NOTES
SUBJECT: SOFTWARE ENGINEERING
SUBJECT CODE: KCA-013
BRANCH: MCA
SEM:3rd
SESSION: 2021-22

Mr. Shubham Patkar

MCA Department

INDEX

Unit-1

Unit-2

Unit-3

Software Measurement and Metrics
Halestead's Software Science
Function Point (FP)

Unit-4

Software Testing and Maintenance

Software Testing
Levels of Testing:
Unit Testing
Integration Testing
Acceptance Testing
Top-Down and Bottom-Up Testing
Functional Testing (Black Box Testing)
Structural Testing (White Box Testing)
Mutation Testing
Performance testing
Coding
Coding Standards
Coding Guidelines

Unit-5

Software Maintenance and project management

Software Maintenance
Software RE-Engineering
Software Reverse engineering
Software Configuration management (CM)
Functions of SCM
SCM Terminology
SCM Activities

CASE (Computer aided software engineering)
Constructive cost model (COCOMO)
Putnam Resource Model
Software Risk

Unit-1: Introduction of Software enginnering

Process and Project

A process is a sequence of steps performed for a given purpose. As mentioned earlier, while developing (industrial strength) software, the purpose is to develop software to satisfy the needs of some users or clients.

A software project is one instance of this problem, and the development process is what is used to achieve this purpose.

Program

Program is set of sequential logical instructions.

Software

Software is a set of items or objects that form a "configuration" that includes

- Programs
- Documents
- Data...

Software consists of

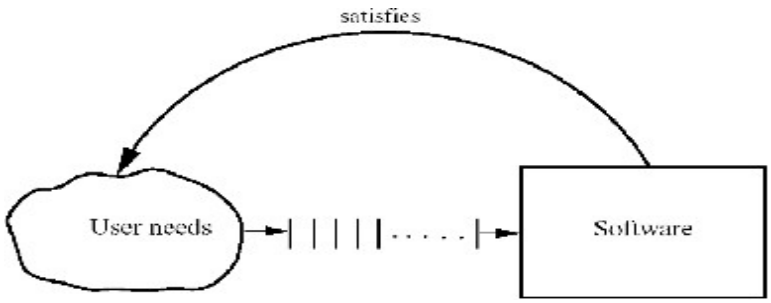
(1) Instructions (computer programs) that when executed provided desired function and performance,

(2) Data structures that enable the programs to adequately manipulate information, and

(3) Documents that describe the operation and use of the programs.

But these are only the concrete part of software that may be seen, there exists also invisible part which is more important:

- Software is the dynamic behavior of programs on real computers and auxiliary equipment.
- "… a software product is a model of the real world, and the real world is constantly changing."
- Software is a digital form of knowledge. ("Software Engineering," 6ed. Sommerville, Addison-Wesley, 2000)
- Computer programs and associated documentation such as requirements, design models and user manuals.
- Software products may be developed for a particular customer or may be developed for a general market.



Software products may be

- Generic - developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
- Bespoke (custom) - developed for a single customer according to their specification.

New software can be created by developing new programs, configuring generic software systems or reusing existing software.

Chronological evolution of software

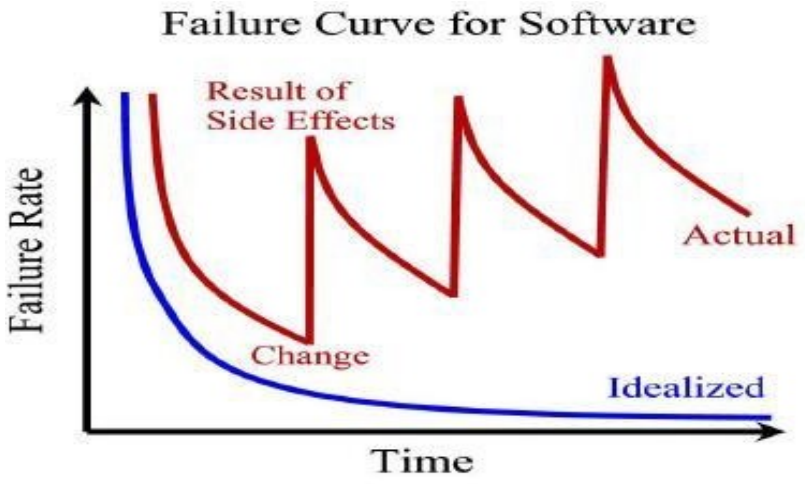
1. 1950-1965 –Early Years
   General Purpose Hardware
   Batch Orientation
   Limited Distribution
   Custom Software
2. 1968-1973- Second Era
   Multi User
   Real Time
   Data Base
   Product Software
   Concept of sw Maintenance
3. 1974-1986-Third Era
   Distributed Systems
   Embedded Systems
   Low cost HW
   Consumer Impact
4. 1987-2000-Fourth Era
   Powerful Desktop
   Object Orient Technology
   Expert Systems
   Artificial Neural Network
   Parallel Computing
   Network Computers

Software Component

- **Lowest Level-** It mirrors the instruction set of the h/w. makes efficient use of memory and optimize programmed execution speed.
- **Machine language-** 1 Level
- **Assembly language**-2 Level
- **Mid Level - 3** Level. Machine independents used to create procedural description of the program. More efficient then machine language C, C++
- **Highest fourth level:-**These are non procedural moves the developer further from the h/w. it use geographical icons.

Ssoftware Characteristics

1) Software does not wear out.



2) Software is not manufactured, but it is developed through the life cycle concept.
3) Reusability of components
4) Software is flexible and can be amended to meet new requirements
5) Cost of its change at if not carried out initially is very high at later stage .

Difference in Program and Software

Program	Software
1. Usually small in size	1.Large
2. Author himself is sole user	2.Large number of users
3. Single developer	3.Team of developers
4. Lacks proper user interface	4.Well-designed interface
5. Lacks proper documentation	5.Well documented & user-manual prepared
6. Ad hoc development.	6.Systematic development

Software Crisis

Software fails because it

- crash frequently
- expensive
- difficult to alter, debug, enhance
- often delivered late
- use resources non-optimally
- Software professionals lack engineering training

- Programmers have skills for programming but without the engineering mindset about a process discipline

Standish Group Report OG States:

1. About US$250 billions spent per year in the US on application development
2. Out of this, about US$140 billions wasted due to the projects getting abandoned or reworked; this in turn because of not following best practices and standards
3. 10% of client/server apps are abandoned or restarted from scratch
4. 20% of apps are significantly altered to avoid disaster
5. 40% of apps are delivered significantly late
6. 30% are only successful

Software engineering Process

Concept of the Software Engineering has evolved in 1986 by BOHEM to reduce the effect of his software crisis because software engineering defined as scientific and systematic approach to develop , operate and maintain software project to meet a given specific object it beautifully envisages the concept of life cycle of any software project through following five phase:-

- Requirement analysis
- Design
- Coding
- Testing
- Maintenance

Software Engineering

Software engineering is concerned with the theories, methods and tools for developing, managing and evolving software products.

- "The systematic application of tools and techniques in the development of computer-based applications." (Sue Conger in The New Software Engineering)
- "Software Engineering is about designing and developing high-quality software." (Shari Lawrence Pfleeger in Software Engineering -- The Production of Quality Software)
- A systematic approach to the analysis, design, implementation and maintenance of software. (The Free On-Line Dictionary of Computing)The systematic application of tools and techniques in the development of computer-based applications. (Sue Conger in The New Software Engineering) Software Engineering is about designing and developing high-quality software. (Shari Lawrence Pfleeger in Software Engineering -- The Production of Quality Software)
- The technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost constraints (R. Fairley)
- A discipline that deals with the building of software systems which are so large that they are built by a team or teams of engineers (Ghezzi, Jazayeri, Mandrioli)

Difference between software engineering and software programming

Software Engineering	Software Programming
1. Single developer	1.Teams of developers with multiple roles
1. "Toy" applications	2.Complex systems
2. Short lifespan	3.Indefinite lifespan
3. Single or few stakeholders Architect = Developer = Manager = Tester = Customer = User	4.Numerous stakeholders Architect ≠ Developer ≠ Manager ≠ Tester ≠ Customer ≠ User
5.One-of-a-kind systems	5.System families

Difference between software engineering and computer science

- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
- Computer science theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering).

Difference between software engineering and system engineering

- System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.

System engineers are involved in system specification, architectural design, integration and deployment

- Software engineering is based on computer science, information science and discrete mathematics whereas traditional engineering is based on mathematics, science and empirical knowledge.
- Traditional engineers construct real artifacts and software engineers construct non-real (abstract) artifacts.
- In traditional engineering, two main concerns for a product are cost of production and reliability measured by time to failure whereas in software engineering two main concerns are cost of development and reliability measured by the no. of errors per thousand lines of source code.

Difference between software engineering and traditional software engineering

- Software engineers often apply new and untested elements in software projects while traditional software engineers generally try to apply known and tested principles and limit the use of untested innovations to only those necessary to create a product that meets its requirements.
- Software engineers emphasize projects that will live for years or decades whereas some traditional engineers solve long-ranged problems that ensure for centuries.

- Software engineering is about 50 years old whereas traditional engineering as a whole is thousands of years old.
- Software engineering is often busy with researching the unknown (eg. To drive an algorithm) right in the middle of a project whereas traditional engineering normally separates these activities. A project is supposed to apply research results in known or new clever ways to build the desired result.
- Software engineering has first recently started to codify and teach best practice in the form of design pattern whereas in traditional, some engineering discipline have thousands of years of best practice experience handed over from generation to generation via a field's literature , standards, rules and regulations.

Software Development Life Cycle (SDLC)Models

Software-development life-cycle is used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way.An information system goes through a series of phases from conception to implementation. This process is called the Software-Development Life-Cycle.

Various reasons for using a life-cycle model include:

- Helps to understand the entire process
- Enforces a structured approach to development
- Enables planning of resources in advance
- Enables subsequent controls of them
- Aids management to track progress of the system

Activities undertaken during feasibility study: - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

Activities undertaken during requirements analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely Requirements gathering and analysis, this phase ends with the preparation of Software requirement Specification (SRS)

Activities undertaken during design: - The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language . Design specification Document is outcome of this phase.

Activities undertaken during coding and unit testing:-The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. Code Listings are generated after this phase,.

Activities undertaken during integration and system testing: - Integration of different modules is undertaken once they have been coded and unit tested During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document.Test Rports are generatd after this phase.

Activities undertaken during maintenance: - Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. This phase continues till the software is in use.

