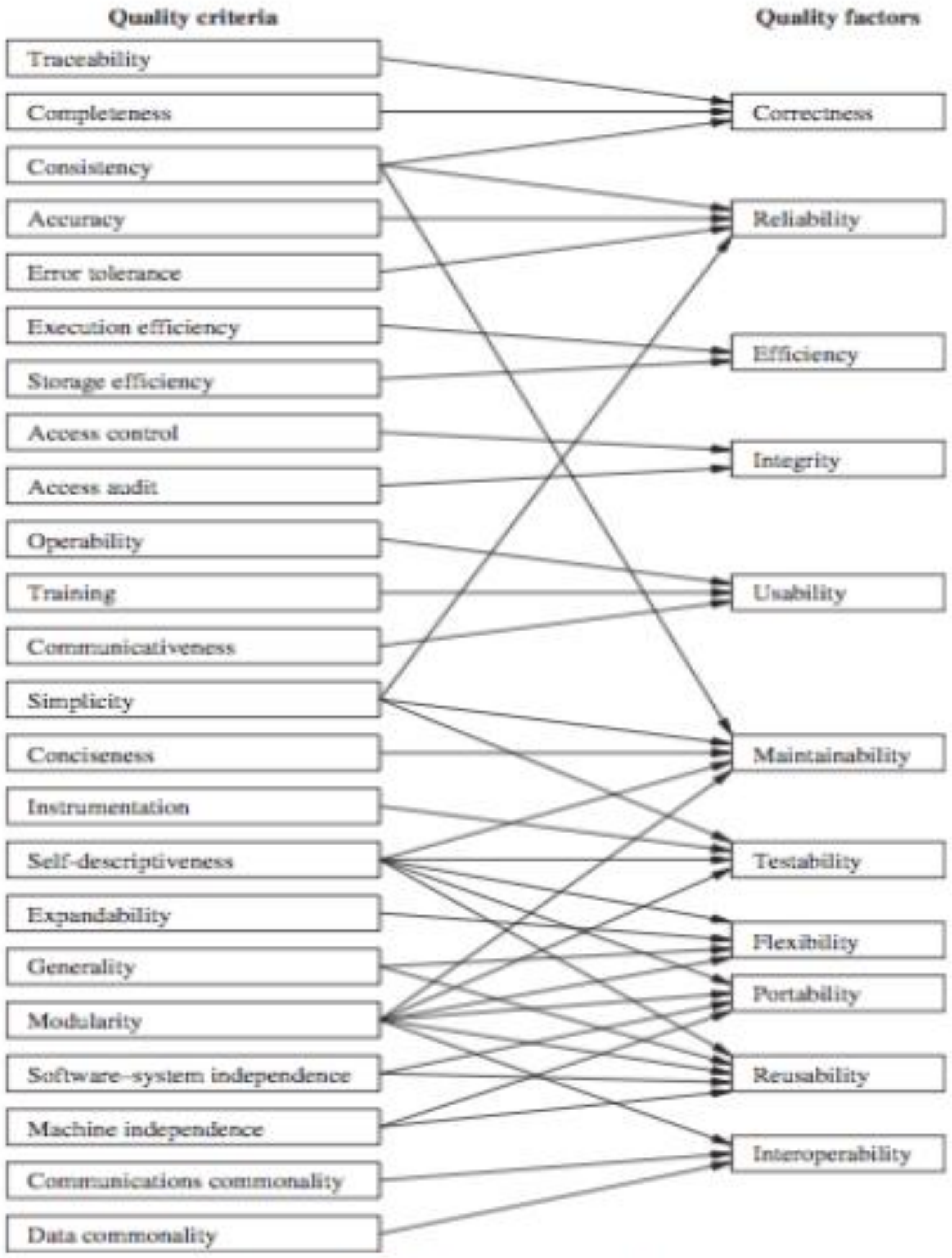


Figure 1: Relation between Quality Factors and Quality Criteria

Quality Metrics

The high-level quality factors cannot be measured directly. For example, we cannot directly measure the testability of a software system. Neither can testability be expressed in "yes" or "no" terms.

