**UNIT-IV**

**CHAPTER-II**

**SOFTWARE RELIABILITY AND QUALITY MANAGEMENT**

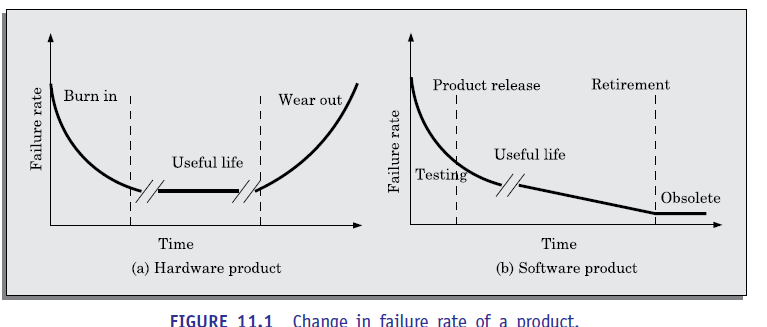
* The reliability of a software product essentially denotes its *trustworthiness or dependability*. Alternatively, the reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time
* Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced. It would have been very nice if we could mathematically characterize this relationship between reliability and the number of bugs present in the system using any simple closed form expression. Unfortunately, it is very diﬃcult to characterize the observed reliability of a system in terms of the number of latent defects in the system using a simple mathematical expression. To get an insight into this issue, consider the following.
* Removing errors from those parts of a software product that are very infrequently executed, makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90 per cent of the execution time of a typical program is spent in executing only 10 per cent of the instructions in the program. The *most used* 10 per cent instructions are often called the *core*1 of a program. The rest 90 per cent of the program statements are called *non-core* and are on the average executed only for 10 per cent of the total execution time. It therefore may not be very surprising to note that removing 60 per cent defects from the least used parts of a program would typically result in only 3 per cent improvement to the program reliability.
* Reliability also depends upon how the product is used, or on its *execution profile*. If the users execute only those features of a program that are “correctly” implemented, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low. Different categories of users of a software product typically execute different functions of a software product.
* For example, for a Library Automation Software the library members would use functionalities such as issue book, search book, etc., on the other hand the librarian would normally execute features such as create member, create book record, delete member record, etc. So defects which show up for the librarian, may not show up for the members.
* Suppose the functions of a Library Automation Software which the library members use are error-free; and functions used by the Librarian have many bugs. Then, these two categories of users would have very different opinions about the reliability of the software. Therefore, the reliability figure of a software product is observer-dependent and it is very diﬃcult to absolutely quantify the reliability of the product.
* Based on the above discussions, we can summaries the main reasons that make software reliability more diﬃcult to measure than hardware reliability:
* The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
* The perceived reliability of a software product is observer-dependent.
* The reliability of a product keeps changing as errors are detected and fixed.
* In the following subsection, we shall discuss why software reliability measurement is a harder prob

**Hardware *versus* Software Reliability**

* An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug. For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease

To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, the inter-failure times remain constant). On the other hand, the aim of software reliability study would be reliability growth (that is, increase in inter failure times) than hardware reliability measurement

* comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product are sketched in Figure 11.1. Observe that the plot of change of reliability with time for a hardware component [Figure 11.1(a)] appears like a “bath tub”. As shown in Figure 11.1(a), for a hardware system the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced.
* A hardware system then enters its useful life, where the rate of failure is almost constant. After some time (called *product life time*) the major components wear out, and the failure rate increases. The initial failures are usually covered through manufacturer’s warranty. A corollary of this observation (though a digression from our topic of discussion) is that it may be unwise to buy a product (even at a good discount to its face value) towards the end of its life time, That is, one need not feel happy to buy a ten-year old car at one-tenth of the price of a new car, since it would be near the rising edge of the bath tub curve, and one would have to spend unduly large time, effort, and money on repairing and end up as the loser.
* In contrast to the hardware products, software products show the highest failure rate just after purchase and installation [see the initial portion of the plot in Figure 11.1 (b)]. As the system is used, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete, no more error correction occurs and the failure rate remains unchanged