Different software life cycle models

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

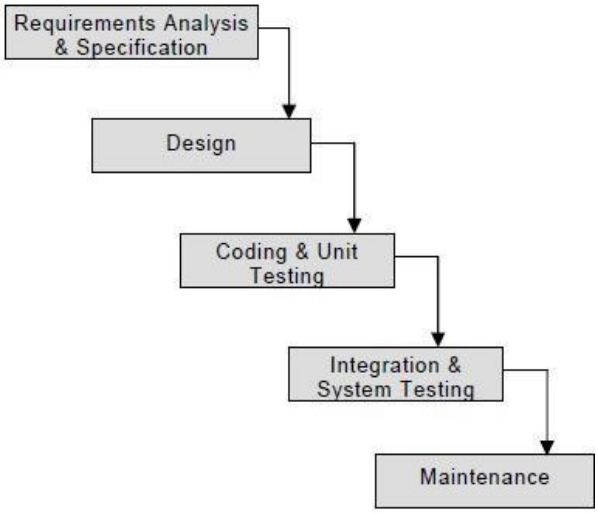
1. Classical Waterfall Model
2. Iterative Waterfall Model
3. Prototyping Model
4. Evolutionary Model
5. Spiral Model
6. Rapid Development Application model (RAD)

Classical Waterfall Model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense t hat it cannot be used in actual software development projects. Thus, this model can be considered to be a theoretical way of developing software. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases as shown in fig:

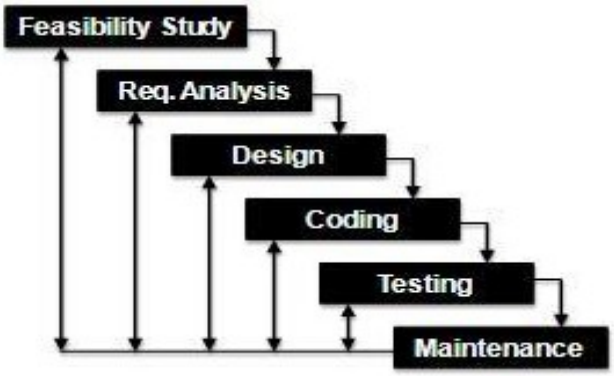
1. Feasibility Study
2. Requirements Analysis and Specification
3. Design
4. Coding and Unit Testing
5. Integration and System Testing
6. Maintenance



Classical Waterfall Model

Iterative Waterfall Model

- Waterfall model assumes in its design that no error will occur during the design phase
- Iterative waterfall model introduces feedback paths to the previous phases for each process phase
- It is still preferred to detect the errors in the same phase they occur
- Conduct reviews after each milestone



Iterative Waterfall Model

Advantages of Waterfall Model

- It is a linear model.
- It is a segmental model.
- It is systematic and sequential.
- It is a simple one.
- It has proper documentation

Disadvantages of Waterfall Model

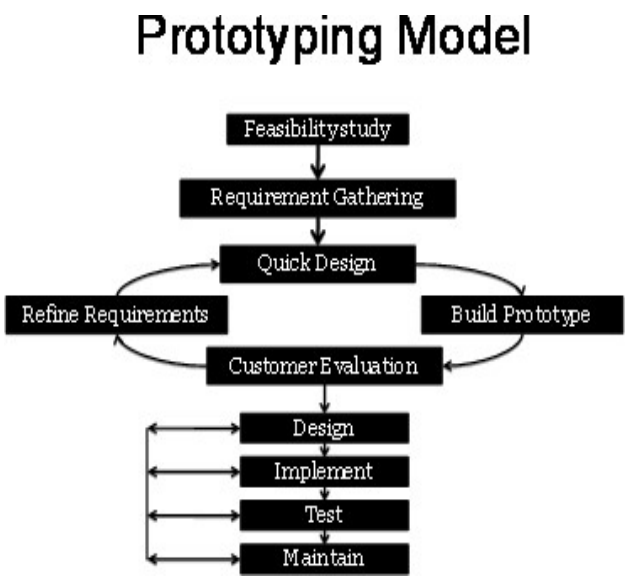
- It is difficult to define all requirements at the beginning of the project.
- Model is not suitable for accommodating any change
- It does not scale up well to large projects
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

Prototype model

The Prototyping Model is a systems development method (SDM) in which a prototype is built, tested, and then reworked as necessary until an acceptable prototype is finally achieved from which the complete system or product can now be developed prototyping paradigm begins with requirements gathering.

- Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- A "quick design" then occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g., input approaches and output formats).

There are several steps in the Prototyping Model as shown in the diagram:-



1. The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the departments or aspects of the existing system.
2. A preliminary design is created for the new system.

3. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system, and represents an approximation of the characteristics of the final product.
4. The users thoroughly evaluate the first prototype, noting its strengths and weaknesses, what needs to be added, and what should to be removed. The developer collects and analyzes the remarks from the users.
5. The first prototype is modified, based on the comments supplied by the users, and a second prototype of the new system is constructed.
6. The second prototype is evaluated in the same manner as was the first prototype.
7. The preceding steps are iterated as many times as necessary, until the users are satisfied that the prototype represents the final product desired.
8. The final system is constructed, based on the final prototype.
9. The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

Customers could believe it's the working system. Developer could take "shortcuts" and only fill the prototype instead of redesigning it. Customers may think that the system is almost done, and only few fixes are needed

Advantage of Prototype model

- Suitable for large systems for which there is no manual process to define there requirements.
- User training to use the system.
- User services determination.
- System training.
- Quality of software is good.
- Requirements are not freezed.

Disadvantage of Prototype model

- It is difficult to find all the requirements of the software initially.
- It is very difficult to predict how the system will work after development.

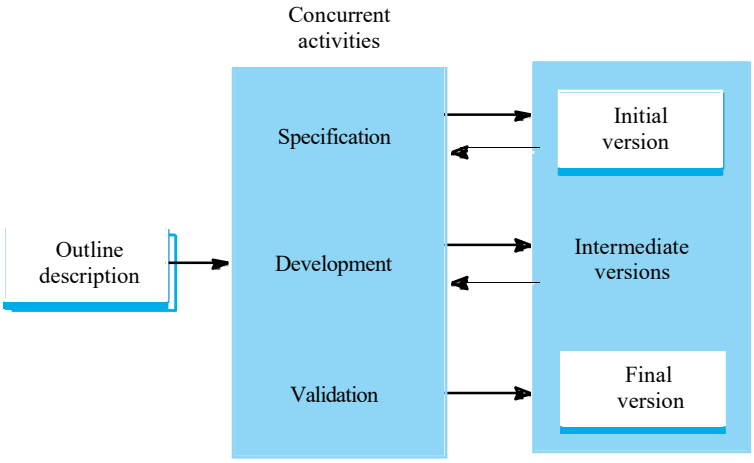
Evolutionary Process Model

In EP model development engineering effort is made first to establish correct, precise requirement definitions and system scope, as agreed by all the users across the organization. This is achieved through application of iterative processes to evolve a system most suited to the given circumstances.

The process is iterative as the software engineer goes through a repetitive process of requirement until all users and stakeholders are satisfied. This model differs from the iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

Main characteristics:

  - The phases of the software construction are interleaved
  - Feedback from the user is used throughout the entire process
  - The software product is refined through many versions
- Types of evolutionary development:
  - Exploratory development
  - Throw-away prototyping



Advantages:

- Deals constantly with changes
  Provides quickly an initial version of the system
- Involves all development teams

- Quick fixes may be involved

Disadvantages: "Invisible" process, not well-supported by documentation

- The system's structure can be corrupted by continuous change

-

-

Spiral model

The Spiral model of software development is shown in fig. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For ex ample, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 2.

The following activities are carried out during each phase of a spiral model.

First quadrant (Objective Setting)

- During the first quadrant, it is n eeded to identify the objectives of the phase.
- Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)

- A detailed analysis is carried out for each identified project risk.
- Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

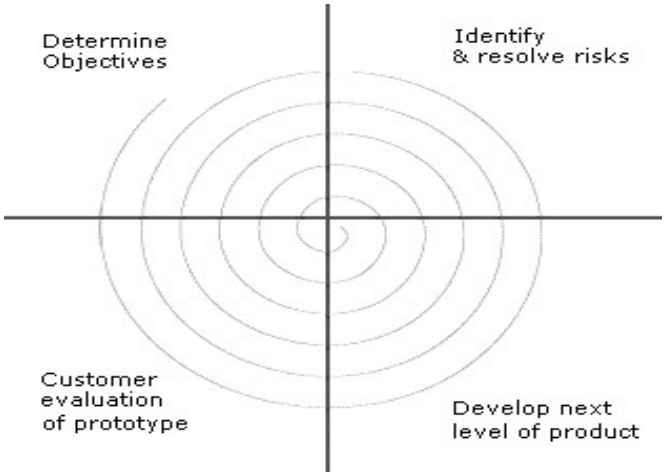
Third Quadrant (Development and Validation)

- Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)

- Review the results achieved so far with the customer and plan the next iteration around the spiral.

- Progressively more complete version of the software gets built with each iteration around the spiral



Advantages of Spiral Model

- It is risk-driven model.
- It is very flexible.
- Less documentation is needed.
- It uses prototyping

Disadvantages of Spiral Model

- No strict standards for software development.
- No particular beginning or end of a particular phase.

Difference between spiral and waterfall model

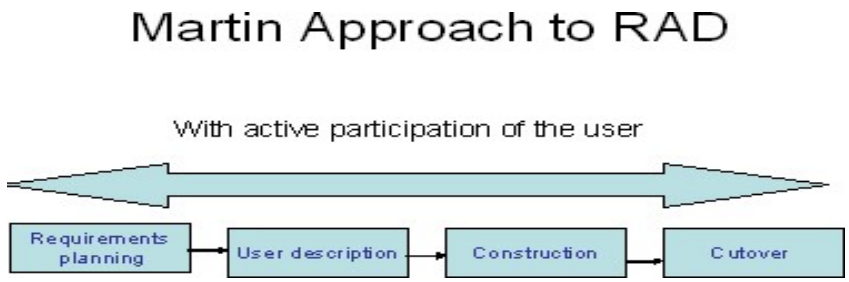
Waterfall model	Spiral model
Separate and distinct phases of specification and development.	Process is represented as a spiral rather than as a sequence of activities with backtracking
After every cycle a useable product is given to the customer.	Each loop in the spiral represents a phase in the process
Effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product.	No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required
Risks are never explicitly assessed and resolved throughout the process	Risks are explicitly assessed and resolved throughout the process

Rapid Development Application model (RAD)

RAD is proposed when requirements and solutions can be modularized as independent system or software components, each of which can be developed by different teams. User involvement is essential from requirement phase to deliver of the product. The process is stared with rapid prototype and is given to user for evolution. In that model user feedback is obtained and prototype is refined. SRS and design document are prepared with the association of users. RAD becomes faster if the software engineer uses the component's technology (CASE Tools) such that the components are really available for reuse. Since the development is distributed into component-development teams, the teams work in tandem and total development is completed in a short period (i.e., 60 to 90 days).

RAD Phases

- **Requirements planning phase** (a workshop utilizing structured discussion of business problems)
- **User description phase** – automated tools capture information from users
- **Construction phase** – productivity tools, such as code generators, screen generators, etc. inside a time-box. ("Do until done")
- **Cutover phase** -- installation of the system, user acceptance testing and user training



Advantage of RAD

- Dramatic time savings the systems development effort
- Can save time, money and human effort
- Tighter fit between user requirements and system specifications
- Works especially well where speed of development is important

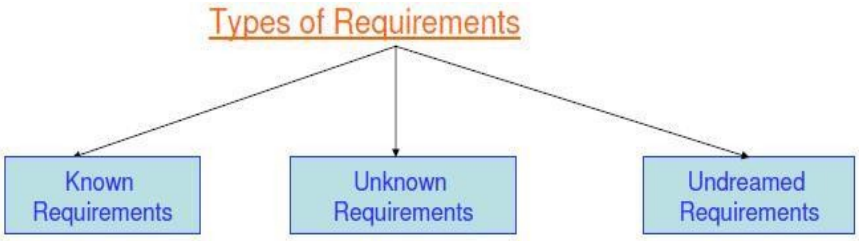
Disadvantage of RAD

- More speed and lower cost may lead to lower overall system quality
- Danger of misalignment of system developed via RAD with the business due to missing information
- May have inconsistent internal designs within and across systems
- Possible violation of programming standards related to inconsistent naming conventions and inconsistent documentation

Unit-2: Software Requirement Specifications

Software Requirement

1. A condition of capability needed by a user to solve a problem or achieve an objective
2. A condition or a capability that must be met or possessed by a system ... to satisfy a contract, standard, specification, or other formally imposed document."



1. **Known Requirements:-** Something a stakeholder believes to be implemented
2. **Unknown requirements:-**Forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder
3. **Undreamt requirements:-** stakeholder may not be able to think of new requirement due to limited knowledge

A Known, Unknown and Undreamt requirements may functional or non functional.