Instead, the degree of testability can be assessed by associating a few quality metrics with testability namely *simplicity*, *instrumentation*, *self-descriptiveness*, and *modularity*. A ***quality metric*** is a measure that captures some aspect of a quality criterion. One or more quality metrics should be associated with each criterion. The metrics can be derived as follows:

• Formulate a set of relevant questions concerning the quality criteria and seek a "yes" or "no" answer for each question.

• Divide the number of "yes" answers by the number of questions to obtain a value in the range of 0 to 1. The resulting number represents the intended quality metric.

Examples

We can ask the following question concerning the *self-descriptiveness* of a product: "*Is all documentation written clearly and simply such that procedures, functions, and algorithms can be easily understood?"* Another question concerning self-descriptiveness is: "*Is the design rationale behind a module clearly understood?"*. Different questions can have different degrees of importance in the computation of a metric, and, therefore, individual "yes" answers can be differently weighted in the above computation.

1.4.2. ISO 9126 Model Quality Characteristics

This model was defined by an expert group, under the guidance of the ISO (International Organization for Standardization). ISO 9126 defines six broad, independent categories of quality characteristics namely: *functionality, reliability, usability, efficiency, maintainability*, and *portability.*

Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

Reliability: A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.

Usability: A set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.

Efficiency: A set of attributes that bear on the relationship between the software's performance and the amount of resource used under stated conditions.

Maintainability: A set of attributes that bear on the effort needed to make specified modifications (which may include corrections, improvements, or adaptations of software to environmental changes and changes in the requirements and functional specifications).

Portability: A set of attributes that focus on the ability of software to be transferred from one environment to another (this includes the organizational, hardware or, software environment).

The ISO 9126 model followed the McCall's model and it categorizes the quality factors as characteristics and sub characteristics. Before using this model, organizations must define their own quality characteristics and sub-characteristics after a fuller understanding of their needs. In other words, organizations must identify the level of the different quality characteristics they need to satisfy within their context of software development.

Comparing the McCall's model and the ISO 9126 model, we can observe that both models have similar quality factors. The main difference that lies between the two models is the fact that the ISO 9126 model emphasizes characteristics *visible* to the users, whereas the McCall model considers *internal* qualities as well. For example, reusability is an internal characteristic of a product. Product developers strive to produce reusable components, whereas customers do not perceive its impact.

