Unit – 5

Software quality, Reliability and other issues

Software Reliability:

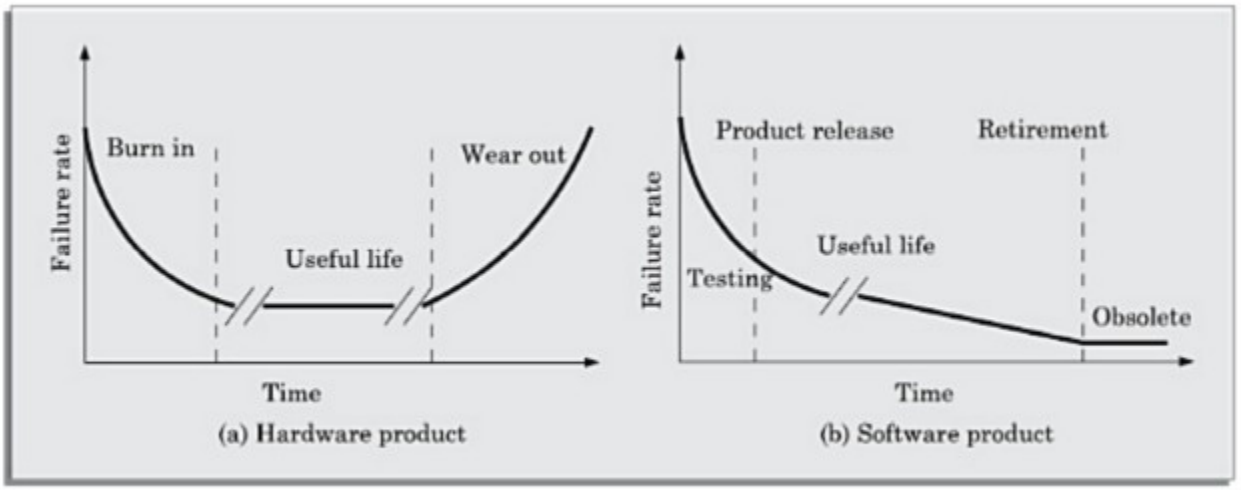
- The reliability of a software product essentially denotes its trustworthiness or dependability.
- Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.
- It is obvious that a software product having 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 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.
    - 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 percent 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 product defects from the least used parts of a system would typically result in only 3 per cent improvement to the product reliability.
- The quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed.

- The quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed.
- Apart from this, 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.
- We can summarize 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.

Hardware Reliability vs Software Reliability:

- An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.
- 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.
- 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.
- Hardware reliability study is concerned with stability
- The aim of software reliability study would be reliability growth.

- A comparison of the changes in failure rate over the product lifetime for a typical hardware product as well as a software product are sketched in the following figure.



Change in failure rate of a product

Reliability Metrics for Software Products:

- The reliability requirements for different categories of software products may be different
- it is necessary that the level of reliability required for a software product should be specified in the software requirements specification (SRS) document.
- We need some metrics to quantitatively express the reliability of a software product.

- A good reliability measure should be observer-independent. We discuss six metrics that correlate with reliability as follows.

1. **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

behavior 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.

2. **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.
    - It is important to note that only run time is considered in the time measurements.
3. **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.
4. **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.
5. **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.
6. **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.

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 are 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.
    - 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 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.

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 straightforward 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.

What is Software Quality Management?

Software Quality Management ensures that the required level of quality is achieved by submitting improvements to the product development process. SQA aims to develop a culture within the team and it is seen as everyone's responsibility.

Software Quality management should be independent of project management to ensure independence of cost and schedule adherences. It directly affects the process quality and indirectly affects the product quality.

Activities of Software Quality Management:

- **Quality Assurance -** QA aims at developing Organizational procedures and standards for quality at Organizational level.
- **Quality Planning -** Select applicable procedures and standards for a particular project and modify as required to develop a quality plan.
- **Quality Control -** Ensure that best practices and standards are followed by the software development team to produce quality products.