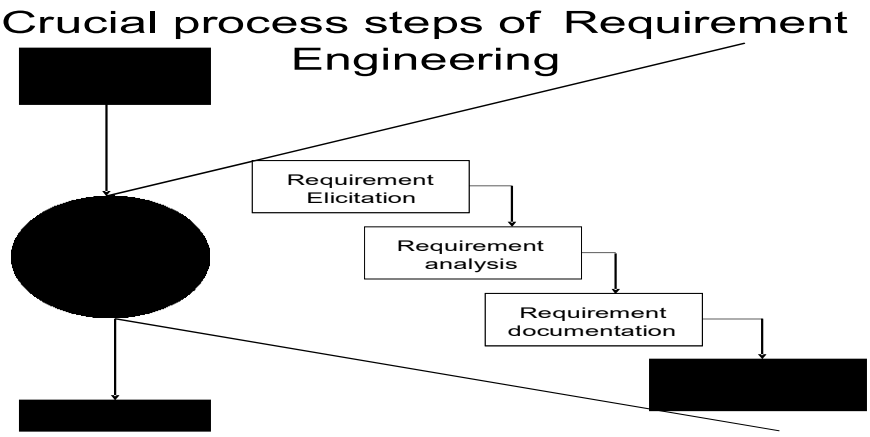
- **Functional requirements: -** describe what the software has to do. They are often called product features. It depends on the type of software, expected users and the type of system where the software is used.
- **Non Functional requirements**: - are mostly quality requirements. That stipulate how well the software does, what it has to do.These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- **User requirements:-** Statements in natural language plus diagrams of the services the system provides and its operational constraints.
- **System requirements: -** A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Requirements engineering Process

- The process of finding out, analyzing, documenting and checking these services and constraints called requirement engineering.
- RE produces one large document, written in a natural language, contains a description of what the system will do without how it will do
- Input to RE is problem statement prepared by the customer and the output is SRS prepared by the developer.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Requirements engineering processes:- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements. However, there are a number of generic activities common to all processes

- Requirements elicitation
- Requirements analysis
- Requirements documentation
- Requirements review



- **Requirement Elicitation: -** known as gathering of requirement. Here requirement are identified with the help of customer and exiting system processes, if they are available.
- **Requirement Analysis: -** analysis of requirement stars with Requirement Elicitation. Requirements are analyzed in order to identify inconstancy, defects etc.
- **Requirement Documentation:-** this is the end product of requirement elicitation and analysis. Documentation is very important as it will be the foundation for the design of the software. The documentation is known as SRS.
- **Requirement Review:-**review process is carried out to improve the quality of the SRS. it may also called verification. It should be a continuous activity that is incorporated into the elicitation, analysis, documentation.
  The primary output of requirement engineering process is requirement specification (SRS).

Feasibility Study: - It is the process of evaluation or analysis of the potential impact of a propose project or program.

Five common factors (TELOS)

- **Technical feasibility:** - Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location?
- **Economic feasibility:-**Are the project's cost assumption realistic?
- **Legal feasibility:** - Does the business model realistic?
- **Operational feasibility: -** Is there any market for the product?

- **Schedule feasibility: -** Are the project's schedule assumption realistic?

Elicitation: - It is also called requirement discovery.Requirements are identified with the help of customer and exiting system processes, if they are available. Requirement Elicitation is most difficult is perhaps most critical most error prone most communication intensive aspect of software development.

Various methods of Requirements Elicitation

1. Interviews

- After receiving the problem statement from the customer, the first step is to arrange a meeting with the customer.
- During the meeting or interview, both the parties would like to understand each other.
- The objective of conducting an interview is to understand the customer's expectation from the software

Selection of stakeholder

1. Entry level personnel
2. Middle level stakeholder
3. Managers
4. Users of the software (Most important)

2. Brainstorming Sessions

- Brainstormy is a group technique that may be used during requirement elicitation to understand the requirement
- Every idea will be documented in a way that every can see it
- After brainstormy session a detailed report will be prepared and reviewed by the facilitator

3. Facilitated Application specification approach (FAST)

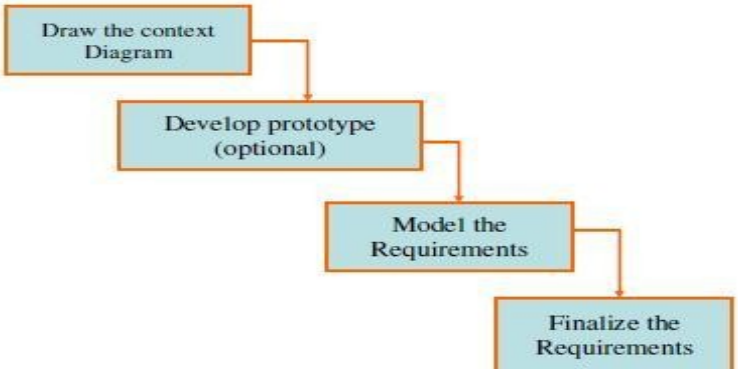
- FAST is similar to brainstormy session and the objective is to bridge the expectation gap-a difference what the developers think they are suppose to build and what the customer think they are going to get
- In order to reduce the gap a team oriented approach is developed for requirement gathering and is called FAST

4. Quality function deployment (QFT)

It incorporates the voice of the customer and converts it in to the document.

Analysis

Requirement analysis phase analyze, refine and scrutinize requirements to make consistent & unambiguous requirements.



1. **Draw the context diagram**

The context diagram is a simple model that defines the boundaries and interface of the proposed system.

2. **Development of prototype**

Prototype helps the client to visualize the proposed system and increase the understanding of requirement. Prototype may help the parties to take final decision.

3. **Model the requirement**

This process usually consists of various graphical representations of function, data entities, external entities and their relationship between them. It graphical view may help to find incorrect, inconsistent, missing requirements. Such models include data flow diagram, entity relationship diagram, data dictionary, state transition diagram.

4. **Finalize the requirement**

After modeling the requirement inconsistencies and ambiguities have been identified and corrected. Flow of data among various modules has been analyzed. Now Finalize and analyzes requirements and next step is to document these requirements in prescribed format.

Documentation

This is the way of representing requirements in a consistent format SRS serves many purpose depending upon who is writing it.

Software requirement specification

Requirements specification is a complete description of the behavior of a system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains non-functional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints).

Need for Software Requirement Specification (SRS)

- The problem is that the client usually does not understand software or the software development process, and the developer often does not understand the client's problem and application area.

- This causes a communication gap between the parties involved in the development project. A basic purpose of software requirements specification is to bridge this communication gap.

Characteristics of good SRS document:- Some of the identified desirable qualities of the SRS documents are the following-

Concise- The SRS document should be concise and at the same time unambiguous, consistent, and complete. An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.

Structured- The SRS document should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope with the customer requirements.

Black-box view- It should only specify what the system should do and refrain from stating how to do. This means that the SRS document should specify the external behaviour of the system and not discuss the implementation issues.

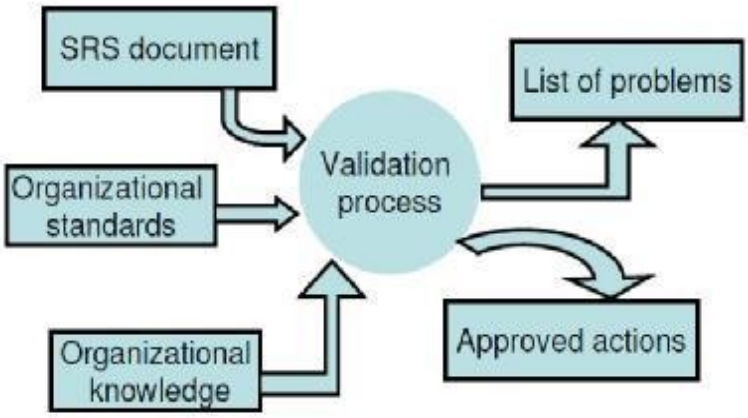
Conceptual integrity- The SRS document should exhibit conceptual integrity so that the reader can easily understand the contents.

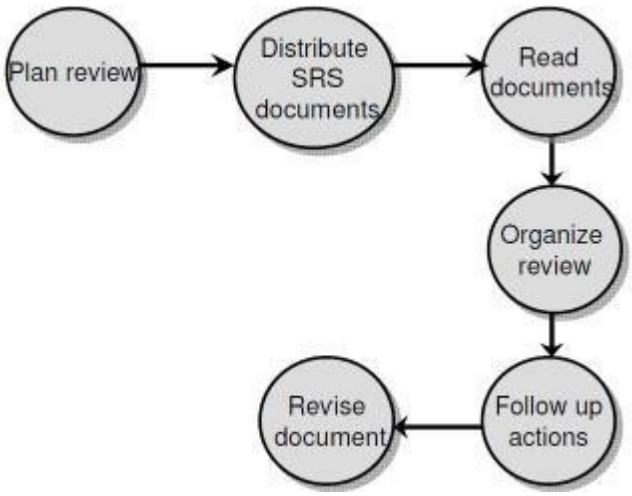
Verifiable- All requirements of the system as documented in the SRs document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Requirements Validation

Requirement Validation is used for checking the document:-

- Completeness & consistency
- Conformance to standards
- Requirements conflicts
- Technical errors
- Ambiguous requirements

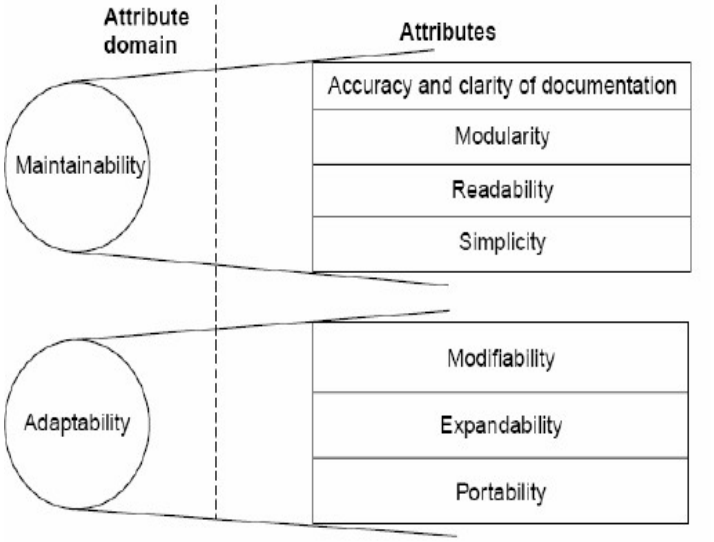


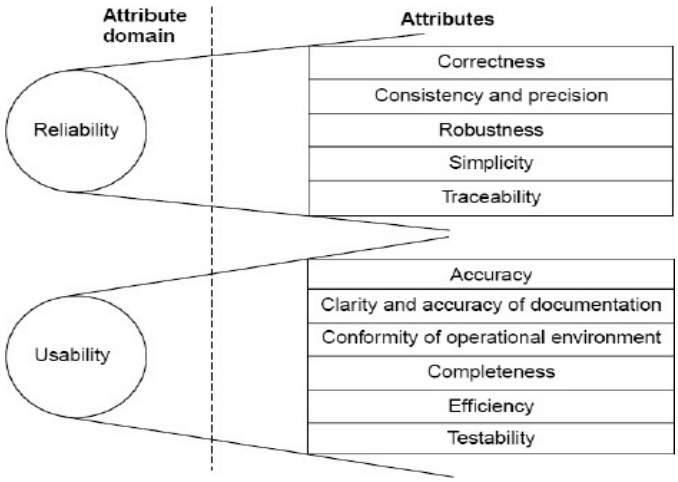


Software Quality

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component or process meets customer or user needs or expectations.

Software Quality attribute





1	Reliability	The extent to which a software performs its intended functions without failure.
2	Correctness	The extent to which a software meets its specifications.
3	Consistency & precision	The extent to which a software is consistent and give results with precision.
4	Robustness	The extent to which a software tolerates the unexpected problems.
5	Simplicity	The extent to which a software is simple in its operations.
6	Traceability	The extent to which an error is traceable in order to fix it.
7	Usability	The extent of effort required to learn, operate and understand the functions of the software

8	Accuracy	Meeting specifications with precision.
9	Clarity & Accuracy of documentation	The extent to which documents are clearly & accurately written.
10	Conformity of operational environment	The extent to which a software is in conformity of operational environment.
11	Completeness	The extent to which a software has specified functions.
12	Efficiency	The amount of computing resources and code required by software to perform a function.
13	Testability	The effort required to test a software to ensure that it performs its intended functions.
14	Maintainability	The effort required to locate and fix an error during maintenance phase.