**Reliability Metrics of Software Products**

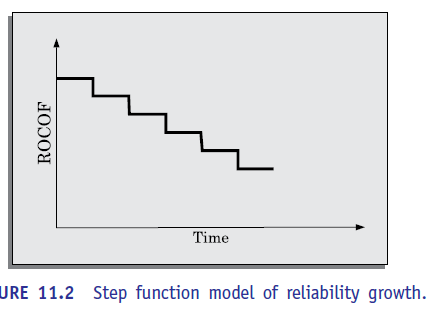
* The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the software requirements specification (SRS) document. In order to be able to do this, we need some metrics to quantitatively express the reliability of a software product. A good reliability measure should be observer-independent, so that different people can agree on the degree of reliability a system has. However, in practice, it is very diﬃcult to formulate a metric using which precise reliability measurement would be possible. In the absence of such measures, we discuss six metrics that are being used to measure software reliability.
* **Rate of occurrence of failure (ROCOF):** ROCOF measures the frequency of occurrence of failures. ROCOF of a software product is measured by dividing the total number of failures observed by the duration of observation. However, many software products do not run continuously (unlike a car or a mixer), but deliver certain service when a demand is placed on them. For example, a library software is idle until a book issue request is made. Therefore, for a typical software product such as a pay-roll software, applicability of ROCOF is limited.
* **Mean time to failure (MTTF):** MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for *n* failures. Let the failures occur at the time instants *t*1*, t*2*, ..., tn*. Then, MTTF can be calculated as 1
* It is important to note that only run time is considered in the time measurements. That is, the time for which the system is down to fix the error, the boot time, etc. are not taken into account in the time measurements and the clock is stopped at these times.
* **Mean time to repair (MTTR):** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
* **Mean time between failure (MTBF):** The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, the time measurements are real time and not the execution time as in MTTF.
* **Probability of failure on demand (POFOD):** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure. We have already mentioned that the reliability of most of the software products should be determined through specific service invocations, rather than making the software run continuously. Thus, POFOD metric is very appropriate for software products that are not required to run continuously.
* **Availability:** Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, operating systems, embedded controllers, etc., which are supposed to be never down and where repair and restart time are significant and loss of service during that time cannot be overlooked

**Shortcomings of reliability metrics of software products**

* All the above reliability metrics suffer from several shortcomings as far as their use in software reliability measurement is concerned. One of the reasons is that these metrics are centered around the probability of occurrence of system failures but take no account of the consequences of failures. That is, these reliability models do not distinguish the relative severity of different failures. Failures which are infrequent and whose consequences are not serious are in practice of little concern in the operational use of a software product.
* These types of failures can at best be minor irritants. On the other hand, more severe types of failures may render the system totally unusable. In order to estimate the reliability of a software product more accurately, it is necessary to classify various types of failures. Please note that the different classes of failures may not be mutually exclusive. The classification is based on widely different set of criteria. As a result, a failure type can at the same time belong to more than one class. A scheme of classification of failures is as follows:
* **Transient:** Transient failures occur only for certain input values while invoking a function of the system.
* **Permanent:** Permanent failures occur for all input values while invoking a function of the system.
* **Recoverable:** When a recoverable failure occurs, the system can recover without having to shut down and restart the system (with or without operator intervention).
* **Unrecoverable:** In unrecoverable failures, the system may need to be restarted.
* **Cosmetic:** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the situation where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface

**Reliability Growth Modelling**

* A reliability growth model is a mathematical model of how the reliability of a software product improves as errors are detected and repaired. Although several different reliability growth models have been proposed, in this text we discuss only three of those.
* **Jelinski and Moranda model (1972)**
* The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Therefore, perfect error fixing is implicitly assumed in this model. Another implicit assumption in this model is that all errors contribute equally to reliability growth (reflected in equal step size).
* Both these two assumptions are rather unrealistic since different errors contribute differently to reliability growth and also error fixes may not be perfect in the sense that an error fix may create other types of errors. Typical reliability growth predicted using this model has been shown in Figure 11.2.