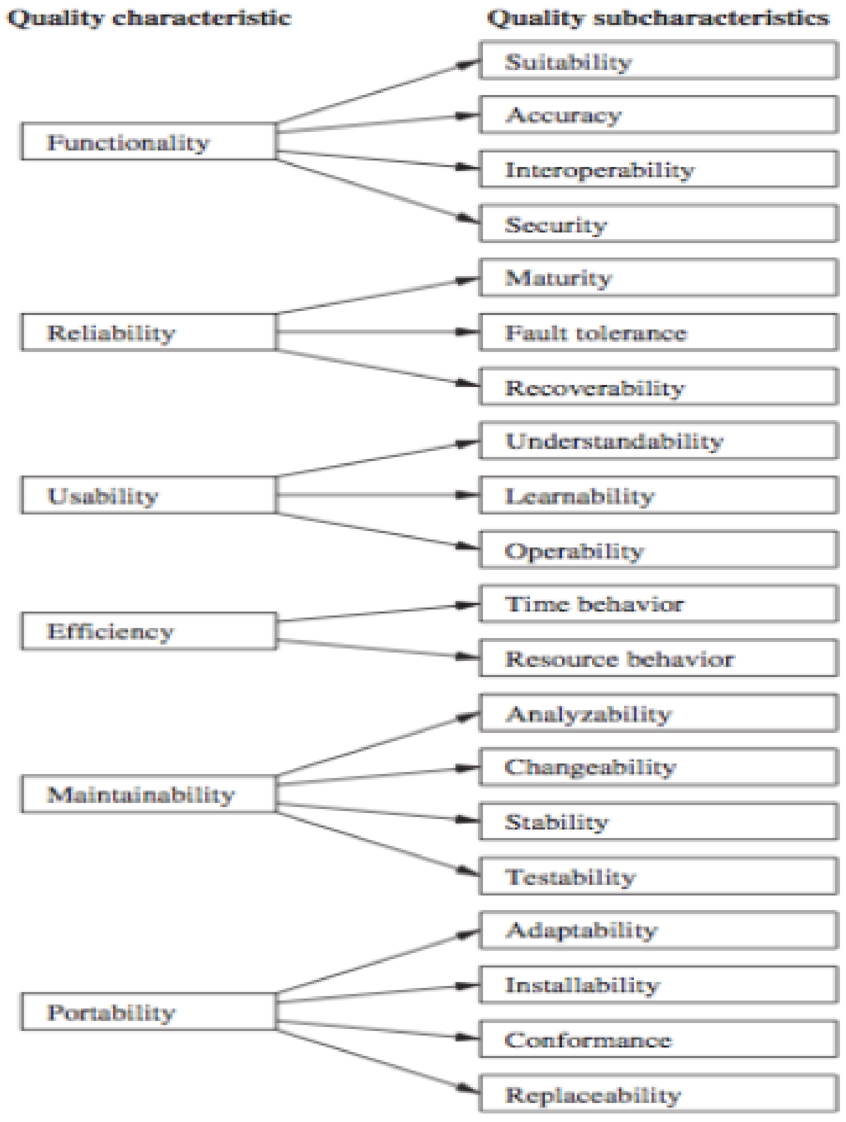


Figure 2: ISO 9126 Quality Characteristics and Sub Characteristics

1.4.3. Capability Maturity Model (CMM)

In software development processes we seek the following three desirable attributes:

• The products are of the highest quality
• Projects are completed according to their plans, including the schedules and budget
• Projects are completed within the allocated budgets

However, developers of large software products rarely achieve the above three attributes. Due to the complex nature of software systems, products are released with known and unknown defects. The unknown defects manifest themselves as unexpected failures during field operation, that is, when we know that the system was released with unknown defects. In many organizations,

software projects are often late and over budget. Due to the business angle in software development, it is important for the survival of organizations to develop low-cost, high-quality products within a short time. In order to move toward that goal, researchers and developers are devising new techniques and tools.
The maturity level of a development process tells us to what extent the organization is capable of producing low-cost, high-quality software. CMM is used to evaluate and improve the way software is built and maintained. First released in 1987, CMM was originally based on the experience of members of SEI. It is only a model, which is an abstract, general framework describing the processes used to develop and maintain software. The approach used by CMM is to describe the principles and leave their implementation up to the managers and technical staff of each organization, who will tailor CMM according to the culture and the experiences of their own environment.
The CMM defines five distinct levels of maturity, where *level 1* is the initial level and *level 5* is the highest level of process maturity. After evaluating the current maturity level of a development process, organizations can work on improving the process to achieve the next higher level of process maturity.
As an organization moves from one level to the next, its process capability improves to produce better quality software at a lower cost.