2.3.1 McCall Software Quality Model

i. Product Operation

Factors which are related to the operation of a product are combined. The factors are:

- Correctness
- Efficiency
- Integrity
- Reliability
- Usability

These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction.

ii. Product Revision

The factors which are required for testing & maintenance are combined and are given below:

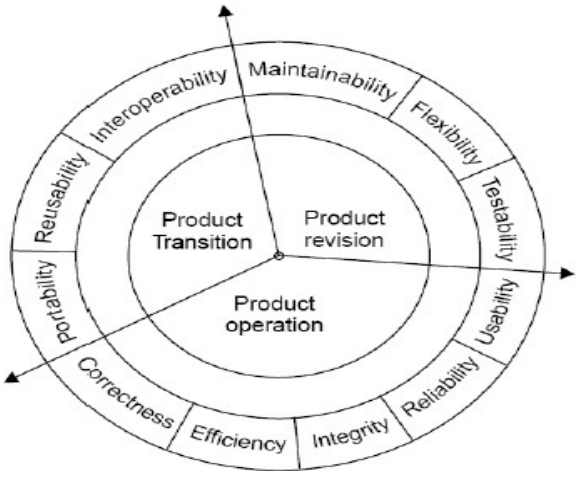
- Maintainability
- Flexibility
- Testability

These factors pertain to the testing & maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision**.**

iii. Product Transition

We may have to transfer a product from one platform to an other platform or from one technology to another technology. The factors related to such a transfer are combined and given below:

- Portability
- Reusability
- Interoperability



Software Quality Assurance

- It is a planned and systematic pattern of all actions necessary to provide adequate confidence that the time or product conforms to established technical requirements."
- Purpose of SQAP is to specify all the works products that need to be produced during the project, activities that need to performed for checking the quality of each of the work product
- It is interested in the quality of not only the final product but also an intermediate product.

Verification:-is the process of determine whether or not product of a given phase of software development full fill the specification established during the previous phase.

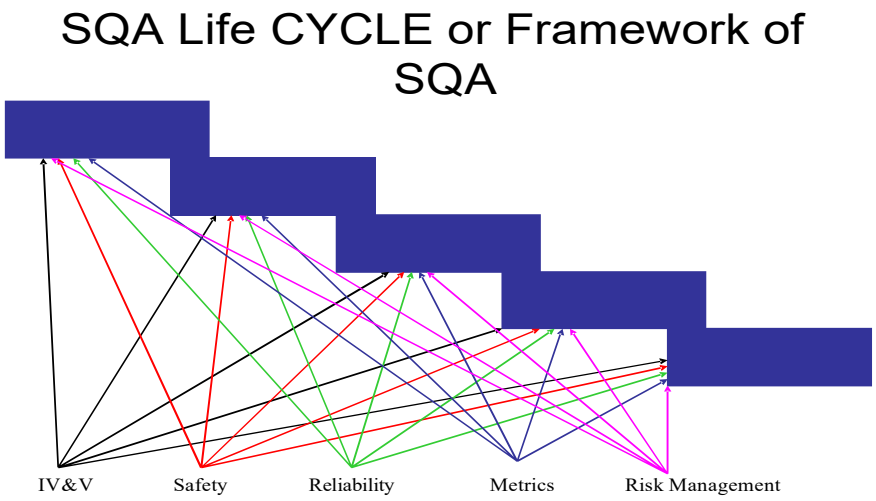
Validation:-is the process of evaluating software at the end of software development to ensure compliance with the software requirement. testing is common method of validation Software V&V is a system engineering process employing rigorous methodologies for evaluating the correctness and quality of the software product throughout the software life cycle.

SQA Plans, Software Quality Frameworks

Quality plan structure

- Product introduction;
- Product plans;
- Process descriptions;
- Quality goals;
- Risks and risk management.

Quality plans should be short, succinct documents.If they are too long, no-one will read them.



International National Organization (ISO)

An international set of standards for quality management. It is applicable to a range of organisations from manufacturing to service industries. ISO 9001 applicable to organisations which design, develop and maintain products. ISO 9001 is a generic model of the quality process that must be instantiated for each organisation using the standard.

Quality standards and procedures should be documented in an organisational quality manual. An external body may certify that an organisation's quality manual conforms to ISO 9000 standards. Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

Process of getting ISO 9000 Certification

- Application
- Pre-assessment
- Document review and adequacy of audit
- Compliance audit
- Registration

- Continued surveillance

The Capability Maturity Model (CMM)

When it is applied to an existing organization's software development processes, it allows an effective approach toward improving them. Eventually it became clear that the model could be applied to other processes. This gave rise to a more general concept that is applied to business processes and to developing people

The Capability Maturity Model involves the following aspects:

- **Maturity Levels***:* a 5-level process maturity continuum - where the uppermost (5th) level is a notional ideal state where processes would be systematically managed by a combination of process optimization and continuous process improvement.
- **Key Process Areas:** a Key Process Area (KPA) identifies a cluster of related activities that, when performed together, achieve a set of goals considered important.
- **Goals:** the goals of a key process area summarize the states that must exist for that key process area to have been implemented in an effective and lasting way. The extent to which the goals have been accomplished is an indicator of how much capability the organization has established at that maturity level. The goals signify the scope, boundaries, and intent of each key process area.
- **Common Features:** common features include practices that implement and institutionalize a key process area. There are five types of common features: commitment to Perform, Ability to Perform, Activities Performed, Measurement and Analysis, and Verifying Implementation.

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

Difference between ISO and SEI-CMM

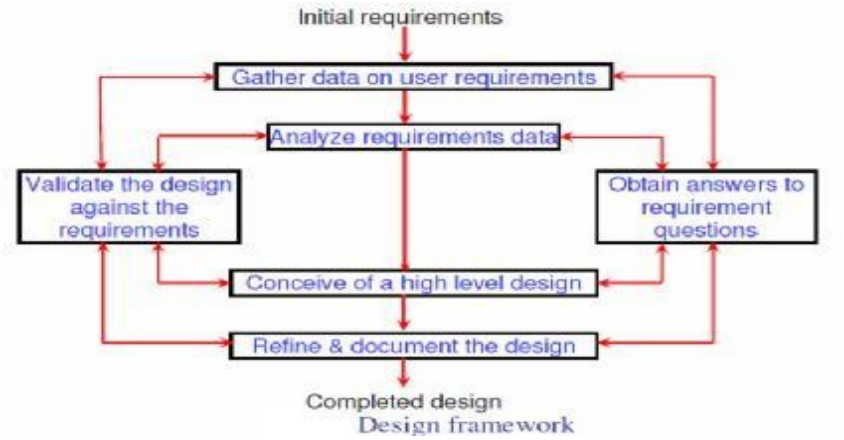
ISO9000	CMM

ISO certification is awarded by an international standard body and can be quoted as an official document	SEI CMM assessment is purely for Internal use
Deals primarily for manufacturing industry and provisioning of services	CMM was developed specially for Software industry and therefore addresses software issues
It aims at level 3 of CMM	Goes beyond Quality Assurance and lead to TQM
Has Customer Focus as primary aim and follows procedural controls	Provide a list of Key Process Areas to proceed from lower CMM level to higher level to provide gradual Quality improvements

Unit-3: Software Design

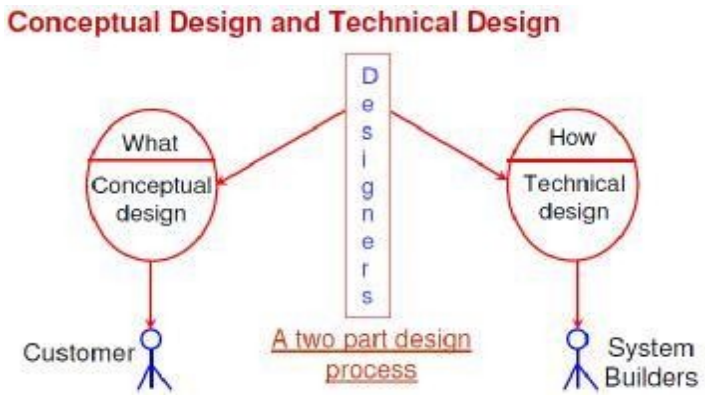
Software design

- Design is the highly significant phase in the software development where the designer plans "how" a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain.
- SRS tell us what a system does and becomes input to design process, which tells us "how" a software system works.
- Software design involves identifying the component of software design, their inner workings, and their interface from the SRS. The principle work of this activity is the software design document (SDD) which is also referred as software design description
- Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) called software design document that is suitable for implementation in a programming language.



Difference between Conceptual and Technical design

- Conceptual design describe the system in language understandable to the customer. it does not contain any technical jargons and is independent of implementation
- By contrast the technical design describe the hardware configuration, software needs, communication interface, input and output of system, network architecture that translate the requirement in to the solution to the customer's problem



Characteristics and objectives of a good software design

Good design is the key of successful product.

- **Correctness:** A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability:** A good design is easily understandable.
- **Efficiency:** It should be efficient.
- **Maintainability:** It should be easily amenable to change.

Features of a design document

- It should use consistent and meaningful names for various design components.
- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram.

Software Design process

- **Architectural Design (Top Level design):-** Describe how software is decomposed and organized into components
- **Detailed Design (Low level design):-**describe the specific behavior of these components. The output of this process is a set of models that records the major decision that has been taken.

Software Design principle

Abstraction

It is a tool that permits a designer to consider a component at abstract level; without worrying about the detail of the implementation of the component.

Encapsulation/Information hiding

the concept of information hiding is to hide the implementation details of shared information and processing items by specifying modules called information hiding modules. Design decisions that are likely to change in the future should be identified and modules should be designed in such a way that those design decisions are hidden from other modules

Coupling and cohesion

Cohesion

It is a measure of the degree to which the elements of a module are functionally related. Cohesion is weak if elements are bundled simply because the perform similar or related functions. Cohesion is weak if elements are bundled simply because they perform similar

or related functions . Cohesion is strong if al parts are needed for the functioning of other parts (e..Important design objective is to maximize module cohesion and minimize module coupling.

Coupling

It is the measure of the degree of interdependence between modules. Coupling is highly between components if they depend heavily on one another, (e.g., there is a lot of communication between them).

Decomposition and modularization

Decomposition and modularization large software in to small independent once, usually with the goal of placing different functionality or responsibility in different component

Design Complexity

Complexity is another design criteria used in the process of decomposition and refinement. A module should be simple enough to be regarded as a single unit for purposes of verification and modification

a measure of complexity for a given module is proportional to the number of other modules calling this module (termed as fan-in), and the number of modules called by the given module (termed as fan-out).

Top-down approach (is also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

Bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose. Mechanisms or be detailed enough to realistically validate the model.

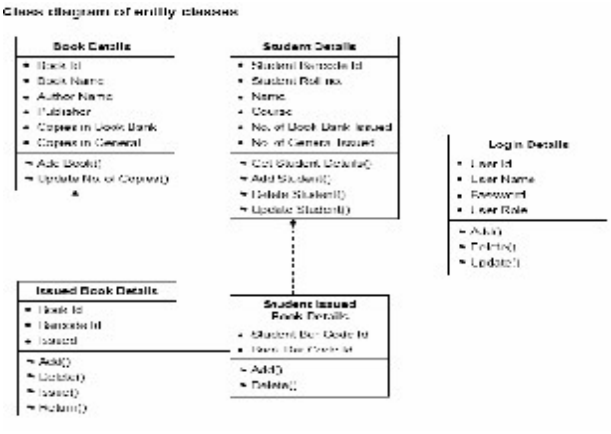
Architectural design

- Architectural design represents the structure of data and program components that are required to build a computer-based system.
- It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.
- Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system.