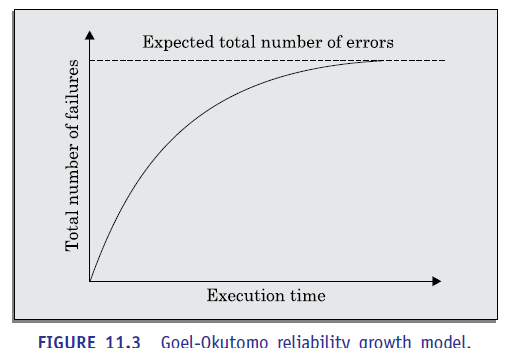


**Littlewood and Verall’s model**

* The Littlewood and Verall’s model is an improvement over the Jelinsky an Moranda step function model in the sense that it allows for negative reliability growth. In the Jelinsky and Moranda model, whenever a failure occurs, the reliability improves by a constant amount because it is assumed that the bug fix is perfect and removes the defect causing the failure. However, in reality when a bug fix is carried out, it may introduce additional errors, and thereby result in a lower reliability for the software. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases.
* It treats an error’s contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

**Goel-Okutomo model**

* In this model, it is assumed that the execution times between the failures are exponentially distributed. The cumulative number of failures, at any time, can therefore be given in terms of *m*(*t*), the expected value of failures between time *t* and time *t* + D*t*. It is assumed that the reliability growth follows a non-homogeneous poisson process (NHPP). That is, the number of error occurrences that are causing failures during the time duration
* *t* to *t* + D*t* is proportional to the expected number of undetected errors existing at time *t*. Once a failure has been detected, it is assumed that the error correction is perfect and immediate. The number of failures at time *t* can be given by, *m*(*t*) = *N*(1 – *e*–*bt*), where *N* is expected total number of defects in the code and *b* is the rate at which the failure rate decreases. The change of the number of failures over time has been plotted graphically in Figure 11.3.



**STATISTICAL TESTING**

* Statistical testing is a testing process whose objective is to determine the reliability of the product rather than discovering errors. The test cases designed for statistical testing with an entirely different objective from those of conventional testing. To carry out statistical testing, we need to first define the operation profile of the product
* **Operation profile:** Different categories of users may use a software product for very different purposes. For example, a librarian might use the Library Automation Software to create member records, delete member records, add books to the library, etc., whereas a library member might use software to query about the availability of a book, and to issue and return books.
* Formally, we can define the operation profile of a software as the probability of a user selecting the different functionalities of the software. If we denote the set of various functionalities offered by the software by { *fi*}, the operational profile would associate with each function {*fi*} with the probability with which an average user would select { *fi*} as his next function to use. Thus, we can think of the operation profile as assigning a probability value *pi* to each functionality *fi* of the software.

**How to define the operation profile for a product?**

* We need to divide the input data into a number of input classes. For example, for a graphical editor software, we might divide the input into data associated with the edit, print, and file operations. We then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected. The operation profile of a software product can be determined by observing and analyzing the usage pattern of the software by a number of users.

**Steps in Statistical Testing**

* The first step is to determine the operation profile of the software.
* The next step is to generate a set of test data corresponding to the determined operation profile.
* The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.
* For accurate results, statistical testing requires some fundamental assumptions to be satisfied. It requires a statistically significant number of test cases to be used. It further requires that a small percentage of test inputs that are likely to cause system failure to be included. Now let us discuss the implications of these assumptions. It is straight forward to generate test cases for the common types of inputs, since one can easily write a test case generator program which can automatically generate these test cases.
* However, it is also required that a statistically significant percentage of the unlikely inputs should also be included in the test suite. Creating these unlikely inputs using a test case generator is very diﬃcult and may have to be manually designed.

