UNIT 5

Contents:

Software Reliability and Quality Management: Software Reliability, Statistical Testing, Software Quality, Software Quality Management System.

Software Maintenance: Software maintenance, Maintenance Process Models, Maintenance Cost.

Software Reuse: what can be reused? Why Almost No Reuse So Far? Basic Issues in Reuse Approach.

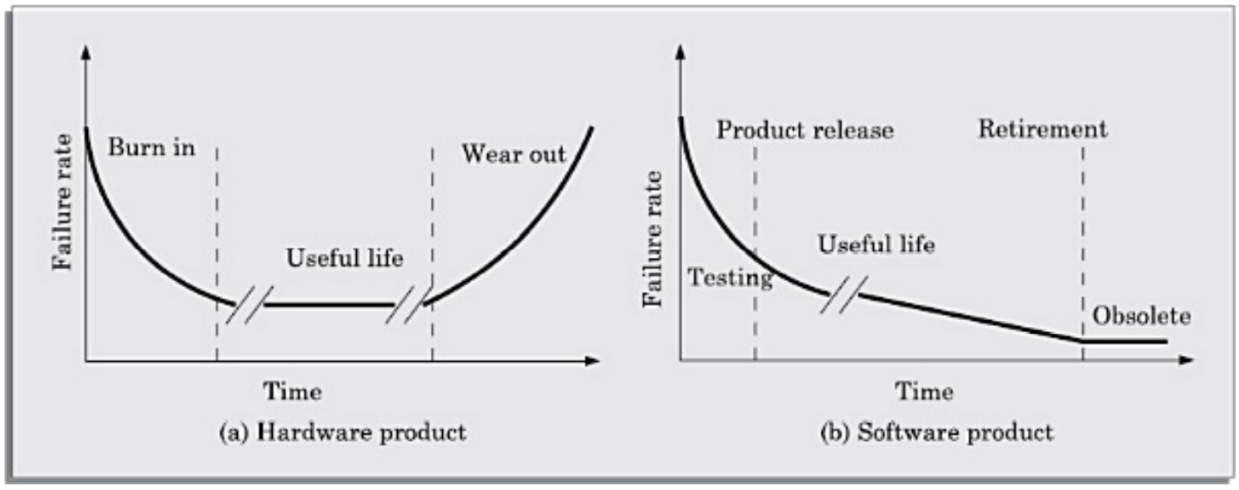
Software Reliability:

- 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.

Shortcomings of reliability metrics of software products:

- All the above reliability metrics suffer from several shortcomings.
- 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.

- These reliability models do not distinguish the relative severity of different failures.
- 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.
- 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.

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 $\{f_i\}$, the operational profile would associate each function $\{f_i\}$ with the probability with which an average user would select $\{f_i\}$ as his next function to use.
    - Thus, we can think of the operation profile as assigning a probability value p_i to each functionality f_i 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.

Software Quality

- Traditionally, the quality of a product is defined in terms of its fitness of purpose.
- 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.
- "Fitness of purpose" is not a wholly satisfactory definition of quality for software products.
  - 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.
  - The higher level attributes, known as quality factors or external attributes can only be measured indirectly.
  - The second-level quality attributes are called quality criteria.

- 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.
- The reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time.

Software Quality Management System:

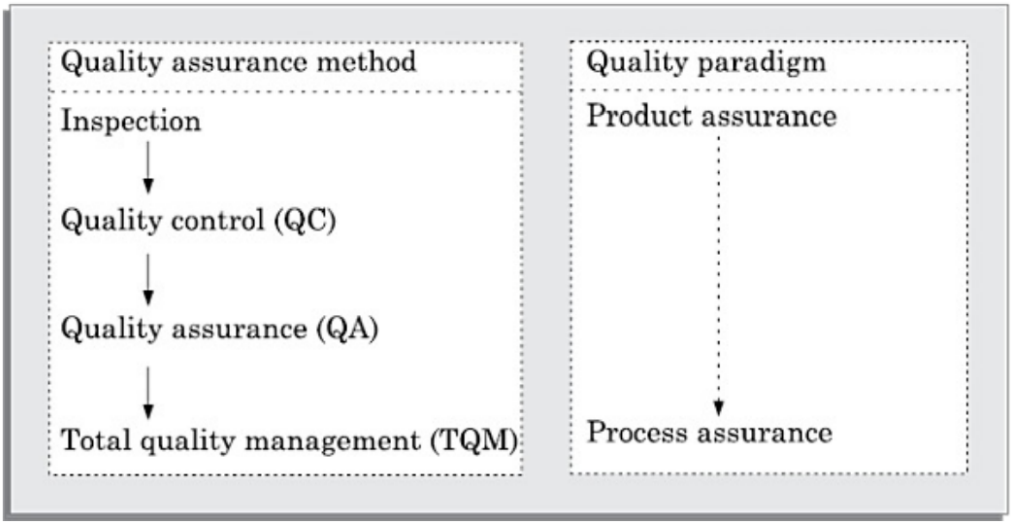
- A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality.
- A quality system is the responsibility of the organization as a whole.
- However, every organization has a separate 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 analyze 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 organization.
- 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.
- ISO 9000 provides guidance on how to organize 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 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 a certain specified tolerance range.
- Since that time, quality systems of organizations have undergone four stages of evolution as shown in figure.



- 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.
- 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 organization's processes are good and are followed rigorously, then the products are bound to be of good quality.
- The modern quality assurance paradigm includes guidance for recognising, defining, analyzing, 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 optimize them through redesign.

Product metrics vs Process metrics:

- All modern quality systems lay emphasis on collection of certain product and process metrics during product development.
- *Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.*
- Examples of product metrics are LOC and function point to measure size, PM (person-month) to measure the effort required to develop it, months to measure the time required to develop the product, time complexity of the algorithms, etc.
- 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, number of latent defects per line of code in the developed product.

Self Study:

- ☐ ***ISO 9000***

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.

Types of Software Maintenance:

- Software maintenance is becoming an important activity of a large number of organizations.
- Given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.
- When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary.
- Whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface.
- Every software product continues to evolve after its development through maintenance efforts
- There are three types of software maintenance, which are described as follows:
  - ***Corrective***: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.
  - ***Adaptive***: A software product might need maintenance 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 or software.
  - ***Perfective***: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Characteristics of Software evolution:

Lehman and Belady have studied the characteristics of evolution of several software products. They have expressed their observations in the form of laws. These are generalizations and may not be applicable to specific cases.

Lehman's first law:

- A software product must change continually or become progressively less useful.
- Every software product continues to evolve after its development through maintenance efforts.
- This law clearly shows that every product irrespective of how well designed must undergo maintenance.

- In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded.
- Good products are maintained and bad products are thrown away.

Lehman's second law:

- The structure of a program tends to degrade as more and more maintenance is carried out on it.
- When you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support.
- If you do not redesign the system, the additions will be more complex than they should be.
- Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law:

- Over a program's lifetime, its rate of development is approximately constant.
- The rate of development can be quantified in terms of the lines of code written or modified.
- Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

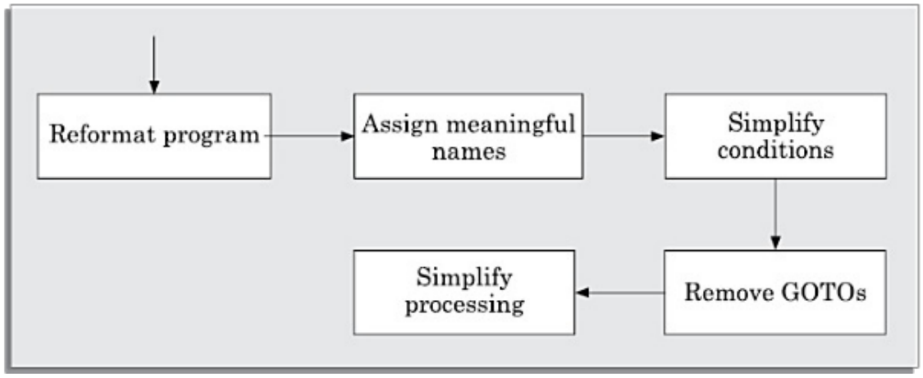
Special problems associated with software maintenance:

- Software maintenance work currently is typically much more expensive than what it should be and takes more time than required.
- The reasons for this situation are the following:
    - *Software maintenance work in organizations is mostly carried out using ad hoc techniques*. Still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.
    - *Software maintenance has a very poor image in industry*. During maintenance it is necessary to thoroughly understand someone else's work, and then carry out the required modifications and extensions.
    - Another problem associated with maintenance work is that *the majority of software products needing maintenance are legacy products*. Though the word legacy implies "aged" software, but There is no agreement on what exactly is a legacy system. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product.

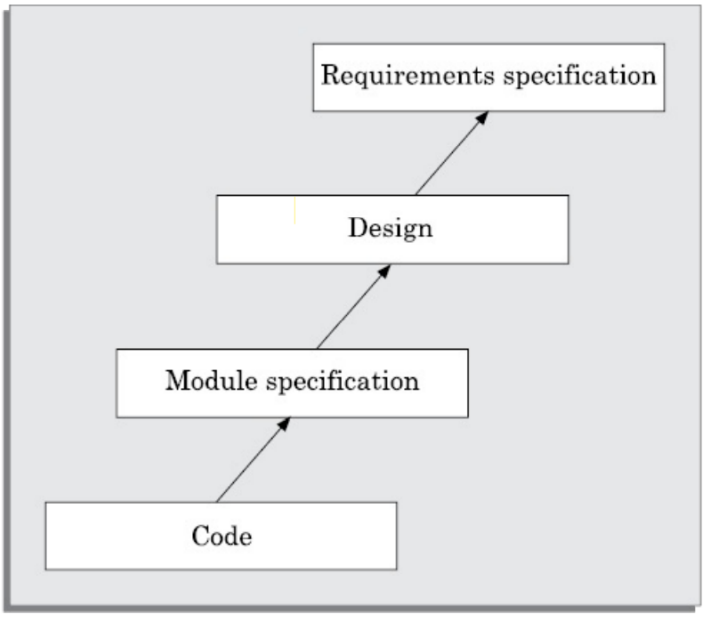
Software Reverse Engineering:

- 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.

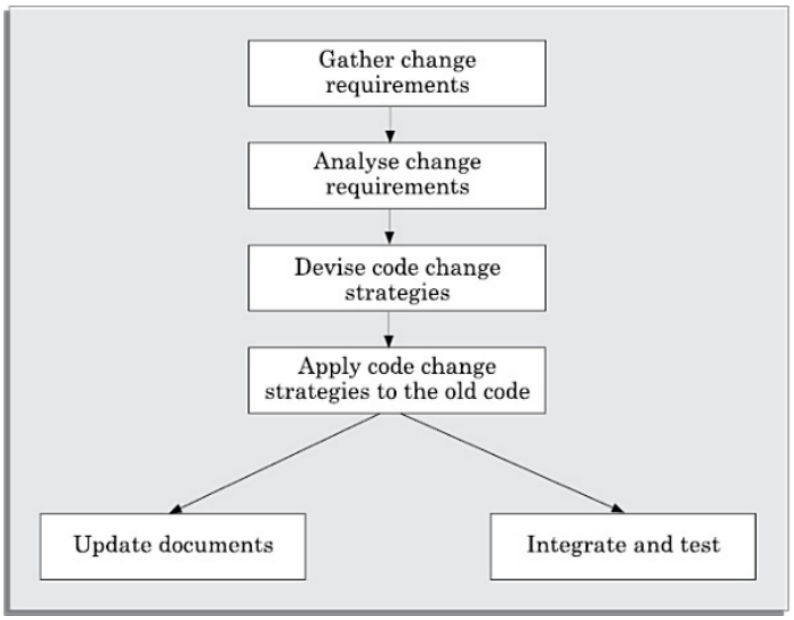
- After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin.
- In order to extract the design, a full understanding of the code is needed.
- The structure chart (module invocation sequence and data interchange 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.

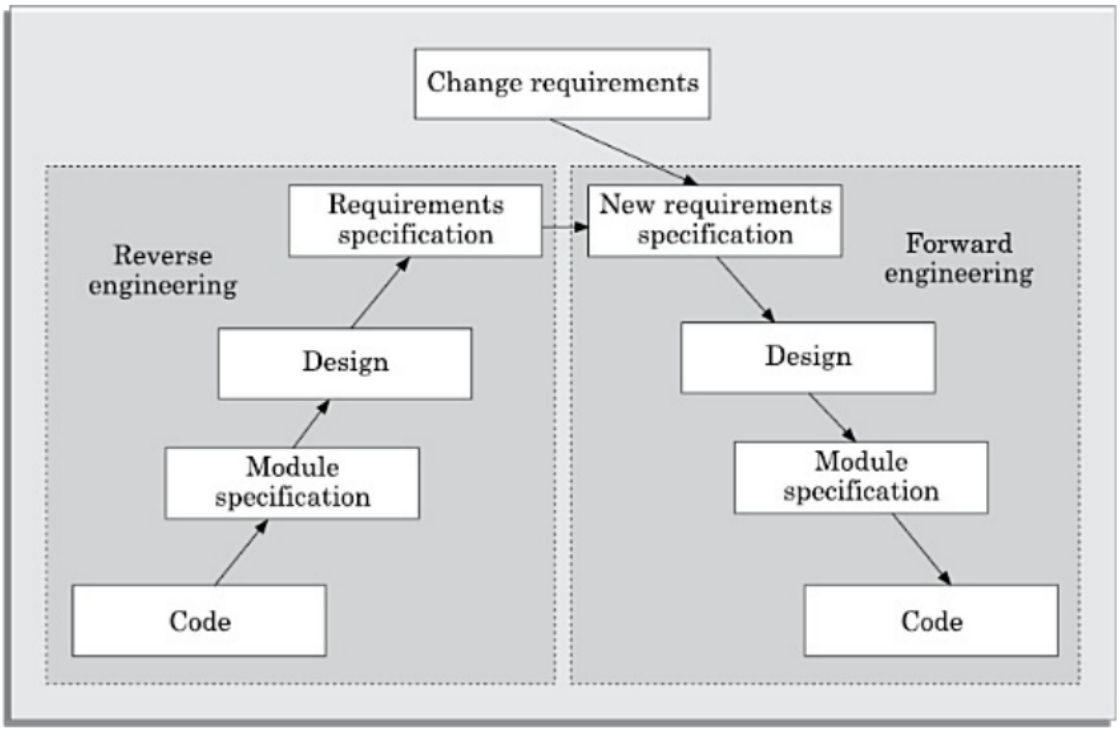


Maintenance Process Model 1

- The association of at least a few members of the 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:

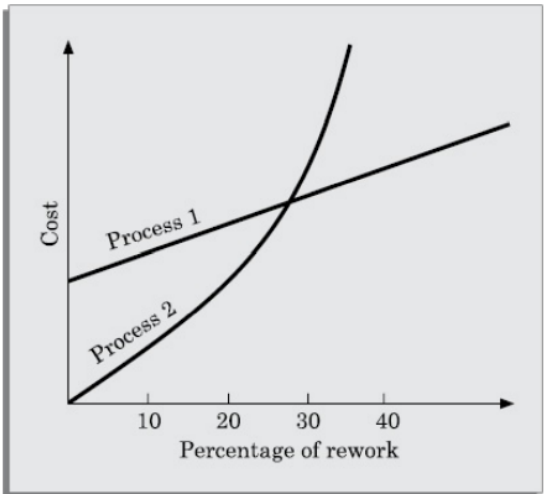
- The second model is preferred for projects where the amount of rework required is significant.
- This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as ***software re-engineering***.



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:

$$\textbf{Maintenance cost} = \textbf{ACT} \times \textbf{Development cost}$$

- 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.

Self study:

- ☐ ***A reuse approach***
  - ☐ ***Domain analysis***
  - ☐ ***Component Classification***
  - ☐ ***Searching***