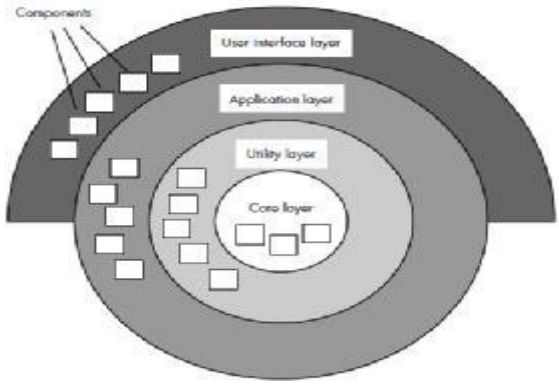
- Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes.
- Once an alternative has been selected, the architecture is elaborated using an architectural design method.

Types of Architectural design

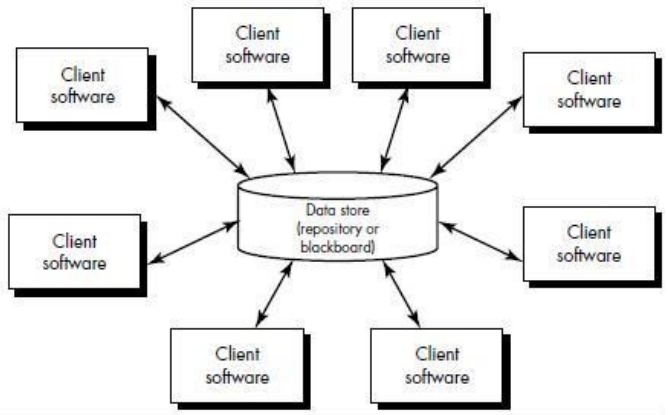
Object-oriented architectures: - The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.



Layered architectures



Data-centered architectures: - A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository. In some cases the data repository is *passive.* That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.



Coupling and cohesion Measures

Coupling: - It is the measure of the degree of interdependence between modules. Coupling is highly between components if they depend heavily on one another, (e.g., there is a lot of communication between them).

Types of Coupling:-

1. Data coupling: communication between modules is accomplished through well-defined parameter lists consisting of data information items

2. Stamp coupling: Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

3. Control coupling: a module controls the flow of control or the logic of another module. This is accomplished by passing control information items as arguments in the argument list.

4. Common coupling: modules share common or global data or file structures. This is the strongest form of coupling both modules depend on the details of the common structure

5. Content coupling: A module is allowed to access or modify the contents of another, e.g. modify its local or private data items. This the strongest form of coupling

Cohesion :- It is a measure of the degree to which the elements of a module are functionally related. Cohesion is weak if elements are bundled simply because the perform similar or related functions. Cohesion is weak if elements are bundled simply because they perform similar or related functions. Cohesion is strong if al parts are needed for the functioning of other parts (e..Important design objective is to maximize module cohesion and minimize module coupling.

Types of Cohesion:-

1.Functional cohesion: A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure or achieved when the components of a module cooperate in performing exactly one function, e.g., POLL_SENSORS, GENERATE_ALARM, etc.

2.Sequential Cohesion: Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

3.Communicational cohesion: is achieved when software units or components of a module sharing a common information or data structure are grouped in one module

4.Procedural cohesion: is the form of cohesion obtained when software components are grouped in a module to perform a series of functions following a certain procedure specified by the application requirements

5. Temporal cohesion: Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span. Examples are functions required to be activated for a particular input event, or during the same state of operation

6.Logical cohesion: refers to modules designed using functions who are logically related, such as input/output functions, communication type functions (such as send and receive),

7.Coincidental cohesion: Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

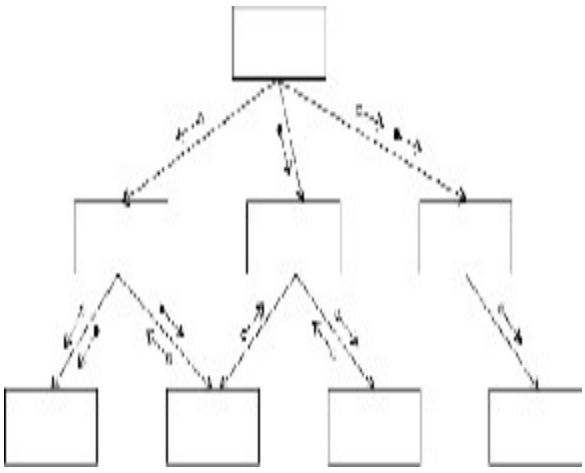
Function oriented design

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional view point.

Design Notations of function oriented design

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts: - function-oriented design, the design can be represented graphically by structure charts. Structure of a program is made up of the modules of that program together with the interconnections between modules. structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the *subordinate* of A, and A is called the *superordinate* of B.



- **Pseudocode: -** Is a tool for planning or documenting the content of the program routine or module as name implies is same as real code. Pseudocode notation can

be used in both the preliminary and detailed design phases. Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as It-Then-Else, While-Do, and End.

Example:-

```
COUNT=0
STOCK=STOCK+QUANTITY
OR
READ THE DATA FROM SOURCE
WRITETHE DATA TO DESTINATION
```

Object Oriented Design

- The object-oriented design approach is fundamentally different from the function-oriented design approaches primarily due to the different abstraction that is used.
- It requires a different way of thinking and partitioning. It can be said that thinking in object-oriented terms is most important for producing truly object-oriented designs.
- Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to do manipulate by the program. Thus, it is orthogonal to function oriented design.
- Object Oriented Design begins with an examination of the real world "things" that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.
- Object Oriented Design is not dependent on any specific implementation language. Problems are modeled using objects.
  Objects have:
  - Behavior (they do things)
  - State (which changes when they do things)

Software measurement and metric

- Pressman explained as "A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of the product or process".
- Fenton defined measurement as " it is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules".
- Measurement is the act of determine a measure

Software metrics

- Pressman explained as " The metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute".
- Software metrics can be defined as "The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

Categories of Metrics

i. **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

ii. **Process metrics**: describe the effectiveness and quality of the processes that produce the software product. Examples are:
- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing
- maturity of the process

iii. **Project metrics:** describe the project characteristics and execution.

Examples are:-
- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Halstead's Software science

- Halstead has proposed metrics for length and volume of a program based of the number of operation and operands.
- Tokens are classified as either operators or operands all software science measures are function of the count of these tokens
- Any symbol or keyboard in a program that specify an algorithmic action is considered an operator ,while a symbol used to represent the data is considered an operand.
- Variable, constants and even labels are operands
- Operators consists of arithmetic symbols such as +,-,/,*and command names such as while ,for, printf, special character such as :=,braces, parentheses etc.
- In a program we define following measurable quantities

  N = N1+N2

  N : program length

where

N1 : total occurrences of operators

N2 : total occurrences of operands

Volume

- The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if uniform binary encoding for the vocabulary is used.

$$V = N^* \log_2 ń$$

Program Level

- The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size) and v is the potential volume

$$L=V^*/V$$

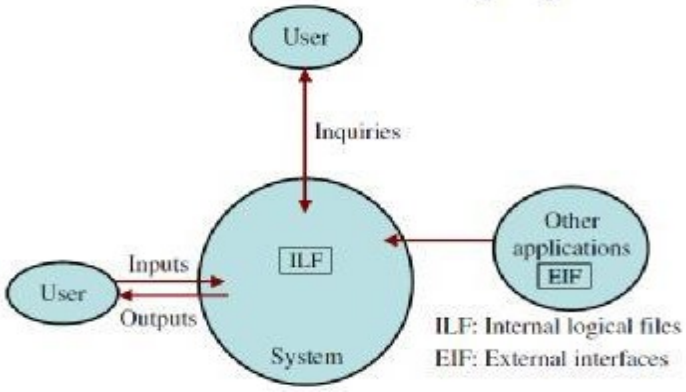
Estimated program Length

$$\tilde{N}= \acute{\eta}1\log_2 \acute{\eta}2+ \acute{\eta}2\log_2 \acute{\eta}2$$

Function points Analysis

- Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.
- It measure functionality from user point of view i.e on the basis of what the user request and receive in return.
- Therefore it deals with the functionality being delivered , and not with LOC, source code, files etc.
- Measuring size in this way has the advantage that size measure is independent of the technology used to deliver the function

The FPA functional units are shown in figure given below:



Counting function points

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

- **Advantages**
  - Users point of view: what user requests & receives from the system
  - Independent of tech.,lang,tool,methods

- Can be estimated from SRS or Design specification Doc.
- Since directly from first phase doc. So easy re-estimation on expansion or modification.

- **Disadvantages**
  - Difficult to estimate
  - Experienced based/subjective

Coding styles- Coding guidelines provide only general suggestions regarding the coding style to be followed.

1) **Do not use a coding style that is too clever or too difficult to understand-** Code should be easy to understand. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.
2) **Avoid obscure side effects-** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code.
3) **Does not use an identifier for multiple purposes-** Programmers often use the same identifier to denote several temporary entities? There are several things which are wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes are as follows:
   - Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult to read and understand the code.
   - Use of variables for multiple purposes usually makes future enhancements more difficult.
4) **The code should be well-documented-** As a rule of thumb, there must be at least one comment line on the average for every three source lines.
5) **Do not use goto statements-** Use of goto statements makes a program unstructured and very difficult to understand.

Unit-4 Software Testing

Software Testing

- Software testing is the process of testing the software product.
- Testing strategy provides a framework or set of activities which are essential for the success of the project. they may include planning, designing of test cases, execution of program with test cases, interpretation of the outcome and finally collection and management of data.
- Testing is the process of executing a program with the intent of finding errors.
- Effective software testing will contribute to the delivery of high quality software product, more satisfied users, lower maintenance cost, more accurate and reliable result.
- Hence software testing is necessary and important activity of software development process.

Error, Mistake, Bug, Fault and Failure

- People make errors. A good synonym is mistake. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.
- When developers make mistakes while coding, we call these mistakes “bugs”.
- A fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.
- A failure occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Test case:

This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

Test suite:

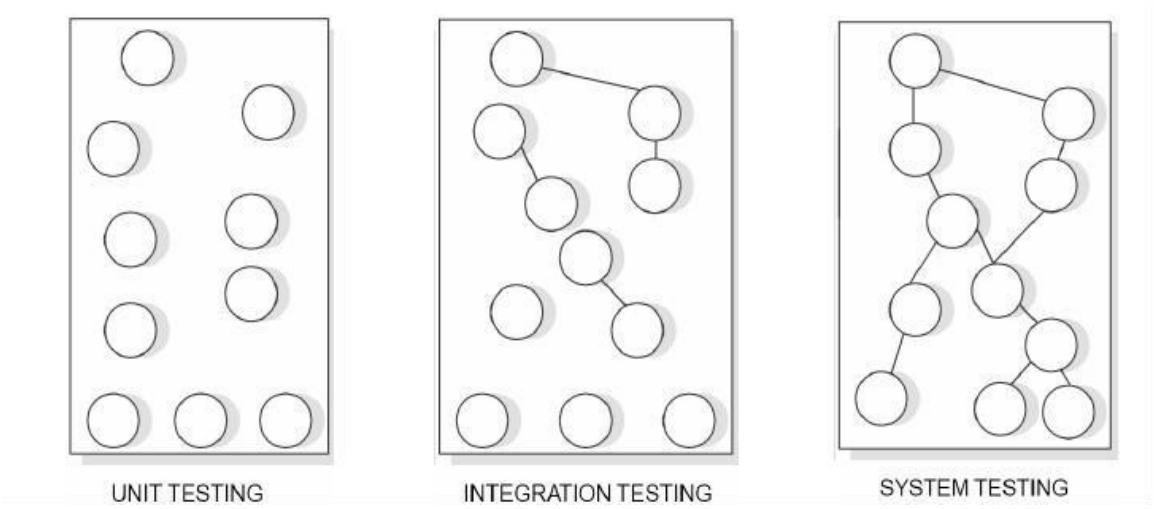
This is the set of all test cases with which a given software product is to be tested.The set of test cases is called a test suite. Hence any combination of test cases may generate a test suite.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Test case template

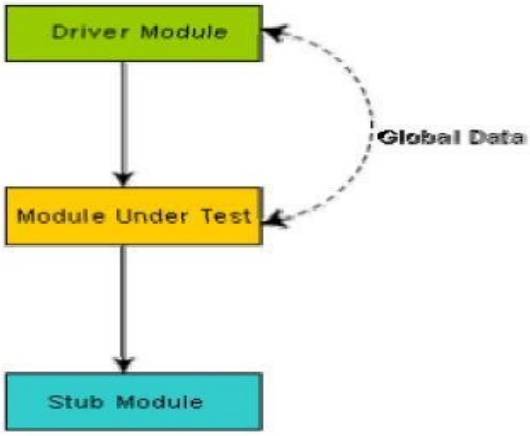
Test data
Inputs which have been devised to test the system
Test cases
Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

Levels of Testing
There are 3 levels of testing:
i. Unit Testing
ii. Integration Testing
iii. System Testing



Unit testing is undertaken after a module has been coded and successfully reviewed.

- Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.
- In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module.



Integration Testing: **Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the componenets were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.

- The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed.
- Focuses on interaction of modules in a subsystem
- Unit tested modules combined to form subsystems
- Test cases to "exercise" the interaction of modules in different ways

Purpose of Integration Testing is to identify bugs of following Types: These are protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values. The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design .

Goal: Test all interfaces between subsystems and the interaction of subsystems

The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

System Testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

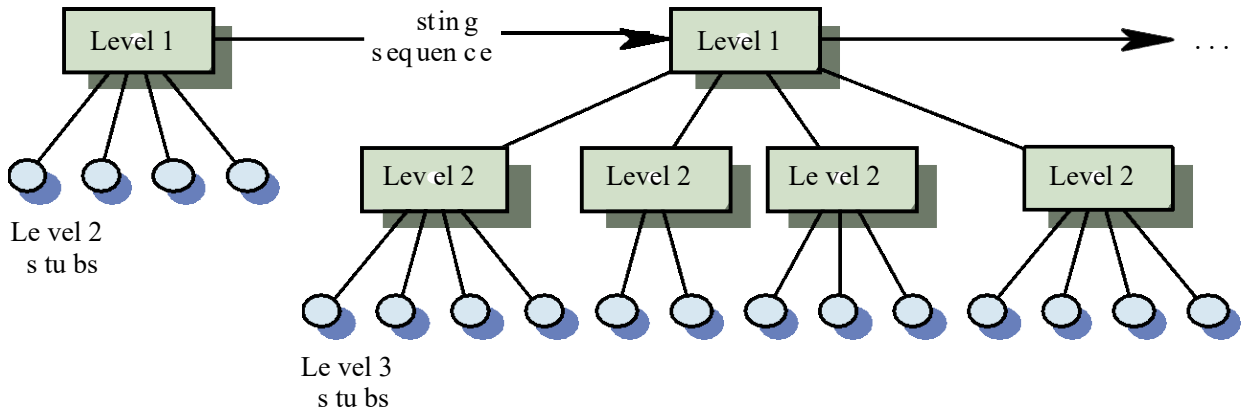
Alpha Testing. Alpha testing refers to the system testing carried out by the test team within the developing organization.

Beta testing. Beta testing is the system testing performed by a select group of friendly customers.

Acceptance Testing. Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

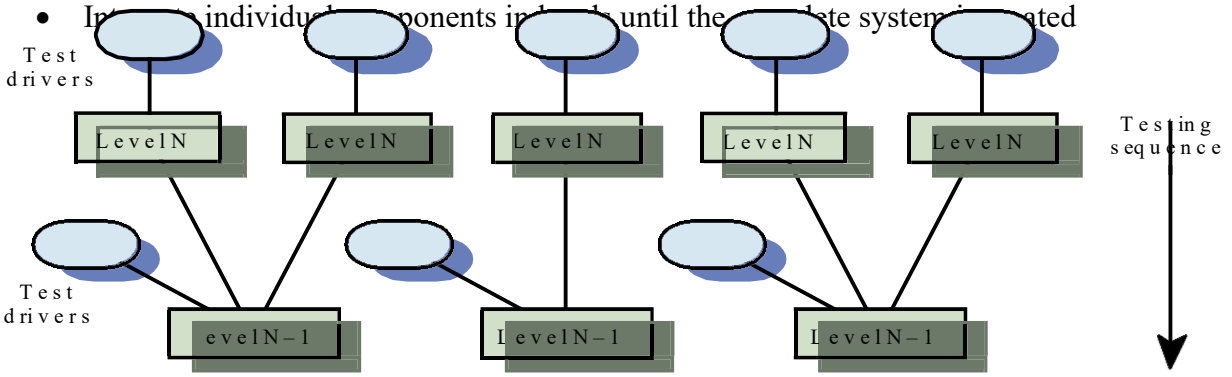
Top Down and Bottom up testing

- starts with the main routine and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested.
- Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test



Bottom-up testing

- each subsystem is tested separately and then the full system is tested
- Primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested
- In individual components in levels until the complete system is created



Functional Testing (Black Box Testing)

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required.

The following are the main approaches to designing black box test cases.

1. **Boundary value analysis**
2. **Equivalence class portioning**
3. **Cause Effect graphing**

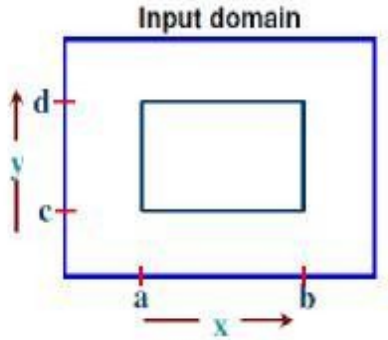
1. Boundary value analysis

- Test cases that are close to boundary value condition have a higher chance of detecting an error.
- Boundary condition means , an input value may be on the boundary, just below the boundary (upper side) or just above the boundary (lower side)
- Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes.

- For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.
- Suppose we have an input variable x with a range from 1-100 the boundary values are 1,2,99 and 100.

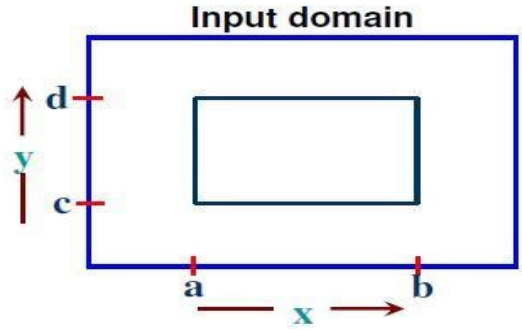
Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \le x \le b$$
$$c \le y \le d$$



- Programs often fail on special values
- These values often lie on boundary of equivalence classes
- Test cases that have boundary values have *high yield*
- These are also called *extreme cases*
- For each equivalence class
    - **choose values on the edges of the class**
    - **choose values just outside the edges**
- E.g. if 0 <= x <= 1.0
  0.0 , 1.0 are edges inside
  -0.1,1.1 are just outside
- Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \le x \le b$$
$$c \le y \le d$$



Hence both the input x and y are bounded by two interval [a,b] and [c,d] respectively. for input x, we may design test cases with value a and b, just above a and also just below b, for

input y, we may have values c and d ,design just above c and just below d. These test cases will have more chance to detect an error

- The basic idea of boundary value analysis is to use input values at their minimum, just minimum, a nominal value, just below their their maximum and at their maximum
- Thus for a program of n variable ,boundary value analysis yields 4n+1 test cases.

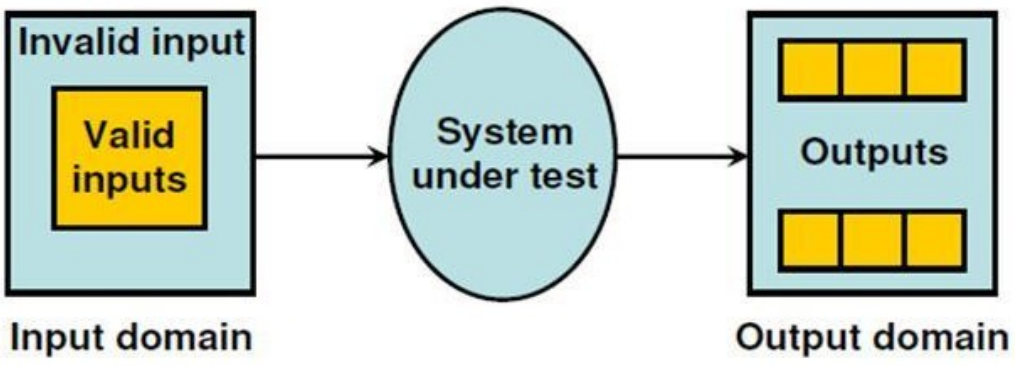
2. Equivalence Class Partitioning

- In this approach, the domain of input values to a program is partitioned into a finite number of equivalence classes, so that if the program works correctly for a value then it will work correctly for all the other values in that class.
- This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.
- The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.
- Equivalence classes for a software can be designed by examining the input data and output data.
- The following are some general guidelines for designing the quivalence classes

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class.



Equivalence partitioning

- E.g. range 0< value<Max specified
  - one range is the valid class
  - input < 0 is an invalid class

– input > max is an invalid class

3. Cause effect graphing

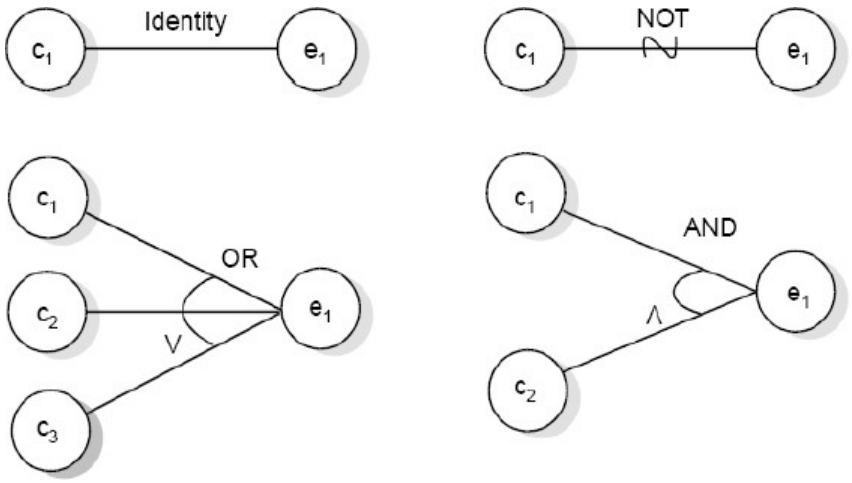
- One weakness with the equivalence class partitioning and boundary value methods is that they consider each input separately. That is, both concentrate on the conditions and classes of one input.
- They do not consider combinations of input circumstances that may form interesting situations that should be tested.
- One way to exercise combinations of different input conditions is to consider all valid combinations of the equivalence classes of input conditions.
- This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors.
- For example, if there are *n different input conditions, such that a* combination of the input conditions is valid, we will have 2^ test cases.
- Cause-effect graphing is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. The technique starts with identifying causes and effects of the system under testing.
- A *cause is a distinct* input condition, and an *effect is a distinct output condition.*
- *Each* condition forms a node in the cause-effect graph. The conditions should be stated such that they can be set to either true or false.

Steps

1. Causes & effects in the specifications are identified.
   - **A cause is a distinct input condition or an equivalence class of input conditions.**
   - **An effect is an output condition or a system transformation.**

2. The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes & effects.
3. The graph is converted in to decision
4. The columns in the decision table are converted into test cases.