ISO 9000

The International organization for Standardization is a world wide federation of national standard bodies. The **International standards organization (ISO)** is a standard which serves as a for contract between independent parties. It specifies guidelines for development of **quality system**.
Quality system of an organization means the various activities related to its products or services. Standard of ISO addresses to both aspects i.e. operational and organizational aspects which includes responsibilities, reporting etc. An ISO 9000

standard contains set of guidelines of production process without considering product itself.

Types of ISO 9000 Quality Standards

ISO 9000 is a series of three standards

The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production, then good quality products are bound to follow automatically. The types of industries to which the various ISO standards apply are as follows.

1. **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

Why ISO Certification required by Software Industry?

There are several reasons why software industry must get an ISO certification. Some of reasons are as follows :

- This certification has become a standards for international bidding.
- It helps in designing high-quality repeatable software products.
- It emphasis need for proper documentation.
- It facilitates development of optimal processes and totally quality measurements.

Features of ISO 9001 Requirements :

- **Document control –** All documents concerned with the development of a software product should be properly managed and controlled.

- **Planning –** Proper plans should be prepared and monitored.
- **Review –** For effectiveness and correctness all important documents across all phases should be independently checked and reviewed .
- **Testing –**The product should be tested against specification.
- **Organizational Aspects –** Various organizational aspects should be addressed e.g., management reporting of the quality team.

SEI Capability Maturity Model

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits

SEI CMM can be used 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. The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work.

Therefore, the results of software process capability assessment can be used to select a contractor. On the other hand, software process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

SEI CMM classifies software development industries into the following five maturity levels. 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.

Level 1: Initial. 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 engineers leave, the successors have great difficulty in understanding the process followed and the work completed. Since formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

Level 2: Repeatable. At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Please remember that opportunity to repeat a process exists only when a company produces a family of products. 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. ISO 9000 aims at achieving this level. Level 4: Managed. At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements 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. Thus, the results of process measurements are used to evaluate project performance rather than improve

the process. Level 5: Optimizing. At this stage, 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 was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best software engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the organization.

Key process areas (KPA) of a software organization Except for SEI CMM level 1, each maturity level is characterized by several Key Process Areas (KPAs) that includes the areas an organization should focus to improve its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig

CMM Level	Focus	Key Process Ares
1. Initial	Competent people	--
2. Repeatable	Project management	Software project planning Software configuration management
3. Defined	Definition of processes	Process definition Training program Peer reviews
4. Managed	Product and process quality	Quantitative process metrics Software quality management
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change management

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. For example, it considers that trying to

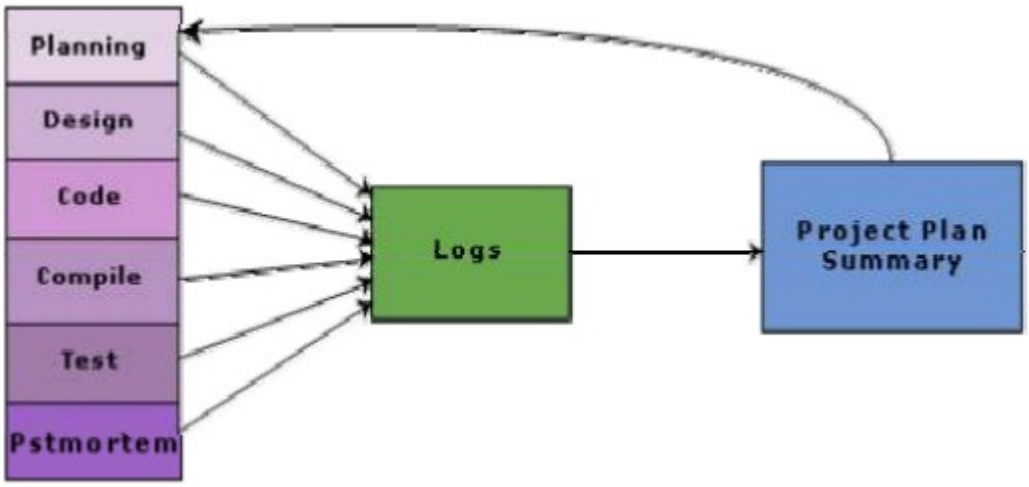
implement a defined process (SEI CMM level 3) before a repeatable process (SEI CMM level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.