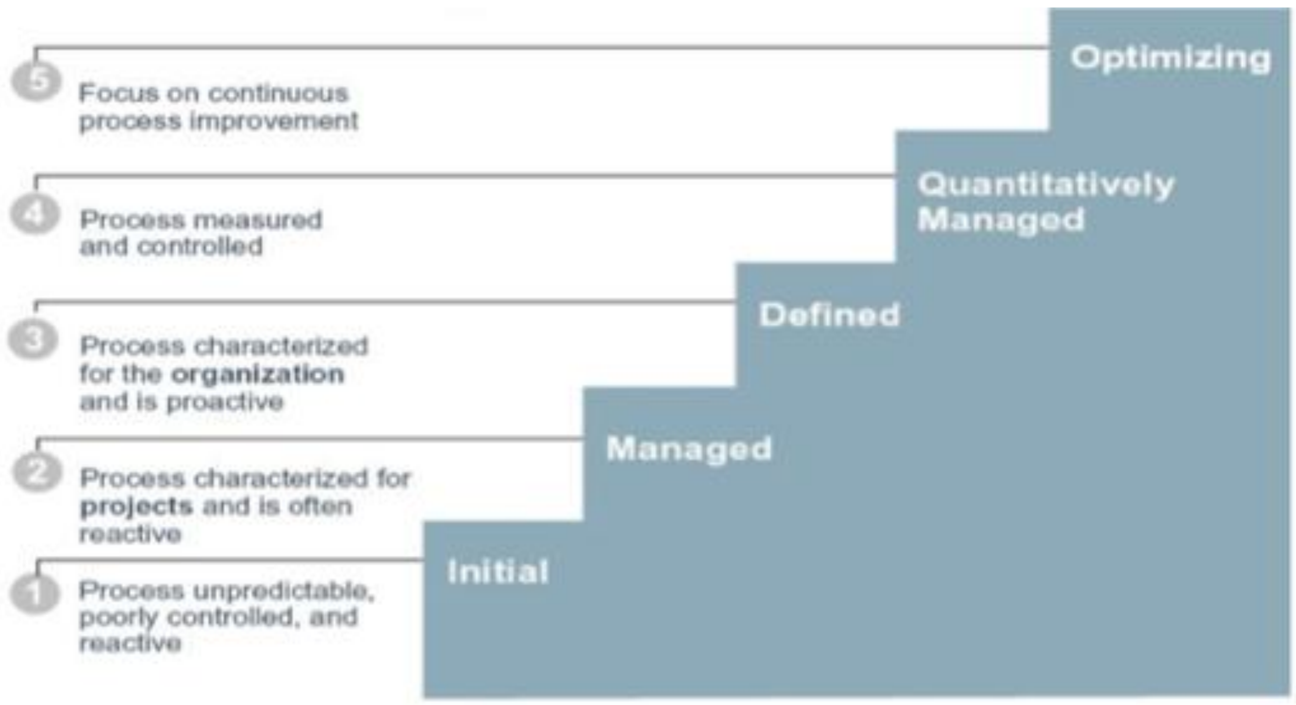

5
Focus on continuous process improvement
Optimizing
4
Process measured and controlled
Quantitatively Managed
3
Process characterized for the organization and is proactive
Defined
2
Process characterized for projects and is often reactive
Managed
1
Process unpredictable, poorly controlled, and reactive
Initial

Figure 3: CMM Maturity Levels
1. Initial level is described as ad hoc, poorly controlled, and often with unpredictable results in terms of schedule, effort, and quality.
2. At the repeatable level, the outputs of the process are consistent (in terms of schedule, effort, and quality) and basic controls are in place, but the processes that produce those results are not defined or understood.

3. Level 3, the defined level, is the point at which the software engineering practices that lead to consistent output are known and understood and are used across the whole organization.
4. The managed level, or **level 4**, is where the defined elements from level 3 are quantitatively instrumented so that level 5, the optimizing level, can be achieved.
5. Level 5, the optimizing level exists in organizations in which the development process operates smoothly as a matter of routine and continuous process improvement is conducted on the defined and quantified processes established in the previous levels.

Thus, CMM is seen as a maturity or growth model in which an organization works its way up the five levels and, even after having attained level 5, is still in the process of continually improving and maturing. Each of the five levels is also defined by the **key processes** associated with it. There are 18 key process areas that make up the five levels. These processes were chosen because of their effectiveness in improving an organization's software process capability. They are considered to be requirements for achieving a maturity level.
Managerial processes are those that primarily affect the way management operates to make decisions and control the project. **Technical processes** are those that primarily affect the way engineers operate to perform the technical and engineering work.
CMM provides a structure for each of the key process areas. Also, each key process area has one or more goals that are considered important for enhancing process capability.
Benefits of Using CMM
- It helps to establish a shared vision of purpose of process improvement within the organization
- It establishes common language for the software process
- It defines a set of priorities for handling problems
- It supports measurement through reliable assessment/appraisals and predefined objectives
- It also supports industry-wide comparisons