SOFTWARE ENGINEERING
UNIT-5

1. **Software Reliability And Quality Management:** Software Reliability,
2. Statistical Testing,
3. Software Quality,
4. Software Quality Management System,
5. ISO 9000,
6. SEI Capability Maturity Model.
7. **Computer Aided Software Engineering:** Case and its Scope,
8. Case Environment,
9. Case Support in Software Life Cycle,
10. Other Characteristics of Case Tools,
11. Towards Second Generation CASE

Tool,

12. Architecture of a Case Environment

SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

- Reliability of a software product is an important concern for most users.
- Users not only want the products they purchase to be highly reliable product.
- This may especially be true for safety-critical and embedded software products.
- It is very difficult to accurately measure the reliability of any software product.
- One of the main problems encountered while quantitatively measuring the reliability of a software product is the fact that reliability is observer-dependent. That is, different groups of users may arrive at different reliability estimates for the same product.
- Besides this, several other problems (such as frequently changing reliability values due to bug corrections) make accurate measurement of the reliability of a software product difficult.
- Even though no entirely satisfactory metric to measure the reliability of a software product exists,
- Software quality assurance (SQA) has emerged as one of the most talked about topics in recent years in software industry circle.
- The major aim of SQA is to help an organization **develop high quality software products** in a repeatable manner.
- A software development organization can be called *repeatable* when its software development process is person-independent. That is, the success of a project does not depend on who exactly are the team members of the project.
- Besides, the quality of the developed software and the cost of development are important issues addressed by SQA.

1. SOFTWARE RELIABILITY

- The reliability of a software product essentially denotes its *trustworthiness or dependability*.
- **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 a 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.
- 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 analysing the behaviour 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.*
- 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.
- 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.
- 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.

Hardware versus Software Reliability

- Hardware components fail due to very different reasons as compared to software components.
- Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.
- 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 (reliability may decrease if a bug fix introduces new errors).
- 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).
- A 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.

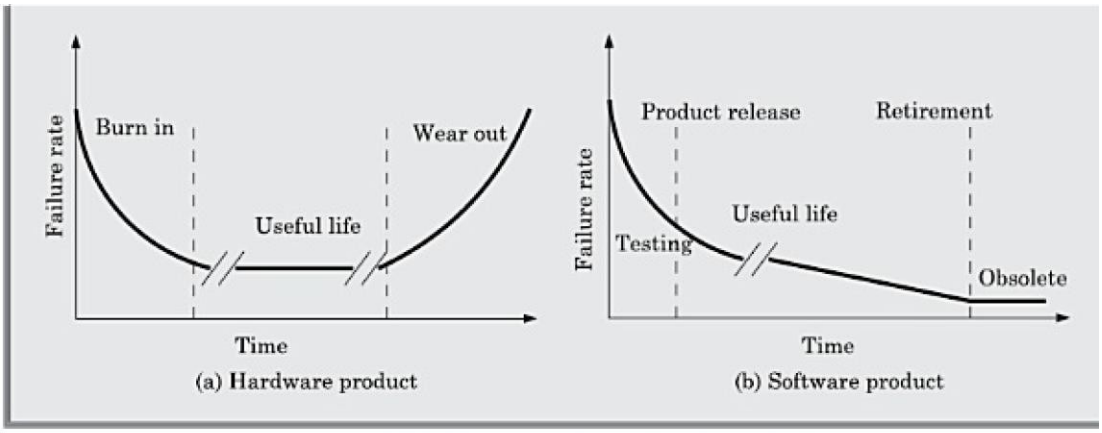

Figure 11.1: Change in failure rate of a product.

- Observe that the plot of change of reliability with time for a hardware component (Figure 11.1(a)) appears like a "bath tub". For a software component the failure rate is

initially high, but decreases as the faulty components identified are either repaired or replaced.

- The 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.
- 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, the software product 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.
- Six metrics that show the reliability as follows:
- **Rate of occurrence of failure (ROCOF):** ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.
- **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 t1, t2, ..., tn. Then, MTTF can be calculated as. 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.
    - 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, and operating systems, and 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 transient and whose consequences are not serious are in practice of little concern in the operational use of a software product.

- 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 shutdown 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 software reliability improves as errors are detected and repaired.
- A reliability growth model **can be used to predict when a particular level of reliability** is likely to be attained.
- Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.
- Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.
- **Jelinski and Moranda model :** 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. Such a model is shown in Figure 11.2.

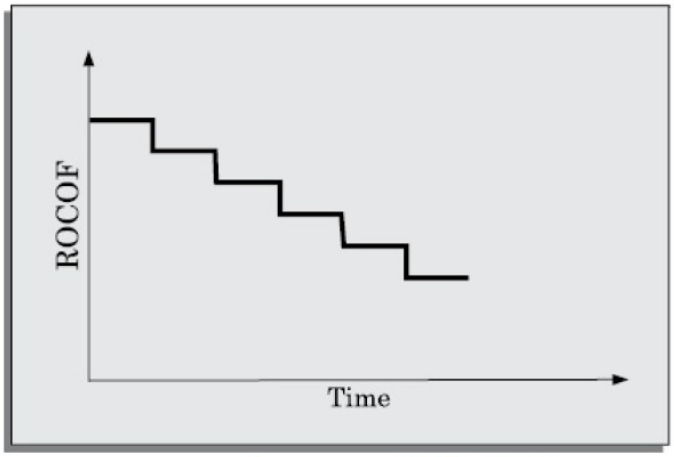

Figure 11.2: Step function model of reliability growth.

However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since we already know that correction of different errors contribute differently to reliability growth.

Figure 11.2: Step function model of reliability growth.

- **Littlewood and Verall's model :** This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors.
    - 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.

2. 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.

- **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 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 the number of test cases with which the system is to be tested should be statistically significant.

3. 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 of 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, we cannot consider it to be a quality product, if it has an almost unusable user interface.
- 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.

McCall's quality factors :

- McCall distinguishes two levels of quality attributes [McCall]. The higherlevel attributes, known as quality factor s or external attributes can only be measured indirectly.
- The second-level quality attributes are called quality criteria.
- Quality criteria can be measured directly, either objectively or subjectively.
- By combining the ratings of several criteria, we can either obtain a rating for the quality factors, or the extent to which they are satisfied.
- For example, the reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time. Thus, reliability is a higher-level quality factor and number of defects is a low-level quality factor.

ISO 9126 :

- ISO 9126 defines a set of hierarchical quality characteristics.

4. SOFTWARE QUALITY MANAGEMENT SYSTEM

- A quality management system (often referred to as quality system) is the principal methodology used by organisations to ensure that the products they develop have the desired quality.
- In the following, some of the important issues associated with a quality system:
- **Managerial structure and individual responsibilities :** A quality system is the responsibility of the organisation as a whole. However, every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation 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 summarising the effectiveness of the quality system in the organisation.
    - A good quality system must be well documented.

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 to 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 organisations have undergone four stages of evolution as shown in Figure 11.3.
    - The initial product inspection method gave way to quality control (QC) principles. Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