Personal software process(PSP)

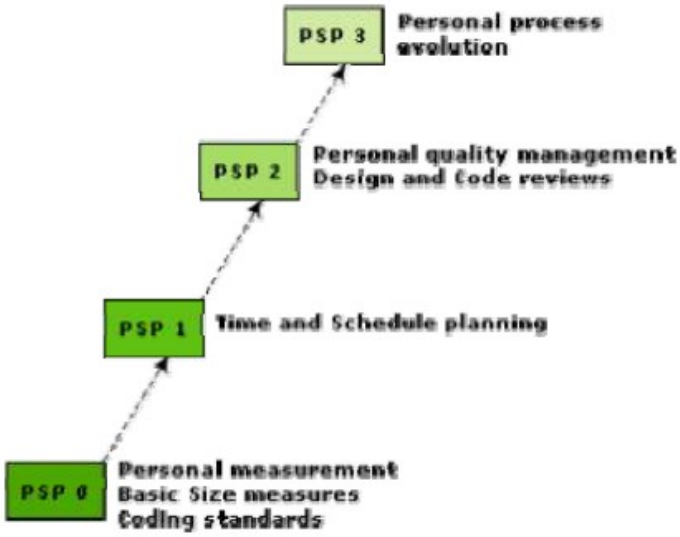
Personal Software Process (PSP) is a scaled down version of the industrial software process. PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyze, design, code, test, or document software products, but assumes that engineers use effective personal practices. PSP recognizes that the process for individual use is different from that necessary for a team. The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating and planning, by showing how to track performance against plans, and provides a defined process which can be tuned by individuals. Time measurement. PSP advocates that engineers should rack the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break etc. An engineer should measure the time he spends for designing, writing code, testing, etc.

PSP Planning. Individuals must plan their project. They must estimate the maximum, minimum, and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in fig. While carrying out the different phases, they must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve their process, etc.



The PSP levels are summarized in fig. PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed from gathering and analyzing defect data earlier projects.



Levels of PSP

Six sigma

The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry. Six Sigma at many organizations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and product to service. The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined as any system behavior that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator. The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies: DMAIC and DMADV. The Six Sigma DMAIC process (define, measure, analyze, improve, control) is an improvement system for existing processes failing below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts. Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include
    - Correctness: Correctness is the degree of which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
    - Maintainability: Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is mean time to change.
    - Integrity: This attribute measures a system's ability to withstand attacks(both accidental and intentional) to its security. To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability that an attack of a specific type will occur within a given time. Security is the probability that the attack of a specific type will be repelled.

      $$\text{Integrity} = \sum [1 - (\text{threat} * (1 - \text{security}))]$$

    - Usability: Usability is an attempt to quantify easy to learn, easy to use, productivity increase, user attitude.
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied throughout the process framework

  $$DRE = E / (E + D)$$
  E = number of errors found before delivery of work product
  D = number of defects found after work product delivery

Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

The following are the set of easily collected metrics:

- Time (hours or days) elapsed from the time a request is made until evaluation is complete.
- Effort (person-hours) to perform the evaluation
- Time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel
- Effort (person-hours) required to make the change Time required (hours-days) to make the change
- Errors uncovered during work to make the change
- Defects uncovered after change is released to the customer base.