**Pros and cons of statistical testing**

* Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users can find to be more reliable (than actually it is!). Also, the reliability estimation arrived by using statistical testing is often more accurate compared to those of other methods discussed.
* However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. There is no simple and repeatable way of defining operation profiles. Also, the number of test cases with which the system is to be tested should be statistically significant

**SOFTWARE QUALITY**

* Traditionally, the quality of a product is defined in terms of its *fitness of purpose*. That is, a good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.

“fitness of purpose” is not a wholly satisfactory definition of quality for software products. To give an example as to why this is so, consider a software product that is functionally correct.

* That is, it correctly performs all the functions that have been specified in its SRS document. Even though it may be functionally correct, it may not be considered it to be a quality product, if it has an almost unusable user interface.
* Another example is that of a product which does everything that the users wanted but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.
* Unlike hardware products, software lasts a long time, in the sense that it keeps evolving to accommodate changed circumstances. The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:
* **Portability:** A software product is said to be *portable*, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.
* **Usability:** A software product has good *usability*, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.
* **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
* **Correctness:** A software product is *correct*, if different requirements as specified in the SRS document have been correctly implemented.
* **Maintainability:** A software product is *maintainable*, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

**Software Quality Models**

* Software quality is often considered transcendental. That is, the quality of a software product can be felt, but cannot be seen or measured. However, the need to be able to quantitatively measure the quality of a software is often felt.
* For example, one may want to set quantitative quality requirements for a software, or to verify whether a software meets the quality requirements set of it.
* Unfortunately, it is hard to directly measure the quality of a software. However, it can be expressed in terms of several attributes of a software that can be directly measured.
* In a hierarchical quality model, the bottom level of the hierarchy can be directly measured, thereby enabling a quantitative assessment of the quality of a software. There are several well-established quality models, including McCall’s, Dromey’s, and Boehm’s.
* Since there was no standardization among the large number of quality models that became available, the ISO 9126 model of quality was developed. We briefly discuss Garvin’s, McCall’s, Dromey’s, Boehm’s quality model, and ISO 9126.

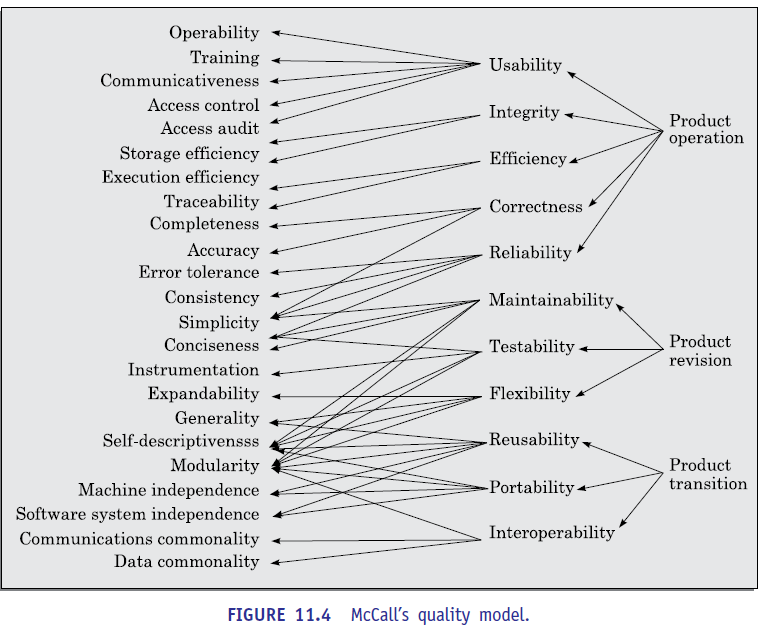
**Garvin’s quality dimensions**

* David Garvin, a professor of Havard Business School, in his book *Total Quality Management*, defined the quality of any product in terms eight general attributes of the product, some of these are measurable and some are not. Garvin reasoned that sometimes users have subjective judgment of the quality of a program (perceived quality) that must be taken into account to judge its quality.
* **Performance:** How well it performs the jobs
* **Features:** How well it supports the required features
* **Reliability:** Probability of a product working satisfactorily within a specific period of time
* **Conformance:** Degree to which the product meets the requirements
* **Durability:** Measure of product life
* **Serviceability:** Speed and effectiveness maintenance
* **Aesthetics:** The look and feel of the product

**Perceived quality:** User’s opinion about the product quality

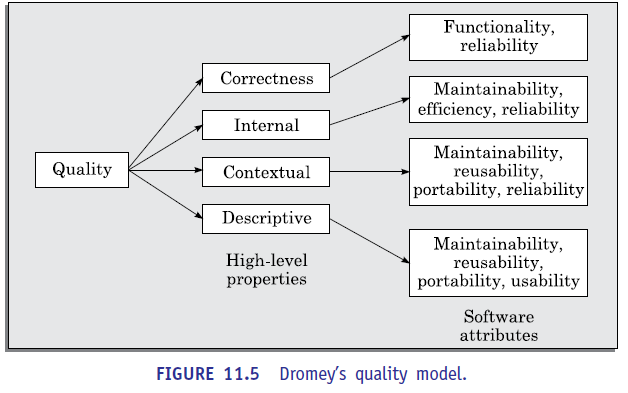
**McCall’ model**

* Jim McCall’s quality model is given in terms of several quality factors that reflect both the users’ and the developers’ priorities. McCall defined the quality of a software in terms of three broad parameters: its operational characteristics (product operations), how easy it is to fix defects (product revision), and how easy it is to port it to different platforms (product transition).
* These three high-level quality attributes are given in terms of eleven quality factors. These eleven quality factors describe the external view of the software, or the quality as perceived by the users. These are then given in terms of 23 quality criteria that describe the internal view of the software or as seen by the developers. The quality factors cannot be measured directly, but can be measured only indirectly through the quality criteria (internal view). The quality criteria can be measured directly. In the following, we briefly describe the eleven quality factors:
* **Correctness:** The extent to which a software product satisfies its specifications
* **Reliability:** The probability of the software product working satisfactorily over a given duration
* **Efficiency:** The amount of computing resources required to perform the required functions
* **Integrity:** The extent to which the data of the software product remains valid
* **Usability:** The effort required to operate the software product
* **Maintainability:** The ease with which it is possible to locate and fix bugs in the software product
* **Flexibility:** The effort required to adapt the software product to changing requirements
* **Testability:** The effort required to test a software product to ensure that it performs its intended function
* **Portability:** The effort required to transfer the software product from one hardware or software system environment to another
* **reusability:** The extent to which a software can be reused in other applications
* **Interoperability:** The effort required to integrate the software with other software The McCall’s quality model is given in Figure 11.4 that shows the quality characteristics in terms of eleven quality factors. The eleven quality factors are given in terms of 23 quality criteria.



**Dromey’s model**

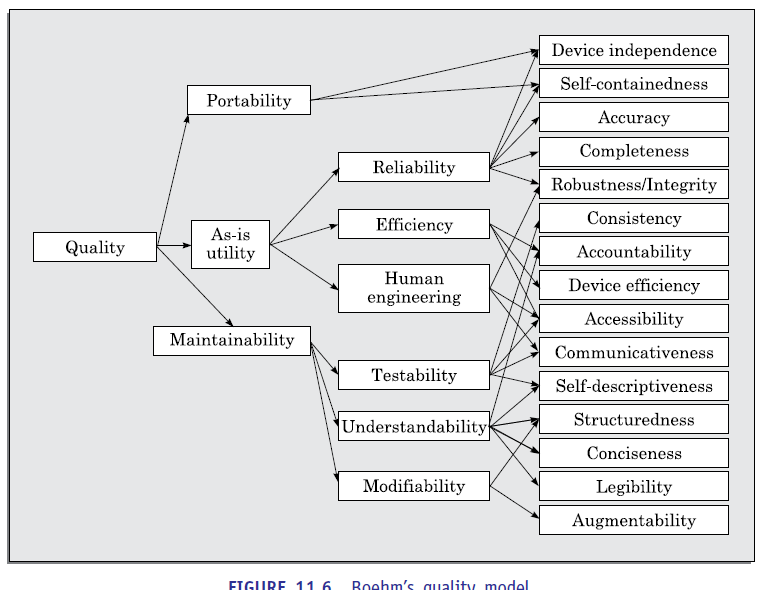
* Dromey proposed that software product quality depends on four major high-level properties of the software: correctness, internal characteristics, contextual characteristics, and certain descriptive properties. Each of these high-level properties of a software product in turn depends on several lower-level quality attributes of the software. Dromey’s hierarchical quality model has been shown in Figure 11.5. The software attributes are directly measurable.
* The high-level properties can be inferred from these. The software quality, in turn, can be inferred from the high-level properties.



**Boehm’s model**

Boehm postulated that the quality of a software could be defined based on three high-level characteristics that are important for the users of the software. These three high-level characteristics are as follows:

* **As-is utility:** How well (easily, reliably, efficiently) can it be used
* **Maintainability:** How easy is it to understand, modify and then retest the software
* **Portability:** How difficult would it be to make the software in a changed environment
* Boehm expressed these high-level product quality attributes in terms of several measurable product attributes. As compared to McCall’s and Dromey’s quality models, Boehm’s quality model is based on a wider range of software attributes and with greater focus on software maintainability.



**ISO 9126**

* It identifies six major external quality characteristics. Each of these characteristics is expressed in terms of a set of sub-characteristics. The six major external quality characteristics are as follows:

**Functionality:** It relates to the correctness of the developed functionalities

**Reliability:** It relates to the capability to maintain the required level of performance

**Usability:** It relates to the effort needed to be able to use the software

**Efficiency:** It relates to the usage of physical resources by the software during its execution

**Maintainability:** It relates the effort needed to make changes to the software

**Portability:** It relates to the effort needed to transfer the software to different environments

* Each sub-characteristic is related to exactly one quality characteristic. This is in contrast to the McCall’s quality attributes that are heavily interrelated. Another difference is that the quality characteristics strictly refer to a software product, whereas McCall’s attributes capture process quality issues as well.

**SOFTWARE QUALITY MANAGEMENT SYSTEM**

* A quality management system (often referred to as *quality system*) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. In the following subsections, we briefly discuss some of the important issues associated with a quality system:

**Managerial structure and individual responsibilities**

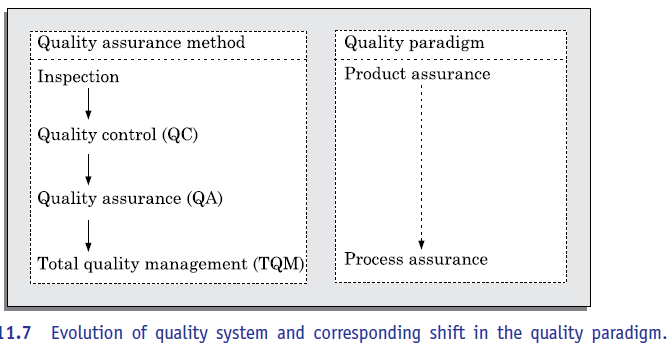
* A quality system is the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities.
* The quality system of an organization should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

**Quality system activities**

* The quality system activities encompass the following:
* Auditing of projects to check if the processes are being followed.
* Collect process and product metrics and analyse them to check if quality goals are being met.
* Review of the quality system to make it more effective.
* Development of standards, procedures, and guidelines.
* Produce reports for the top management summarizing the effectiveness of the quality system in the organisation.
* A good quality system must be well documented. Without a properly documented quality system, the application of quality controls and procedures become *ad hoc*, resulting in large variations in the quality of the products delivered. Also, an undocumented quality system sends clear messages to the staff about the attitude of the organization towards quality assurance. International standards such as ISO 9000 provide guidance on how to organise a quality system.

**Evolution of Quality Systems**

* Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products and eliminate defective products.
* For example, a company manufacturing nuts and bolts would inspect its finished goods and would reject those nuts and bolts that are outside certain specified tolerance range. Since that time, quality systems of organizations have undergone four stages of evolution as shown in Figure 11.7.
* The initial product inspection method gave way to quality control (QC) principles. Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products. The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.
* The modern quality assurance paradigm includes guidance for recognizing, defining, analysing, and improving the production process.
* *Total quality management* (TQM) advocates that the process followed by an organization must continuously be improved through process measurements. TQM goes a step further than quality assurance and aims at *continuous process improvement.* TQM goes beyond documenting processes to optimizing them through redesign.
* A term related to TQM is *business process re-engineering* (BPR), which is aims at re-engineering the way business is carried out in an organization, whereas our focus in this text is reengineering of the software development process. From the above discussion, we can say that over the last six decades or so, the quality paradigm has shifted from product assurance to process assurance



**Product Metrics *versus* Process Metrics**

* All modern quality systems lay emphasis on collection of certain product and process metrics during product development. The users assess the quality of a software product based on its external attributes, whereas during development, the developers assess the product’s quality based on various internal attributes. We can also say that during development, the developers can ensure the quality of a software product based on measurement of the relevant internal attributes.
* The internal attributes may measure either some aspects of the product or of the development process (called *process metrics*). Let us first understand the basic differences between product and process metrics. Product metrics help to measure the characteristics of a product being developed.
* A few examples of product metrics and the specific product characteristics that they measure are as follows:
* LOC and function point metrics are used to measure size
* Person month (PM) metric is used to measure the effort required to develop thesoftware
* Time required to develop the product is measured in months
* Process metrics help to measure how a development process is performing.

Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, and the number of latent defects per line of code in the developed product

**ISO 9000**

International standards organization (ISO) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987.

**What is ISO 9000 Certiﬁcation?**

* ISO 9000 certification serves as a reference for contract between independent parties. In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not. In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organization applies to all its activities related to its products or services.
* The ISO standard addresses both operational aspects (that is, the process) and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 makes a set of recommendations for repeatable and high-quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.
* ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO 9003.
* The types of software companies to which the different ISO standards apply are as follows:
* **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.
* **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organizations.
* **ISO 9003:** This standard applies to organizations involved only in installation and testing of products

**ISO 9000 for Software Industry**

* ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company. Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very diﬃcult to interpret them in the context of software development organizations.
* An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of product manufacturing activities. Two major differences between software development and development of other kinds of products are as follows:
* Software is intangible and therefore diﬃcult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is diﬃcult to control and manage anything that you cannot see and feel. In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to accurately determine how much work has been completed and to estimate how much more time will it take.
* During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. As an example, consider a steel making company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organisations.

**SEI CAPABILITY MATURITY MODEL**

* SEI *capability maturity model* (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book “Quality is Free” [Crosby, 1979].
* The United States Department of Defense (US DoD) is the largest buyer of software product. It often faced diﬃculties in vendor performances, and had to many times live with low quality products, late delivery, and cost escalations. In this context, SEI CMM was originally developed to assist the US Department of Defense (DoD) in software acquisition.
* The rationale was to include the likely contractor performance as a factor in contract awards. Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts. It was observed that the SEI CMM model helped organizations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits. Gradually many commercial organizations that are not contractors of the US DoD began to adopt CMM as a framework for their own internal improvement initiatives.
* In simple words, CMM is a reference model for apprising the software process maturity into different levels. This can be used to predict the most likely outcome to be expected from the next project that the organisation undertakes. It must be remembered that SEI CMM can be used in two ways—capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result.
* Capability evaluation provides a way to assess the software process capability of an organization. Capability evaluation is administered by the contract awarding authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work.
* On the other hand, software process assessment is used by an organization with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.
* The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system starting from scratch. SEI CMM classifies software development in

**Level 1: Initial**

* The initial level places no specific requirements on an organization. Therefore, a software development organization at this level is characterized by *ad hoc* activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic.
* Therefore, it is also called *chaotic level*. The success of projects depends on individual efforts and heroics. When a developer leaves the organization, the successor would have great diﬃculty in understanding the process that was followed and the work completed. Also, no formal project management practices are followed. As a result, time pressure builds up towards the end of the delivery time, and short-cuts are tried out to meet the deadline leading to low quality products. Industries into the following five maturity levels:

**Level 2: Repeatable**

* At this level, the basic project management practices such as tracking cost and schedule are established in the organization. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis, COCOMO, etc., are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Though there is a rough understanding among the developers about the process being followed, the process is not documented.
* Configuration management practices are used for all project deliverables. Please remember that opportunity to repeat a process exists only when a company produces a family of products. Since the products are very similar, the success story on development of one product can repeated for another.
* In a non-repeatable software development organisation, a software product development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals.
* On the other hand, in a non-repeatable software development organization, the chances of successful completion of a software project is to a great extent depends on who the team members are. For this reason, the successful development of one product by such an organization does not automatically imply that the next product development will be successful

**Level 3: Defined**

* At this level, the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured. The organization builds up the capabilities of its employees through periodic training programs. Also, review techniques are emphasized and documented to achieve phase containment of errors. ISO 9000 aims at achieving this level.

**Level 4: Managed**

* At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. In other words, the results of process measurements are used to evaluate project performance rather than improve the process.

**Level 5: Optimising**

* At this stage, both process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it is found that the code reviews are not very effective and a large number of errors are detected only during the unit testing, then the process would be fine tuned to make the reviews more effective. Also, the lessons learned from specific projects are incorporated into the process.
* Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from process measurements and also, from application of innovative ideas and technologies. At CMM level 5, an organization would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organization-wide. Level 5 organizations usually have a department whose sole responsibility is to assimilate latest tools and technologies and propagate them organization-wide.
* Since the process changes continuously, it becomes necessary to effectively manage a changing process. Therefore, level 5 organisations use configuration management techniques to manage process changes.
* Except for level 1, each maturity level is characterized by several Key Process Areas (KPAs). The KPAs for a level indicate the areas that an organization at the immediate lower level should focus to improve its software process to this level. Each of the focus areas identifies a number of key practices or activities that need to be implemented. In other words, we can say that KPAs capture the focus areas of a level
* SEI CMM provides a list of key areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one stage enhances the capability already built up. It is implicit an organization at some level trying to acquire the competency required of much higher level may be counterproductive.
* For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes diﬃcult to follow the defined process due to schedule and budget pressures.
* Substantial evidence has now been accumulated which indicate that adopting SEI CMM has several business benefits. However, the organizations trying out the CMM frequently face a problem that stems from some of the characteristics of the CMM itself. We elaborate these aspects in the following.

**CMM Shortcomings:** CMM does suffer from several shortcomings. The important among these are the following:

* The most frequent complaint by organizations while trying out the CMM-based process improvement initiative is that they understand *what is needed to be improved*, but they need more guidance about *how to improve it.*
* Another shortcoming (that is common to ISO 9000) is that thicker documents, more detailed information, and longer meetings are considered to be better. This is in contrast to the accepted agile practices—reducing complexity and keeping the documentation to the minimum without sacrificing the relevant details.
* Getting an accurate measure of an organization's current maturity level is also an issue. The CMM takes an activity-based approach to measuring maturity; if you do the prescribed set of activities then you are at a certain level. There is nothing that characterizes or quantifies whether you do these activities well enough to deliver the intended results.

**Comparison Between ISO 9000 Certiﬁcation and SEI/CMM**

* Let us compare some of the key characteristics of ISO 9000 certification and the SEI CMM model for quality appraisal:
* ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organization in oﬃcial documents, communication with external parties, and in tender quotations. However, SEI CMM assessment is purely for internal use.
* SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
* SEI CMM goes beyond quality assurance and prepares an organization to ultimately achieve TQM. In fact, ISO 9001 aims at level 3 of SEI CMM model.
* SEI CMM model provides a list of key process areas (KPAs) on which an organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement. In contrast, an organization adopting ISO 9000 either qualifies for it or does not qualify.