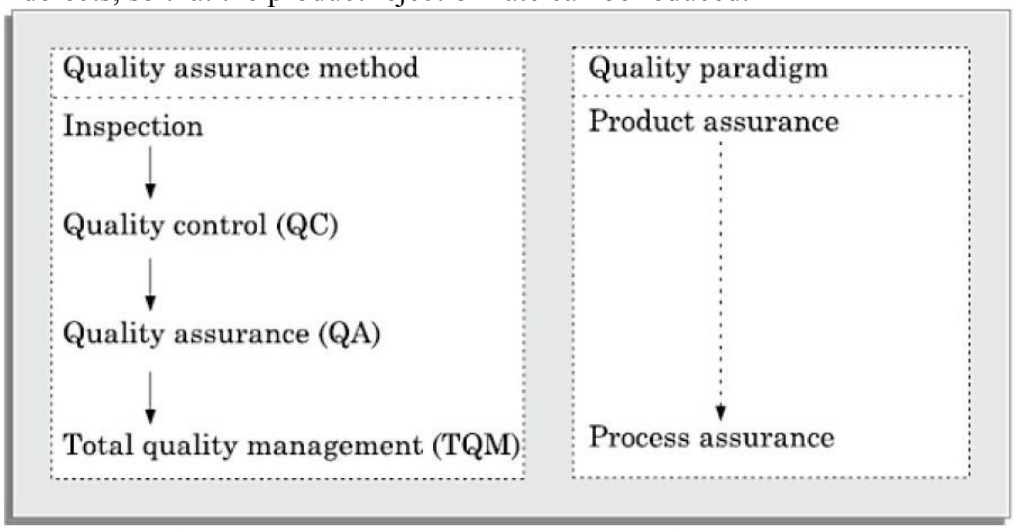

Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.

- 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 basic premise of modern quality assurance is that if an organisation's processes are good then the products are bound to be of good quality.
- The modern quality assurance paradigm includes guidance for recognising, 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 optimising them through redesign.

Product Metrics versus Process Metrics :

- All modern quality systems lay emphasis on collection of certain product and process metrics during product development.
- Let us first understand the basic differences between product and process metrics.
- Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

5. ISO 9000

- International Organisation for Standards (ISO) is a consortium of 63 countries established to formulate and foster standardisation. 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 organisation applies to all its activities related to its products or services.
- The ISO standard addresses both operational aspects (that is, the process) and organisational aspects such as responsibilities, reporting, etc.
- It is important to realise that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product it self.
- ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO 9003.
- The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.

The types of software companies to which the different ISO standards apply are as follows:

- **ISO 9001:** This standard applies to the organisations 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 organisations which do not design products but are only involved in production.
- **ISO 9003:** This standard applies to organisations involved only in installation and testing of products.

ISO 9000 for Software Industry

- ISO 9000 is a generic standard that is applicable to big 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 organisations.
- But in every big industries software plays very important role.
- So ISO 9000 is applicable to Software Industries.

Why Get ISO 9000 Certification?

Some of the benefits that accrue to organisations obtaining ISO certification:

- Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification.
- This is especially true in the international market.
- In fact, many organisations awarding international software development contracts insist that the development organisation have ISO 9000 certification.
- For this reason, it is vital for software organisations involved in software export to obtain ISO 9000 certification.

- ISO 9000 requires a well-documented software production process to be in place.
- A well- documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost effective.
- ISO 9000 certification points out the weak points of an organizations and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and TQM.

How to Get ISO 9000 Certification?

- An organisation intending to obtain ISO 9000 certification applies to a ISO 9000 registrar for registration.
- The ISO 9000 registration process consists of the following stages:
- **Application stage:** Once an organisation decides to go for ISO 9000 certification, it applies to a registrar for registration.
- **Pre-assessment:** During this stage the registrar makes a rough assessment of the organisation.
- **Document review and adequacy audit:** During this stage, the registrar reviews the documents submitted by the organisation and makes suggestions for possible improvements.
- **Compliance audit:** During this stage, the registrar checks whether the suggestions made by it during review have been complied to by the organisation or not.
- **Registration:** The registrar awards the ISO 9000 certificate after successful completion of all previous phases.
- **Continued surveillance:** The registrar continues monitoring the organisation periodically.
- ISO mandates that a certified organisation can use the certificate for corporate advertisements but cannot use the certificate for advertising any of its products. This is probably due to the fact that the ISO 9000 certificate is issued for an organisation's process and not to any specific product of the organisation.

Salient Features of ISO 9001 Requirements

- Salient features all the the requirements as follows:
- **Document control:** All documents concerned with the development of a software product should be properly managed, authorised, and controlled. This requires a configuration management system to be in place.
- **Planning:** Proper plans should be prepared and then progress against these plans should be monitored.
- **Review:** Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.
- **Testing:** The product should be tested against specification.
- **Organisational aspects:** Several organisational aspects should be addressed e.g., management reporting of the quality team.

5.6 ISO 9000-2000

- ISO revised the quality standards in the year 2000 to fine tune the standards.
- The major changes include a mechanism for continuous process improvement.
- There is also an increased emphasis on the role of the top management, including establishing a measurable objectives for various roles and levels of the organisation. The new standard recognises that there can be many processes in an organisation.

6. SEI CAPABILITY MATURITY MODEL

- **" SEI CMM - Software Engineering Institute Capability Maturity Model "** 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" [Crosby79].
- The Unites States Department of Defence (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 U.S. Department of Defense (DoD) in software acquisition.
- 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 organisations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits.
- Gradually many commercial organisations began to adopt CMM as a framework for their own internal improvement initiatives.
- SEI CMM classifies software development industries into the following five maturity levels:

Level 1: Initial

- A software development organisation at this level is characterised by ad hoc activities.

Level 2: Repeatable

- At this level, the basic project management practices such as tracking cost and schedule are established. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis is used.

Level 3: Defined

- At this level, the processes for both management and development activities are defined and documented. There is a common organisation-wide understanding of activities, roles, and responsibilities.

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.

Level 5: Optimising

- At this stage, process and product metrics are collected. Process and product measurement data are analysed for continuous process improvement.
-

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 organisation 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.
- Thus, it provides a way for achieving gradual quality improvement. In contrast, an organisation adopting ISO 9000 either qualifies for it or does not qualify.

Capability Maturity Model Integration (CMMI)

- Capability maturity model integration (CMMI) is the successor of the capability maturity model (CMM).
- The CMM was developed from 1987 until 1997. In 2002, CMMI Version 1.1 was released. Version 1.2 followed in 2006.
- CMMI aimed to improve the usability of maturity models by integrating many different models into one framework.
- After CMMI was first released in 1990, it was adopted and used in many domains. For example, CMMs were developed for disciplines such as systems engineering (SE-CMM), people management (PCMM), software acquisition (SA-CMM), and others.
- Although many organisations found these models to be useful, they also struggled with problems caused by overlap, inconsistencies, and integrating the models.
- In this context, CMMI is generalised to be applicable to many domains.
- For example, the word "software" does not appear in definitions of CMMI. This unification of various types of domains into a single model makes CMMI extremely abstract. The CMMI, like its predecessor, describes five distinct levels of maturity.

7. COMPUTER AIDED SOFTWARE ENGINEERING (CASE)

- CASE tools help to improve software development effort and maintenance effort.
- CASE has emerged as a much talked topic in software industries.
- Software is becoming the costliest component in any computer installation. Even though hardware prices keep dropping like never and falling below even the most optimistic expectations, software prices are becoming costlier due to increased manpower costs.
- CASE tools promise effort and cost reduction in software development and maintenance.
- Therefore, deployment and development of CASE tools have become pet subjects for most software project managers.
- For software engineers, CASE tools promises to reduce the burden of routine jobs, and help develop better quality products more efficiently.

CASE AND ITS SCOPE

- A CASE tool is a generic term used to denote any form of automated support for software engineering,
- In a more restrictive sense a CASE tool can mean any tool used to automate some activity associated with software development.
- Many CASE tools are now available.
- Some of these tools assist in phase-related tasks such as
    - specification,
    - structured analysis,
    - design,
    - coding,
    - testing, etc. and

- others to non-phase activities such as project management and configuration management.

- The primary objectives in using any CASE tool are:
  - To increase productivity.
  - To help produce better quality software at lower cost.

8. CASE ENVIRONMENT

- CASE tools are not integrated, then the data generated by one tool would have to input to the other tools.
- This may also involve format conversions as the tools developed by different vendors are likely to use different formats.
- CASE tools are characterised by the stage or stages of software development life cycle on which they focus.
- The central repository all the CASE tools in a CASE environment share common information among themselves.
- Thus a CASE environment facilitates the automation of the step-by-step methodologies for software development.
- In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development.
- The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger.
- The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements are highlighted.
- Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc.
- A schematic representation of a CASE environment is shown in Figure 12.1.

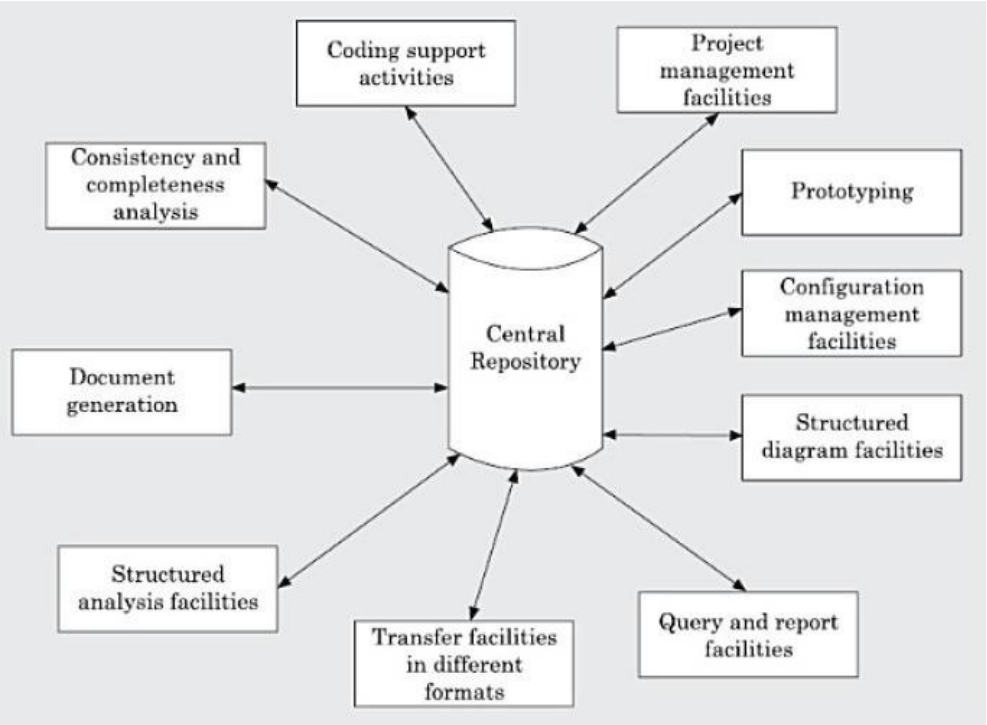

Figure 12.1: A CASE environment.

- The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc.,
- All these tools are integrated. If you click on an error reported by the compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

Benefits of CASE

Let us examine some of these benefits:

- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases.
- Use of CASE tools leads to considerable improvements in quality.
- CASE tools help produce high quality and consistent documents.
- CASE tools reduce the complexity in a software engineers work.
- CASE tools have led to revolutionary cost saving in software maintenance efforts.

9. CASE SUPPORT IN SOFTWARE LIFE CYCLE

- Let us examine the various types of support that CASE provides during the different phases of a software life cycle.
- CASE tools should support a development methodology, help enforce the same, and provide certain amount of consistency checking between different phases.
- Some of the possible support that CASE tools usually provide in the software development life cycle are discussed below.

Prototyping Support

- We have already seen that prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on.
- The prototyping CASE tool's requirements are as follows:
    - Define user interaction.
    - Define the system control flow.
    - Store and retrieve data required by the system.
    - Incorporate some processing logic.
- There are several stand alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype.

A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, a prototyping CASE tool should support the user to create a GUI using a graphics editor.
- The user should be allowed to define all data entry forms, menus and controls. It should integrate with the data dictionary of a CASE environment.
- The user should be able to define the sequence of states through which a created prototype can run.
- The user should also be allowed to control the running of the prototype.

Structured Analysis and Design

- A CASE tool should support one or more of the structured analysis and design technique.
- The CASE tool should support effortlessly drawing analysis and design diagrams.
- The CASE tool should support drawing fairly complex diagrams and preferably through a hierarchy of levels.

- It should provide easy navigation through different levels and through design and analysis.
- The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy.

Code Generation

- More pragmatic support expected from a CASE tool during code generation phase are the following:
    - The CASE tool should support generation of module skeletons or templates in one or more popular languages.
    - It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
    - The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular programming languages.
    - It should generate database tables for relational database management systems.

Test Case Generator

- The CASE tool for test case generation should have the following features:
    - It should support both design and requirement testing
    - It should generate test set reports in ASCII format which can be directly imported into the test plan document.

10. OTHER CHARACTERISTICS OF CASE TOOLS

- The characteristics listed in this section are not central to the functionality of CASE tools but significantly enhance the effectively and usefulness of CASE tools.

Hardware and Environmental Requirements

- In most cases, it is the existing hardware that would place constraints upon the CASE tool selection.
- Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities.
- Therefore, we have to emphasise on selecting the most optimal CASE tool configuration for a given hardware configuration.

Documentation Support

- The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository.
- This helps in producing up-to-date documentation.
- The CASE tool should integrate with one or more of the commercially available desktop publishing packages.
- It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

Pro ject Management

- It should support collecting, storing, and analysing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

External Interface

- The tool should allow exchange of information for reusability of design.
- The information which is to be exported by the tool should be preferably in ASCII format and support open architecture.

Reverse Engineering Support

- The tool should support generation of structure charts and data dictionaries from the existing source codes.
- It should populate the data dictionary from the source code.
- If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

Data Dictionary Interface

- The data dictionary interface should provide view and update access to the entities and relations stored in it.
- It should have print facility to obtain hard copy of the viewed screens.
- It should provide analysis reports like cross-referencing, impact analysis, etc.
- Ideally, it should support a query language to view its contents.

11. TOWARDS SECOND GENERATION CASE TOOL

- An important feature of the second generation CASE tool is the direct support of any adapted methodology.
- This would necessitate the function of a CASE administrator for every organisation, who can tailor the CASE tool to a particular methodology.
- In addition, we may look forward to the following features in the second generation CASE tool:

Intelligent diagramming support:

- The fact that diagramming techniques are useful for system analysis and design is well established. The future CASE tools would provide help to draw the diagrams.

Integration with implementation environment:

- The CASE tools should provide integration between design and implementation.

Data dictionary standards:

- It is highly unlikely that any one vendor will be able to deliver a total solution.
- Moreover, a preferred tool would require tuning up for a particular system.
- Thus the user would act as a system integrator. This is possible only if some standard on data dictionary emerges.

12. ARCHITECTURE OF A CASE ENVIRONMENT

- The architecture of CASE environment is shown in Figure 12.2.
- The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository.

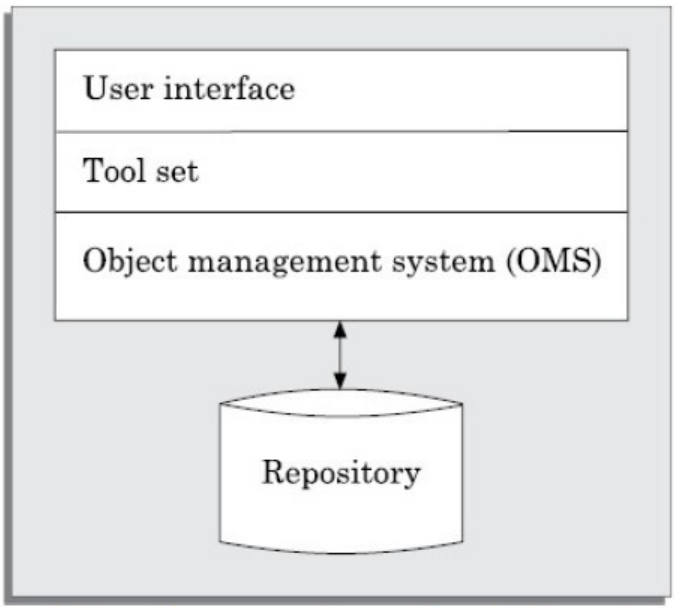

Figure 12.2: Architecture of a modern CASE environment.

User interface

- The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

Object management system and repository

- Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc.
- The object management system maps these logical entities into the underlying storage management system (repository).
- The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records.
- There are a few types of entities but large number of instances.
- By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each.
- Thus the object management system takes care of appropriately mapping these entities into the underlying storage management system.