Establishing a Software Metrics Program

1. Identify business goal
2. Identify what you want to know
3. Identify subgoals
4. Identify subgoal entities and attributes
5. Formalize measurement goals
6. Identify quantifiable questions and indicators related to subgoals
7. Identify data elements needed to be collected to construct the indicators
8. Define measures to be used and create operational definitions for them
9. Identify actions needed to implement the measures
10. Prepare a plan to implement the measures

CASE tool and its scope :

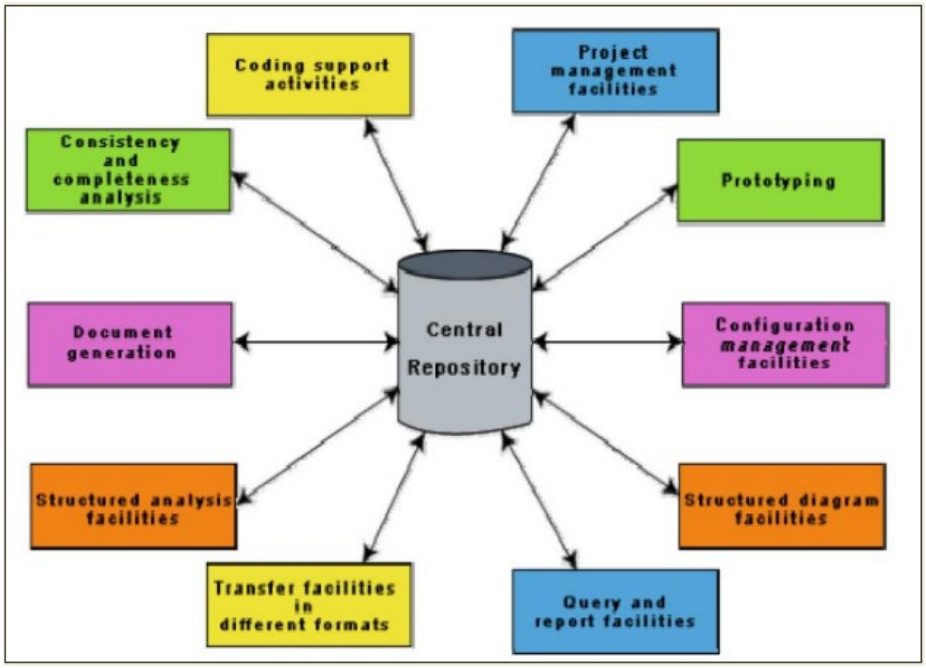
A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE 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. Reasons for using CASE tools

The primary reasons for using a CASE tool are:

- To increase productivity
- To help produce better quality software at lower cost

CASE environment:

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts. This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilities the automation of the step-by-step methodologies for software development. A schematic representation of a CASE environment is shown in fig.



Software Maintenance :

- Software maintenance denotes any changes made to a software product after it has been delivered to the customer.
- Maintenance is inevitable for almost any kind of product.
- Most products need maintenance due to the wear and tear caused by use.
- Software products need maintenance to correct errors, enhance features, port to new platforms, etc.

Need for Maintenance – Software Maintenance must be performed in order to:

- Correct faults.
- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.
- Migrate legacy software.
- Retire software.

Categories of Software Maintenance – Maintenance can be divided into the following:

1. Corrective maintenance: Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.

2. Adaptive maintenance: This includes modifications and updations when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.

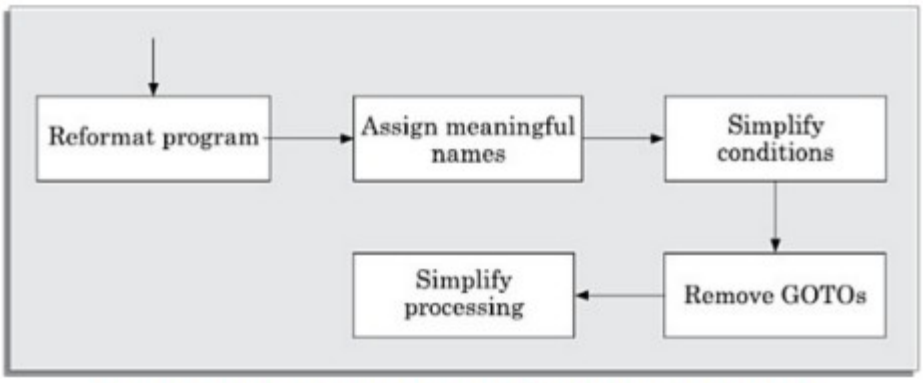
3. Perfective maintenance: A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer demands.