Basic cause effect graph symbols



Structural Testing (White Box Testing)

- A complementary approach to functional testing is called structural / white box testing.
- It permits us to examine the internal structure of the program.
- In the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

Various criteria of structural testing

- Path Testing
- Flow Graph
- DD Graph
- Cyclometer complexity
- Graph Matics
- mutation testing

1. Path Testing

Path based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

This type of testing involves:

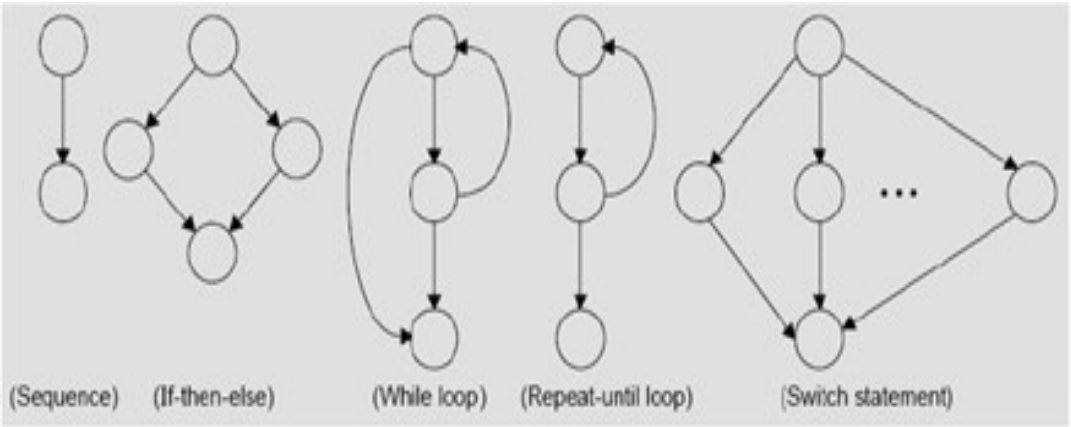
- generating a set of paths that will cover every branch in the program.
- finding a set of test cases that will execute every path in the set of program paths.

Basically two types of graph are designed through path testing:-

i) Control flow graph

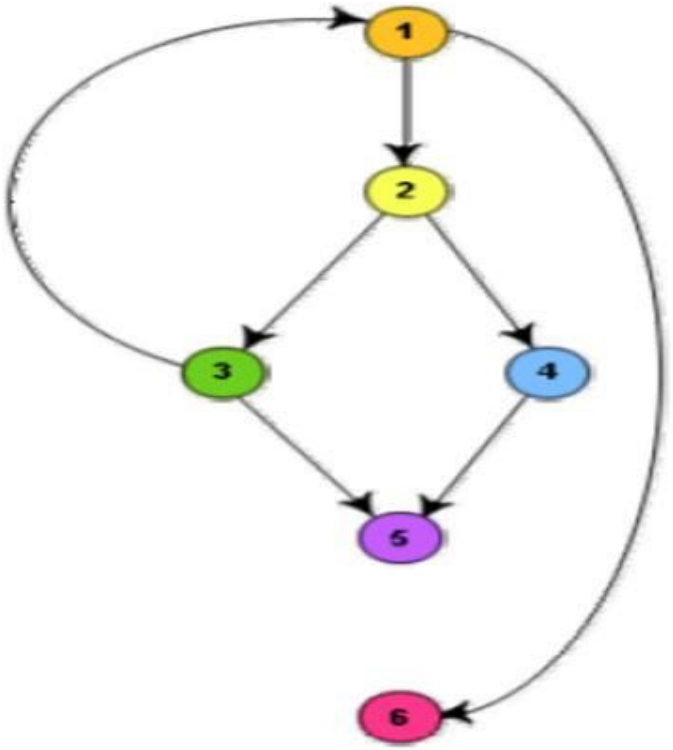
The control flow of a program can be analyzed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control. If I and j are nodes in the program graph ,there is an edge from node I to node j if the statement (fragment) corresponds to node j can be executed immediately after the statement

corresponding to node j.Flow graph can easily be generated from the code of any problem



Example

```
int compute_gcd(int x, int y){
1 while(x! =y){
2       if(x>y) then
3               x=x-y;
4       else  y=y-x;
5 }
6 return x;
}
```



- Flow graph representation is the first step of path testing
- Second step is to draw the DD graph from the flow graph
- DD path graph is known as decision to decision path graph in which on concentrate decision nodes.
- The nodes of flow graph which are sequence are combined into single node.
- Hence DD graph is directed graph in which nodes are sequence of statements and edges show control flow between nodes
- The DD path graph is used to find independent path.
- Execute all independent path at least once.

2. Cyclomatic Complexity

- Is also known as structural complexity because it gives internal view of code
- For more complicated programs it is not easy to determine the number of independent paths of the program.
- McCabe's cyclomatic complexity defines the number of linear independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute.
- The McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E - N + 2(P)$$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

Method 2: An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program.

If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

Difference between functional and structural testing

Black-box testing	White Box Testing
Functional Testing also called Specification based Testing.	Structural testing also called Code Based Testing
In **functional testing**, the program or system is treated as a blackbox. It is subjected to inputs, and its outputs are verified for conformance to specified behaviour. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation.	**Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.
• Functional testing methods: Boundary value, worst case, robustness, Equivalence partitioning, Cause effect graphing, Decision table based testing etc.	• Structural testing methods: Data flow, control flow, Mutation, Path testing, graph metrics based testing etc.

Mutation Testing

- Software is first tested by using an initial test suite built up from the different white box testing strategies.
- After the initial testing is complete, mutation testing is taken up.
- Mutation testing is a fault based technique.
- The idea behind mutation testing is to make few arbitrary changes to a program at a time.
- Multiple copies of a program are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test

data are capable of detecting the change between the original program and the mutated program

- A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated
- Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically

Performance testing

- Performance testing is carried out to check whether the system needs the nonfunctional requirements identified in the SRS document.
- There are several types of performance testing. Among of them nine types are discussed below.
- The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document.
- All performance tests can be considered as black-box tests.
    - Stress testing
    - Volume testing
    - Configuration testing
    - Compatibility testing
    - Regression testing
    - Recovery testing
    - Maintenance testing
    - Documentation testing
    - Usability testing

Coding

- Coding is undertaken once the design phase is complete and the design document have been successfully reviewed
- In the coding phase every module identified and specified in the design document is independently coded and unit tested
- Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards.
- Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously.

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

- representative coding standards
- representative coding Guidelines

Coding Standards

- Programmers spend more time reading code than writing code
- They read their own code as well as other programmers code
- Readability is enhanced if some coding conventions are followed by all
- Coding standards provide these guidelines for programmers
- Generally are regarding naming, file organization, statements/declarations,
- Naming conventions

Coding Guidelines

- Package name should be in lower case (mypackage, edu.iitk.maths)
- Type names should be nouns and start with uppercase (Day, DateOfBirth,…)
- Var names should be nouns in lowercase; vars with large scope should have long names; loop iterators should be i, j, k…
- Const names should be all caps
- Method names should be verbs starting with lower case (eg getValue())
- Prefix *is* should be used for boolean method
-

Exit

Unit-5 Software Maintenance and project management

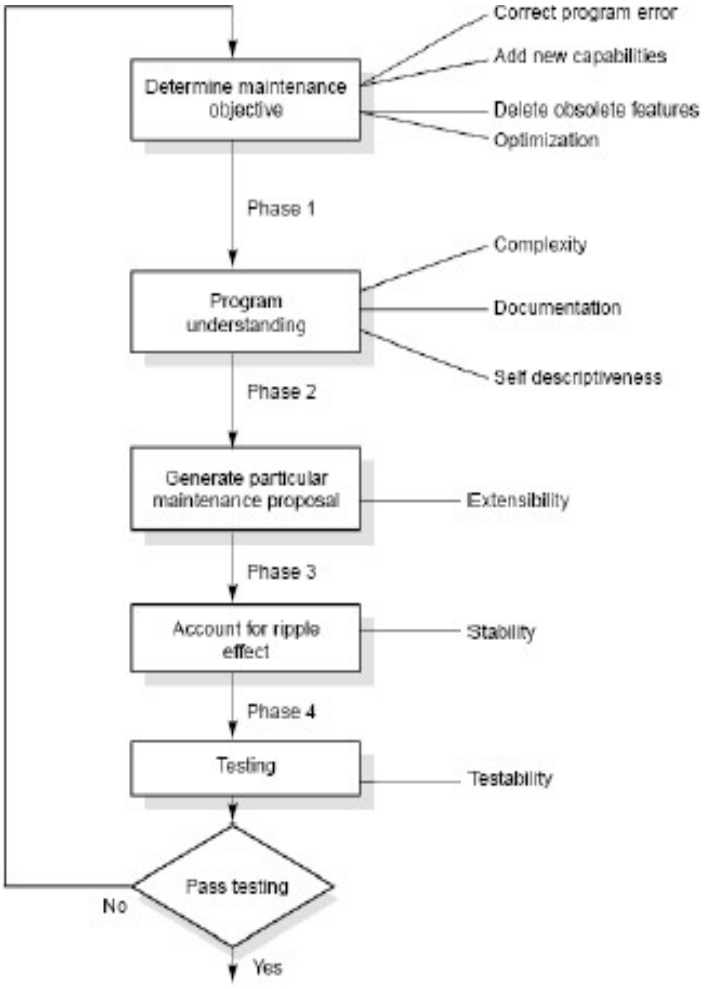
Software Maintenance

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization. Software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.70 percent of the cost of software is devoted to maintenance.

Need for Software Maintenance:

- Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.
- software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.
- 70 percent of the cost of software is devoted to maintenance

Software Maintenance Process



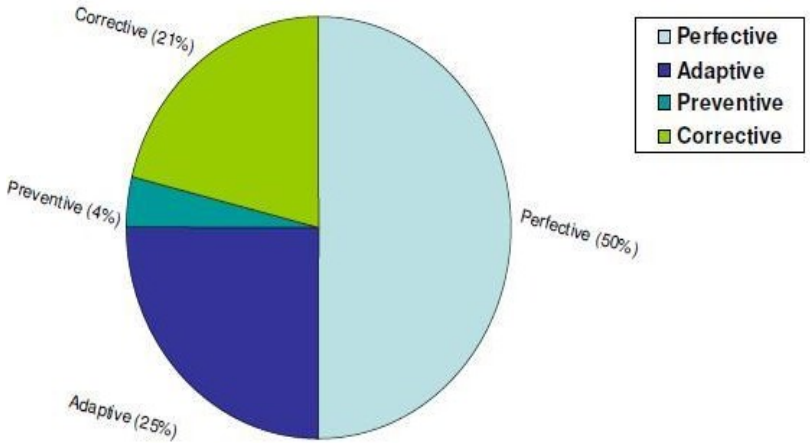
Various types of maintenance

Corrective: Corrective maintenance of a software product is necessary 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.

Distribution of maintenance effort



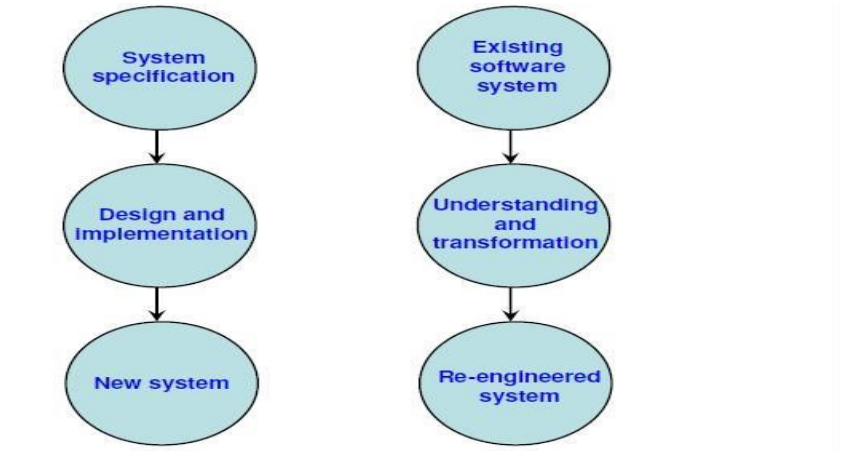
Estimation of maintenance cost

- Maintenance Effort is very significant and consume about 40-70 % of the cost of entire life cycle
- It is advisable to invent more effort in early phase of software life cycle to reduce the maintenance cost
- The defect repair ratio increase heavily from analysis phase to implementation phase
- Good software engineering techniques such as precise specification, loose coupling and configuration management all reduce maintenance cost.

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Software RE-Engineering

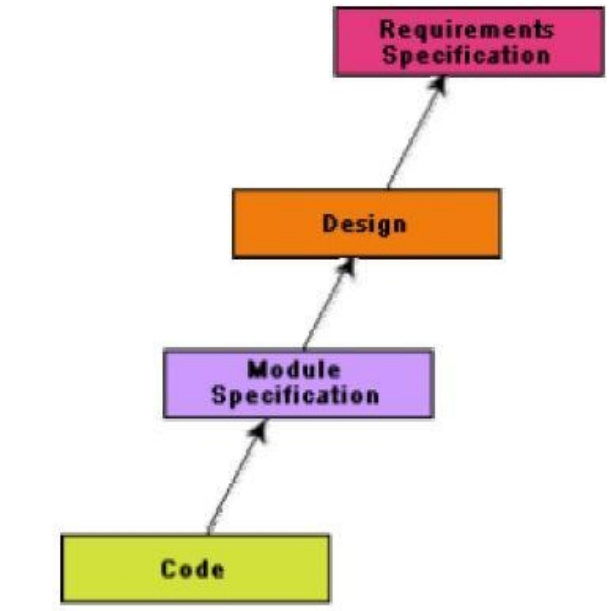
Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable. The critical distinction between re-engineering and new software development is the starting point for the development as shown in Fig:-



Comparison of new software development with re-engineering

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.



- **Redocumentation: -** Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level.
- **Design recovery: -** Design recovery entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains.

Software Configuration management (CM)

Configuration management (CM) is the process of controlling and documenting change to a developing system. As the size of an effort increase, so does the necessity of implementing effective CM. Software configuration management (SCM) is a set of activities that are designed to control change by identifying the work products that are likely to change, establishing relationships among them.The process of software development and maintenance is controlled is called configuration management. The configuration management is different in development and maintenance phases of life cycle due to different environments.

Configuration Management Activities: - The activities are divided into four broad categories.

1. The identification of the components and changes
2. The control of the way by which the changes are made
3. Auditing the changes
4. Status accounting recording and documenting all the activities that have take place

Functions of SCM

- **Identification -**identifies those items whose configuration needs to be controlled, usually consisting of hardware, software, and documentation.
- **Change Control -** establishes procedures for proposing or requesting changes, evaluating those changes for desirability, obtaining authorization for changes, publishing and tracking changes, and implementing changes. This function also identifies the people and organizations who have authority to make changes at various levels.
- **Status Accounting -**is the documentation function of CM. Its primary purpose is to maintain formal records of established configurations and make regular reports of configuration status. These records should accurately describe the product, and are used to verify the configuration of the system for testing, delivery, and other activities.
- **Auditing -**Effective CM requires regular evaluation of the configuration. This is done through the auditing function, where the physical and functional configurations are compared to the documented configuration. The purpose of auditing is to maintain the integrity of the baseline and release configurations for all controlled products

SCM Terminology

1. Version Control

- A version control tool is the first stage towards being able to manage multiple versions.
- Once it is in place, a detailed record of every version of the software must be kept. This comprises the-
  - Name of each source code component, including the variations and revisions
  - The versions of the various compilers and linkers used
  - The name of the software staff who constructed the component
  - The date and the time at which it was constructed

2. Change control process

Change control process comes into effect when the software and associated documentation are delivered to configuration management change request form as shown in fig which should record the recommendations regarding the change.

3. Software documentation

It is the written record of the facts about a software system recorded with the intent to convey purpose, content and clarity.

Two type of documentation:-

User documentation

SCM Activities

- **Configuration item identification**
  - modeling of the system as a set of evolving components
- **Promotion management**
  - is the creation of versions for other developers
- **Release management**
  - is the creation of versions for the clients and users
- **Branch management**
  - is the management of concurrent development
- **Variant management**
  - is the management of versions intended to coexist
- **Change management**
  - is the handling, approval and tracking of change requests

5.9 CASE (Computer aided software engineering)

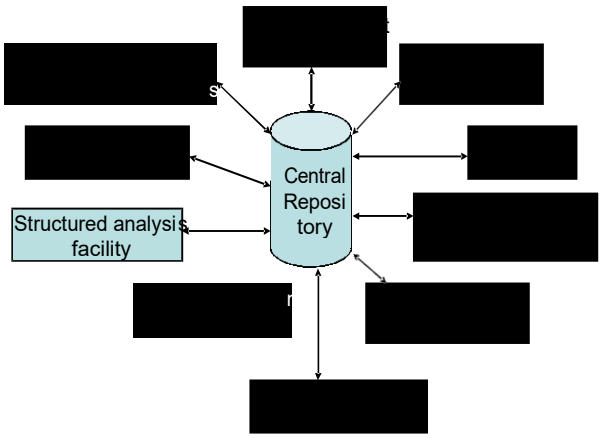
- CASE tool is a generic term used to denote any form of automated support for software engineering
- CASE tool can mean any tool used to automate some activity associated with software development
- Some CASE tools are used in phase- related task such as structured analysis, design, coding testing etc
- Other are related to non phase activities such as project management and configuration management etc
- Primary objectives of deploying case tools are:-
  - To increase productivity

– To produce better quality software at low cost

Benefits of CASE

- Cost saving through all development phases. CASE put the effort reduction between 30% to 40%
- Use of CASE tool leads to considerable improvement to quality
- CASE tools help to produces high quality and consistent documents
- Use of CASE environment has an impact on style of working of a company, and makes it conscious of structured and orderly approach

CASE Environment



Constructive cost model (COCOMO)

- COCOMO is one of the most widely used software estimation models in the world
- It was developed by Barry Boehm in 1981
- COCOMO predicts the effort and schedule for a software product development based on inputs relating to the **size** of the software and a number of **cost drivers** that affect productivity

COCOMO has three different models that reflect the complexity:

1. Basic Model
2. Intermediate Model
3. Detailed Model

The Development Modes: Project Characteristics

Basic Model

Basic model aim at estimating, in a quick and rough fashion, most of the small to medium sized software projects.These models of software development are considered in this model: organic, semi -detected and embedded. KLOC (thousands of lines of code) is a traditional measure of how large a computer program is or how long or how many people it will take to write it.

Basic cocomo coefficient

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Organic Mode

- developed in a familiar, stable environment,
- similar to the previously developed projects
- relatively small and requires little innovation

Semidetached Mode

- intermediate between Organic and Embedded

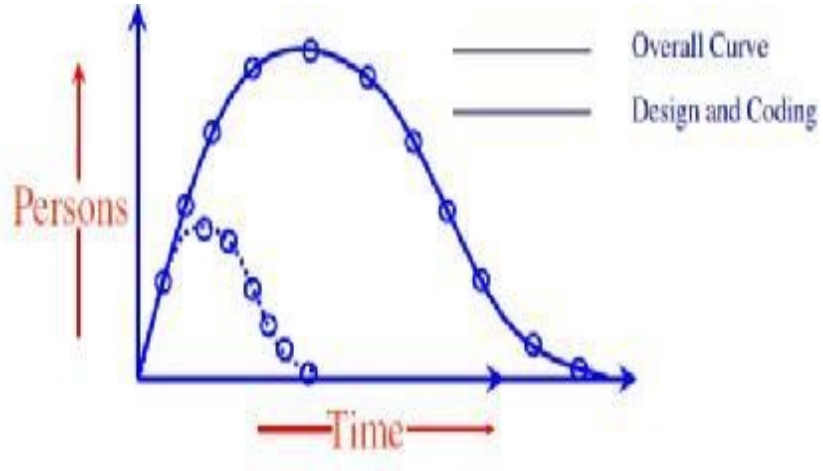
Embedded Mode

- tight, inflexible constraints and interface requirements
- The product requires great innovation

Putnam Resource Model

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. It analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$



Software Risk

Software Risk is future uncertain events with a probability of occurrence and a potential for loss .Risk identification and management are the main concerns in every software project. Effective analysis of software risks will help to effective planning and assignments of work.

Categories of risks

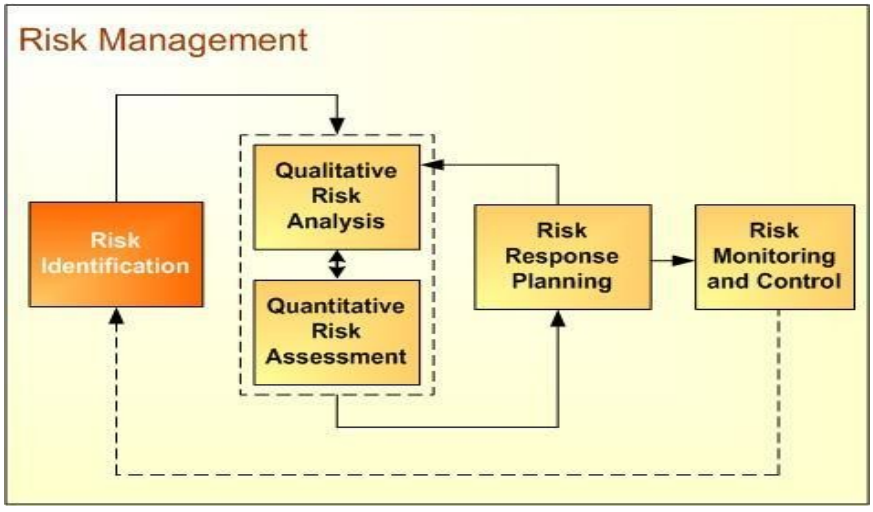
- Schedule Risk
- Operational risk
- Technical risk

Risk management is the identification, assessment, and prioritization of risks followed by coordinated and economical application of resources to minimize, monitor, and control the probability and/or impact of unfortunate events.

Risk Management Process, describes the steps you need to take to identify, monitor and control risk. Within the Risk Process, a risk is defined as any future event that may prevent you to meet your team goals. A Risk Process allows you to identify each risk, quantify the impact and take action now to prevent it from occurring and reduce the impact should it eventuate.

This Risk Process helps you:

- Identify critical and non-critical risks
- Document each risk in depth by completing Risk Forms
- Log all risks and notify management of their severity
- Take action to reduce the likelihood of risks occurring
- Reduce the impact on your business, should risk eventuate



- **Risk identification**, Determining what risks or hazards exist or are anticipated, their characteristics, remoteness in time, duration period, and possible outcomes. **Risk analysis** is the process of defining and analyzing the dangers to individuals, businesses and government agencies posed by potential natural and human-caused adverse events. In IT, a risk analysis report can be used to align technology-related objectives with a company's business objectives. A risk analysis report can be either quantitative or qualitative.

- **Risk Planning** Risk Planning is developing and documenting organized, comprehensive, and interactive strategies and methods for identifying risks. It is also used for performing risk assessments to establish risk handling priorities, developing risk handling plans, monitoring the status of risk handling actions, determining and obtaining the resources to implement the risk management strategies. Risk planning is used in the development and implementation of required training and communicating risk information up and down the project stakeholder organization.
- **Risk monitoring and control** is the process of identifying and analyzing new risk, keeping track of these new risks and forming contingency plans incase they arise. It ensures that the resources that the company puts aside for a project is operating properly. Risk monitoring and control is important to a project because it helps ensure that the project stays on track.