4. Preventive maintenance: This type of maintenance includes modifications and updations to prevent future problems of the software. It goals to attend problems, which are not significant at this moment but may cause serious issues in future

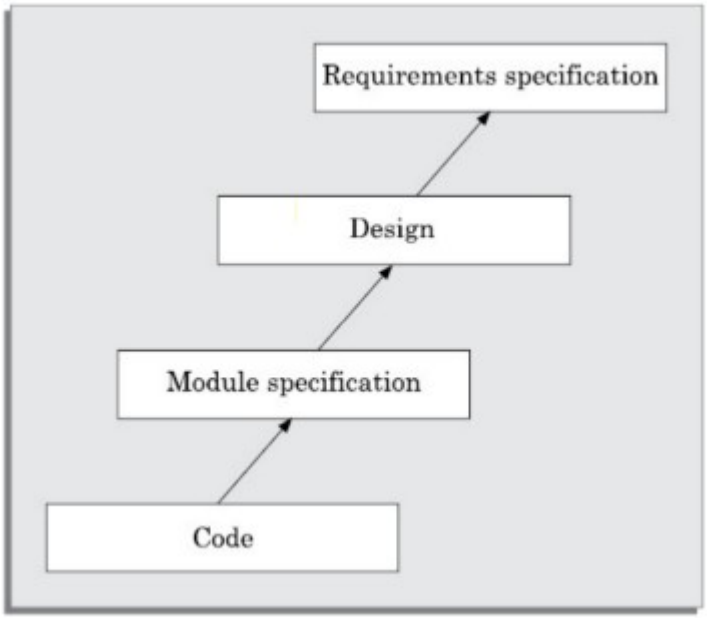
Software Reverse Engineering:

- Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.
- The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

- Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured.
- Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.
- The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities.



Cosmetic changes carried out before reverse engineering



A process model for reverse engineering

- After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin.
- In order to extract the design, a full understanding of the code is needed.
- The structure chart (module invocation sequence and data interchange among modules) should also be extracted.
- The SRS document can be written once the full code has been thoroughly understood and the design extracted.

Software maintenance process models:

- Before discussing process models for software maintenance, we need to analyze various activities involved in a typical software maintenance project.
- The activities involved in a software maintenance project are not unique and depend on several factors such as:

  (i) the extent of modification to the product required,

  (ii) the resources available to the maintenance team,

  (iii) the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.),

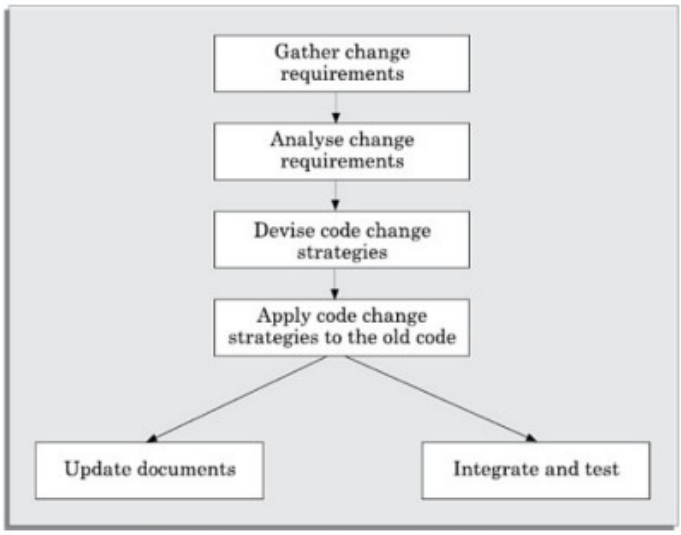
  (iv) the expected project risks, etc.

- When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents.
- However, more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

- No single maintenance process model can be developed. Two broad categories of process models can be proposed.

First Model:

- The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later.
- The project starts by gathering the requirements for changes.
- The requirements are next analyzed to formulate the strategies to be adopted for code change.



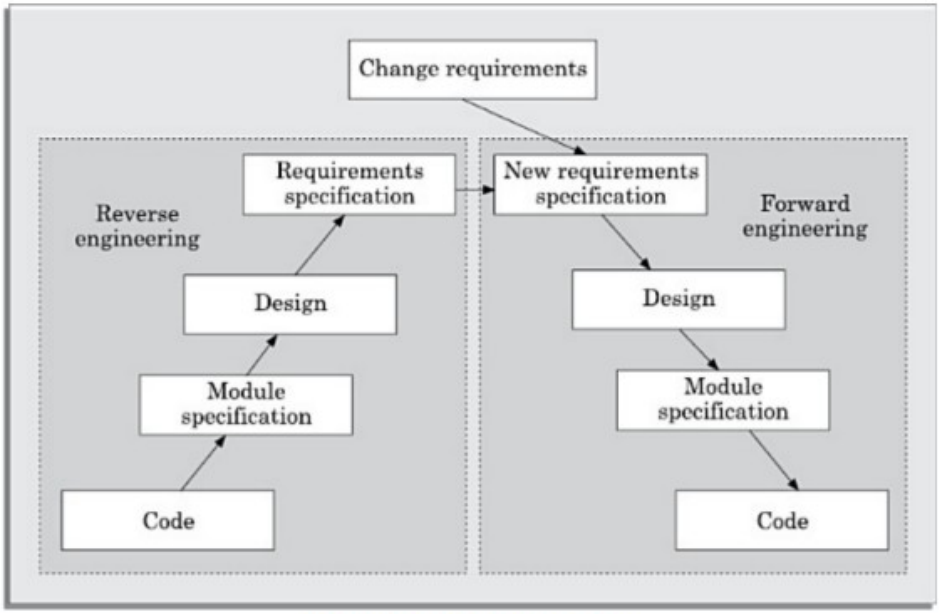
Maintenance Process Model 1

- The association of at least a few members of th original development team goes a long way in reducing the cycle time.
- The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system.

- Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

Second Model:

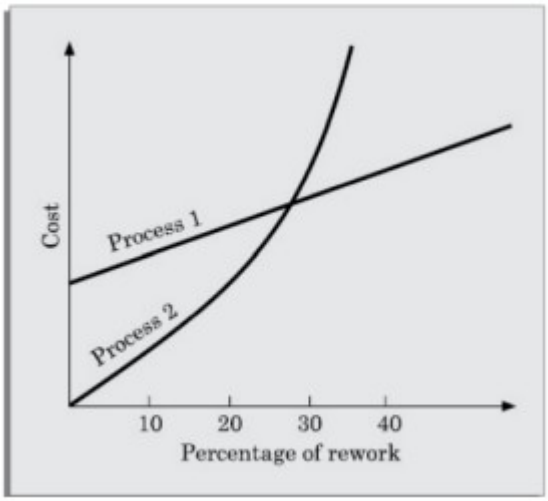
- The second model is preferred for projects where the amount of rework required is significant.
- This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering.



Maintenance Process model 2

- The reverse engineering cycle is required for legacy products.
- During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications.
- The module specifications are then analyzed to produce the design.

- The design is analyzed (abstracted) to produce the original requirements specification.
- The change requests are then applied to this requirements specification to arrive at the new requirements specification.
- At this point a forward engineering is carried out to produce the new code.
- At the design, module specification, and coding a substantial reuse is made from the reverse engineered products.
- An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency.
- This approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15 per cent.



Empirical estimation of maintenance cost versus percentage of rework.

- Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2 as follows:

- Re-engineering might be preferable for products which exhibit a high failure rate.
- Re-engineering might also be preferable for legacy products having poor design and code structure.

Estimation of maintenance cost:

- We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product.
- However, maintenance costs vary widely from one application domain to another.
- For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.
- Boehm proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model.
- Boehm's maintenance cost estimation is made in terms of a quantity called the annual change traffic (ACT).
- Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion

$$\text{ACT} = \frac{\text{KLOC}_{\text{added}} + \text{KLOC}_{\text{deleted}}}{\text{KLOC}_{\text{total}}}$$

where, KLOCadded is the total kilo lines of source code added during maintenance. KLOCdeleted is the total KLOC deleted during maintenance.

- Thus, the code that is changed, should be counted in both the code added and code deleted.
- The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = ACT × Development cost

- Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

Software Reuse :

- Software products are expensive.
- Therefore, software project managers are always worried about the high cost of software development and are desperately looking for ways to cut development costs.
- A possible way to reduce development cost is to reuse parts from previously developed software.
- A reuse approach that is of late gaining prominence is component-based development.
- Component-based software development is different from traditional software development in the sense that software is developed by assembling software from off-the-shelf components.

What can be reused?

- It is important to deliberate about the kinds of the artifacts associated with software development that can be reused.
- Almost all artifacts associated with software development, including project plan and test plan can be reused.
- However, the prominent items that can be effectively reused are:
    - Requirements specification
    - Design
    - Code
    - Test cases
    - Knowledge

- Knowledge is the most abstract development artifact that can be reused.
- Out of all the reuse artifacts, reuse of knowledge occurs automatically without any conscious effort in this direction.
- However, two major difficulties with unplanned reuse of knowledge is that a developer experienced in one type of product might be included in a team developing a different type of software.
- Also, it is difficult to remember the details of the potentially reusable development knowledge.
- A planned reuse of knowledge can increase the effectiveness of reuse.
- For this, the reusable knowledge should be systematically extracted and documented.
- But, it is usually very difficult to extract and document reusable knowledge.

Why almost no reuse so far?

- A common scenario in many software development industries is explained further.
- Engineers working in software development organizations often have a feeling that the current system that they are developing is similar to the last few systems built.
- However, no attention is paid on how not to duplicate what can be reused from previously developed systems.
- Everything is being built from scratch.
- The current system falls behind schedule and no one has time to figure out how the similarity between the current system and the systems developed in the past can be exploited.

- Even those organizations which embark on a reuse program, in spite of the above difficulty, face other problems.
- Creation of components that are reusable in different applications is a difficult problem.
- Even when the reusable components are carefully created and made available for reuse, programmers prefer to create their own, because the available components are difficult to understand and adapt to the new applications.
- The following observation is significant:
    - The routines of mathematical libraries are being reused very successfully by almost every programmer.
    - No one in their mind would think of writing a routine to compute sine or cosine.
    - Let us investigate why reuse of commonly used mathematical functions is so easy.
    - Several interesting aspects emerge.
        - ■ Cosine means the same to all.
        - ■ Everyone has clear ideas about what kind of argument should cosine take, the type of processing to be carried out and the results returned.
        - ■ Secondly, mathematical libraries have a small interface.
        - ■ For example, cosine requires only one parameter.
        - ■ Also, the data formats of the parameters are standardized.
- These are some fundamental issues which would remain valid for all our subsequent discussions on reuse.

Basic issues in any reuse program:

The following are some of the basic issues that must be clearly understood for starting any reuse program:

- Component creation: For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important.
- Component indexing and storing: Indexing requires classification of the reusable components so that they can be easily searched when we look for a component for reuse. The components need to be stored in a relational database management system (RDBMS) or an object-oriented database system (ODBMS) for efficient access when the number of components becomes large.
- Component searching: The programmers need to search for the right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.
- Component understanding: The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.
- Component adaptation: Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.
- Repository maintenance: A component repository once created requires continuous maintenance. New components, as and when created have to be entered into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository