NOTES

ON

SOFTWARE ENGINEERING

CSE-6th SEM

UNIT-I
INTRODUCTION TO SOFTWARE ENGINEERING

Software: Software is

(1) Instructions (computer programs) that provide desired features, function, and performance, when executed
(2) Data structures that enable the programs to adequately manipulate information,
(3) Documents that describe the operation and use of the programs.

Characteristics of Software:

(1) Software is developed or engineered; it is not manufactured in the classical sense.
(2) Software does not "wear out"
(3) Although the industry is moving toward component-based construction, most software continues to be custom built.

Software Engineering:

(1) The systematic, disciplined quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.
(2) The study of approaches as in (1)

EVOLVING ROLE OF SOFTWARE:

Software takes dual role. It is both a **product** and a **vehicle** for delivering a product.

As a **product**: It delivers the computing potential embodied by computer Hardware or by a network of computers.

As a **vehicle**: It is information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as single bit or as complex as a multimedia presentation. Software delivers the most important product of our time-information.

1. It transforms personal data
2. It manages business information to enhance competitiveness
3. It provides a gateway to worldwide information networks
4. It provides the means for acquiring information.
5. Dramatic Improvements in hardware performance
6. Vast increases in memory and storage capacity
7. A wide variety of exotic input and output options

THE CHANGING NATURE OF SOFTWARE:

The 7 broad categories of computer software present continuing challenges for software engineers:

1) System software
2) Application software
3) Engineering/scientific software
4) Embedded software
5) Product-line software
6) Web-applications
7) Artificial intelligence software.

- **System software:** System software is a collection of programs written to service other programs. The systems software is characterized by

  heavy interaction with computer hardware

  1. heavy usage by multiple users
  2. concurrent operation that requires scheduling, resource sharing, and sophisticated process management
  3. complex data structures
  4. multiple external interfaces

E.g. compilers, editors and file management utilities.

- **Application software:**
  - Application software consists of standalone programs that solve a specific business need.
  - It facilitates business operations or management/technical decision making.
  - It is used to control business functions in real-time

  E.g. point-of-sale transaction processing, real-time manufacturing process control.

Engineering/Scientific software: Engineering and scientific applications range
- from astronomy to volcanology
- from automotive stress analysis to space shuttle orbital dynamics
- from molecular biology to automated manufacturing

computer aided design, system simulation and other interactive applications.

Embedded software:

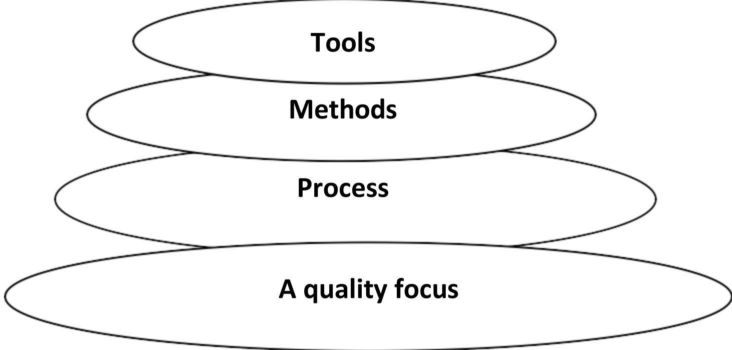
Embedded software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself.

- It can perform limited and esoteric functions or provide significant function and control capability.

E.g. Digital functions in automobile, dashboard displays, braking systems etc.

A GENERIC VIEW OF PROCESS

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY:



Software Engineering Layers

Software engineering is a layered technology. Any engineering approach must rest on an organizational commitment to quality. The bedrock that supports software engineering is a quality focus.

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers. Process defines a framework that must be established for effective delivery of software engineering technology.

The software forms the basis for management control of software projects and establishes the context in which

1.technical methods are applied,
2.work products are produced,
3. milestones are established,
4 quality is ensured,

And change is properly managed.

A PROCESS FRAMEWORK:

- Software process must be established for effective delivery of software engineering technology.
- **A process framework** establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.
- The process framework encompasses a set of umbrella activities that are applicable across the entire software process.
- Each framework activity is populated by a set of software engineering actions
- Each software engineering action is represented by a number of different task sets- each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones.

"A **process** defines who is doing what, when, and how to reach a certain goal."

\

THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI):

The CMMI represents a process meta-model in two different ways:

- As a continuous model
- As a staged model.

Each process area is formally assessed against specific goals and practices and is rated according to the following capability levels.

Level 0: Incomplete. The process area is either not performed or does not achieve all goals and objectives defined by CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed;

Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributes and work products, measures and other process-improvement information to the organizational process assets".

Level 4: Quantitatively managed. All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment."Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

Level 5: Optimized. All level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration"

The CMMI defines each process area in terms of “specific goals” and the “specific practices” required to achieve these goals. Specific practices refine a goal into a set of process-related activities.

The specific goals (SG) and the associated specific practices(SP) defined for project planning are

SG 1 Establish estimates

SP 1.1 Estimate the scope of the project

SP 1.2 Establish estimates of work product and task attributes

SP 1.3 Define project life cycle

SP 1.4 Determine estimates of effort and cost

SG 2 Develop a Project Plan

SP 2.1 Establish the budget and schedule

SP 2.2 Identify project risks

SP 2.3 Plan for data management

SP 2.4 Plan for needed knowledge and skills

SP 2.5 Plan stakeholder involvement

SP 2.6 Establish the project plan

SG 3 Obtain commitment to the plan

SP 3.1 Review plans that affect the project

SP 3.2 Reconcile work and resource levels

SP 3.3 Obtain plan commitment

PERSONAL AND TEAM PROCESS MODELS:

The best software process is one that is close to the people who will be doing the work.Each software engineer would create a process that best fits his or her needs, and at the same time meets the broader needs of the team and the organization. Alternatively, the team itself would create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.

Personal software process (PSP)

The personal software process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.

The PSP process model defines five framework activities: planning, high-level design, high level design review, development, and postmortem.

Planning: This activity isolates requirements and, base on these develops both size and resource estimates. In addition, a defect estimate is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High level design: External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High level design review: Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development: The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important task and work results.

Postmortem: Using the measures and metrics collected the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

PSP stresses the need for each software engineer to identify errors early and, as important, to understand the types of errors that he is likely to make.

PSP represents a disciplined, metrics-based approach to software engineering.

Team software process (TSP): The goal of TSP is to build a "self-directed project team that organizes itself to produce high-quality software. The following are the objectives for TSP:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams(IPT) of 3 to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

A self-directed team defines

- roles and responsibilities for each team member
- tracks quantitative project data
- identifies a team process that is appropriate for the project
- a strategy for implementing the process
- defines local standards that are applicable to the teams software engineering work;
- continually assesses risk and reacts to it
- Tracks, manages, and reports project status.
-

TSP defines the following framework activities: launch, high-level design, implementation, integration and test, and postmortem.
TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
Scripts define specific process activities and other more detailed work functions that are part of the team process.
Each project is "launched" using a sequence of tasks.

The following launch script is recommended

- Review project objectives with management and agree on and document team goals
- Establish team roles
- Define the teams development process
- Make a quality plan and set quality targets
- Plan for the needed support facilities

PROCESS MODELS

Prescriptive process models define a set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software. These process models are not perfect, but they do provide a useful roadmap for software engineering work.

A prescriptive process model populates a process framework with explicit task sets for software engineering actions.

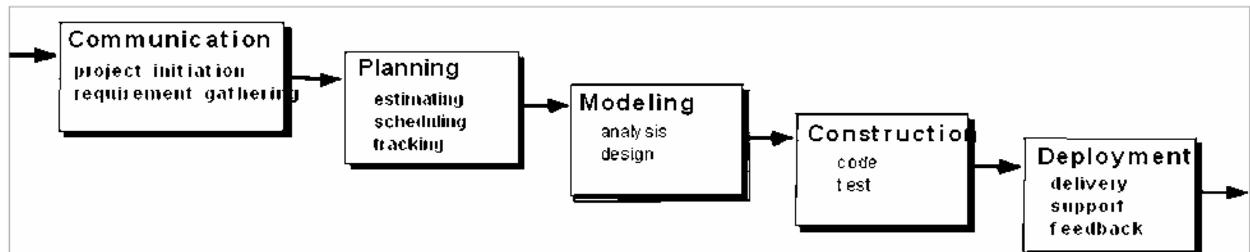
THE WATERFALL MODEL:

The waterfall model, sometimes called the *classic life cycle*, suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

Context: Used when requirements are reasonably well understood.

Advantage:

It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



The **problems** that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exist at the beginning of many projects.
3. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

INCREMENTAL PROCESS MODELS:

1) The incremental model
2) The RAD model

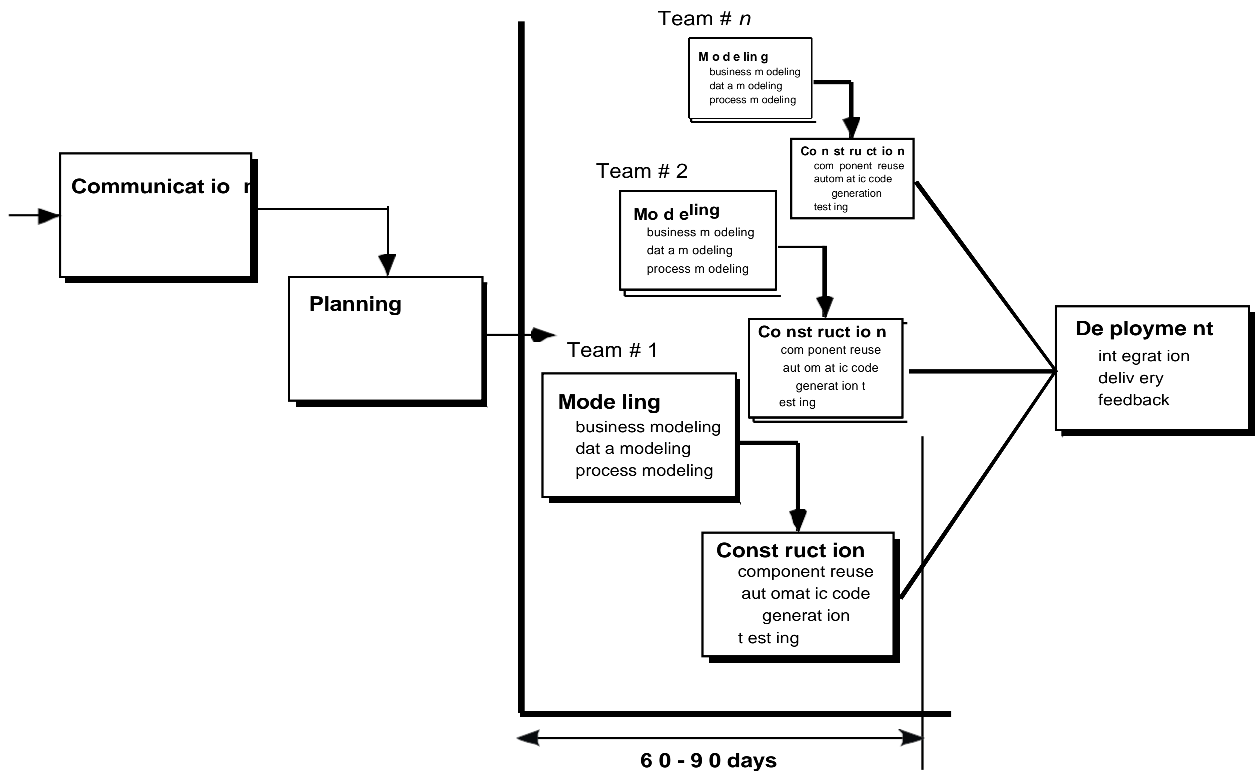
THE INCREMENTAL MODEL:

Context: Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, additional staff can be added to implement the next increment. In addition, increments can be planned to manage technical

THE RAD MODEL:

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a “high-speed” adaption of the waterfall model, in which rapid development is achieved by using a component base construction approach.

Context: If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a “fully functional system” within a very short time period.



The RAD approach maps into the generic framework activities.
Communication works to understand the business problem and the information characteristics that the software must accommodate.

Planning is essential because multiple software teams works in parallel on different system functions.

Modeling encompasses three major phases- business modeling, data modeling and process modeling- and establishes design representation that serve existing software components and the application of automatic code generation.

Deployment establishes a basis for subsequent iterations.

The RAD approach has **drawbacks**:

For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.

If developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail

If a system cannot be properly modularized, building the components necessary for RAD will be problematic

If high performance is an issue, and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work; and

RAD may not be appropriate when technical risks are high.

EVOLUTIONARY PROCESS MODELS:

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

PROTOTYPING:

Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.

Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

Advantages:

The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.

The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.

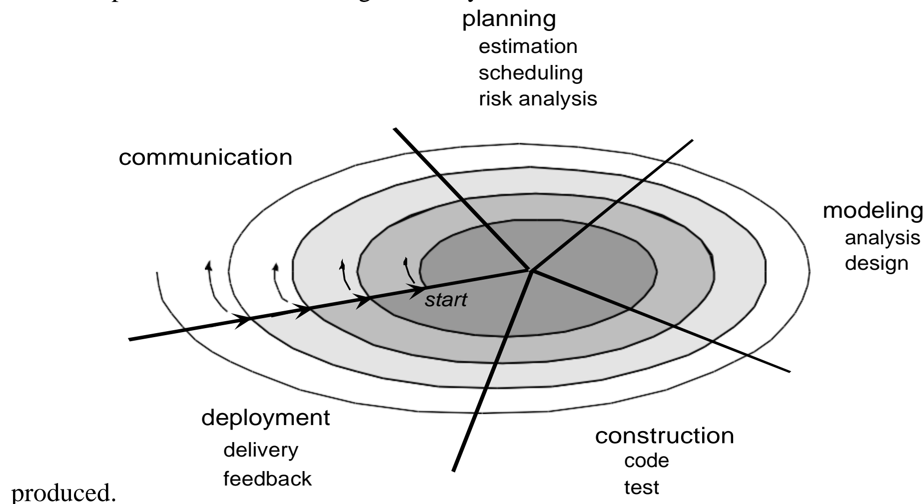
Prototyping can be **problematic** for the following reasons:

1. The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.
2. The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to

demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

THE SPIRAL MODEL

- The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- The spiral model can be adapted to apply throughout the entire life cycle of an application, from concept development to maintenance.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly morecomplete versions of the engineered system are



produced.

- **Anchor point milestones**- a combination of work products and conditions that are attained along the path of the spiral- are noted for each evolutionary pass.
- The first circuit around the spiral might result in the development of product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.
- Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.

➢ The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

➢ If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.

➢ Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

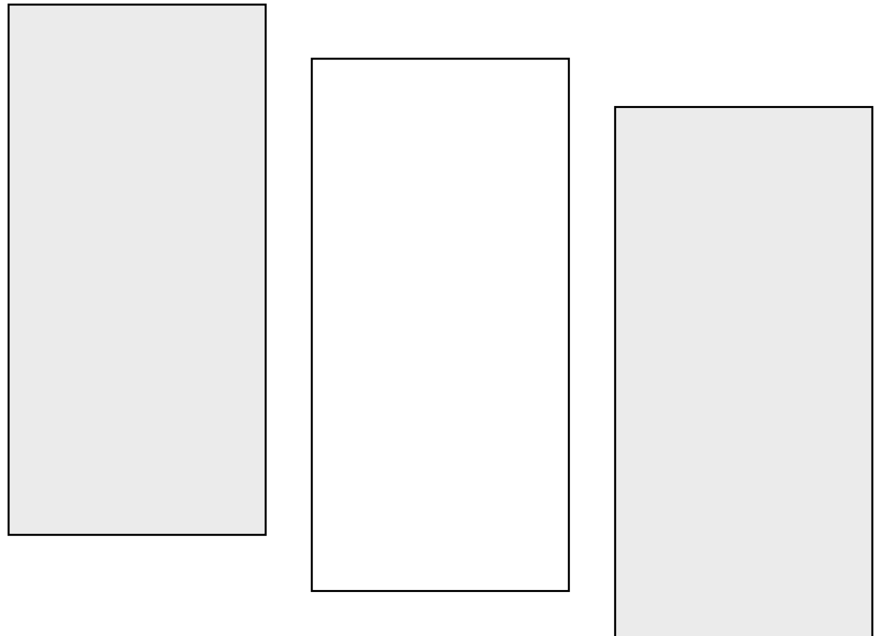
Advantages:

It provides the potential for rapid development of increasingly more complete versions of the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model uses prototyping as a risk reduction mechanism but, more importantly enables the developer to apply the prototyping approach at any stage in the evolution of the product.

Draw Backs:
The spiral model is not a panacea. It may be difficult to convince customers that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

.



UNIT-II

SOFTWARE REQUIREMENTS

Software requirements are necessary

- To introduce the concepts of user and system requirements
- To describe functional and non-functional requirements
- To explain how software requirements may be organised in a requirements document

What is a requirement?

- The requirements for the system are the description of the services provided by the system and its operational constraints
- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;

Both these statements may be called requirements

Requirements engineering:

- The process of finding out, analysing documenting and checking these services and constraints is called requirement engineering.
- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Requirements abstraction (Davis):

*If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements d**ocument** for the system."*

Types of requirement:

- **User requirements**
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **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.

Definitions and specifications:

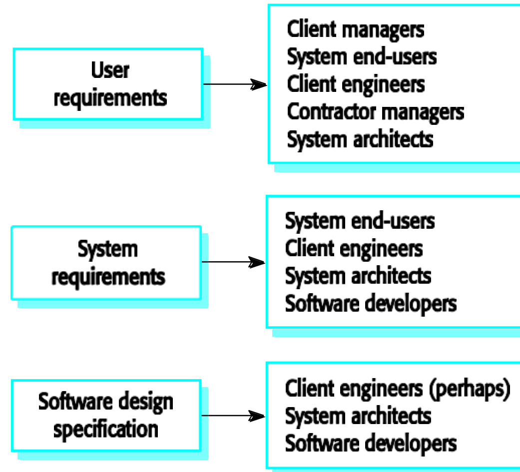
User Requirement Definition:

The software must provide the means of representing and accessing external files created by other tools.

System Requirement specification:

- The user should be provided with facilities to define the type of external files.
- Each external file type may have an associated tool which may be applied to the file.
- Each external file type may be represented as a specific icon on the user's display.
- Facilities should be provided for the icon representing an external file type to be defined by the user.
- When an user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

Requirements readers:



1) **Functional and non-functional requirements:**

Functional requirements

- Statements of services the system should provide how the system should react to particular inputs and how the system should behave in particular situations.

Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

Domain requirements

- Requirements that come from the application domain of the system and that reflect characteristics of that domain.

FUNCTIONAL REQUIREMENTS:

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

The functional requirements for **The LIBSYS system:**

- A library system that provides a single interface to a number of databases of articles in different libraries.
- Users can search for, download and print these articles for personal study.

Examples of functional requirements

- The user shall be able to search either all of the initial set of databases or select a subset from it.
- The system shall provide appropriate viewers for the user to read documents in the document store.

- Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

Requirements imprecision

- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term 'appropriate viewers'
  - User intention - special purpose viewer for each different document type;
  - Developer interpretation - Provide a text viewer that shows the contents of the document.

Requirements completeness and consistency:

In principle, requirements should be both complete and consistent.

Complete

- They should include descriptions of all facilities required.

Consistent

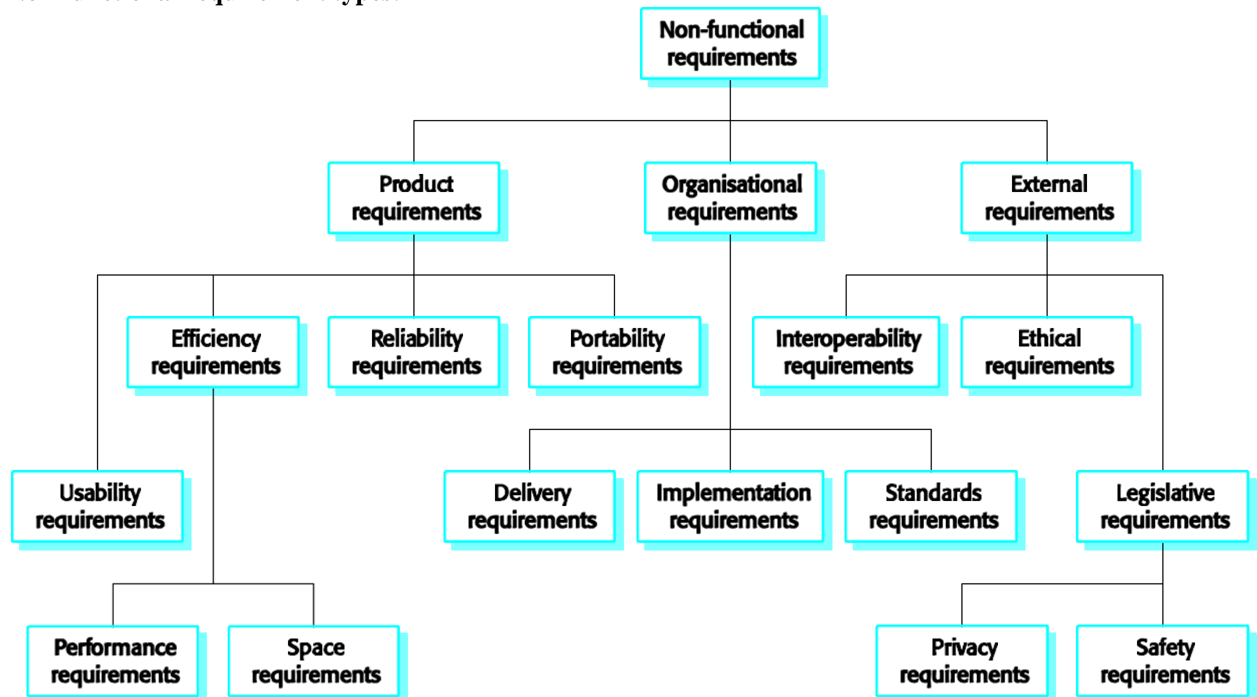
- There should be no conflicts or contradictions in the descriptions of the system facilities.

In practice, it is impossible to produce a complete and consistent requirements document.

NON-FUNCTIONAL REQUIREMENTS

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

Non-functional requirement types:



Non-functional requirements :

Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- *Eg:* The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

Goals and requirements:

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - A general intention of the user such as ease of use.
  - The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.
- Verifiable non-functional requirement
  - A statement using some measure that can be objectively tested.
  - Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.
- Goals are helpful to developers as they convey the intentions of the system users.

USER REQUIREMENTS

- Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.
- User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language

Lack of clarity
- Precision is difficult without making the document difficult to read.

Requirements confusion
- Functional and non-functional requirements tend to be mixed-up.

Requirements amalgamation
- Several different requirements may be expressed together

SYSTEM REQUIREMENTS

- More detailed specifications of system functions, services and c r requirements.
- They are intended to be a basis for designing the system.
- They may be incorporated into the system contract.
- System requirements may be defined or illustrated using system models

System requirement specification using a standard form:

1. Function
2. Description
3. Inputs
4. Source
5. Outputs
6. Destination
7. Action
8. Requires
9. Pre-condition
10. Post-condition
11. Side-effects

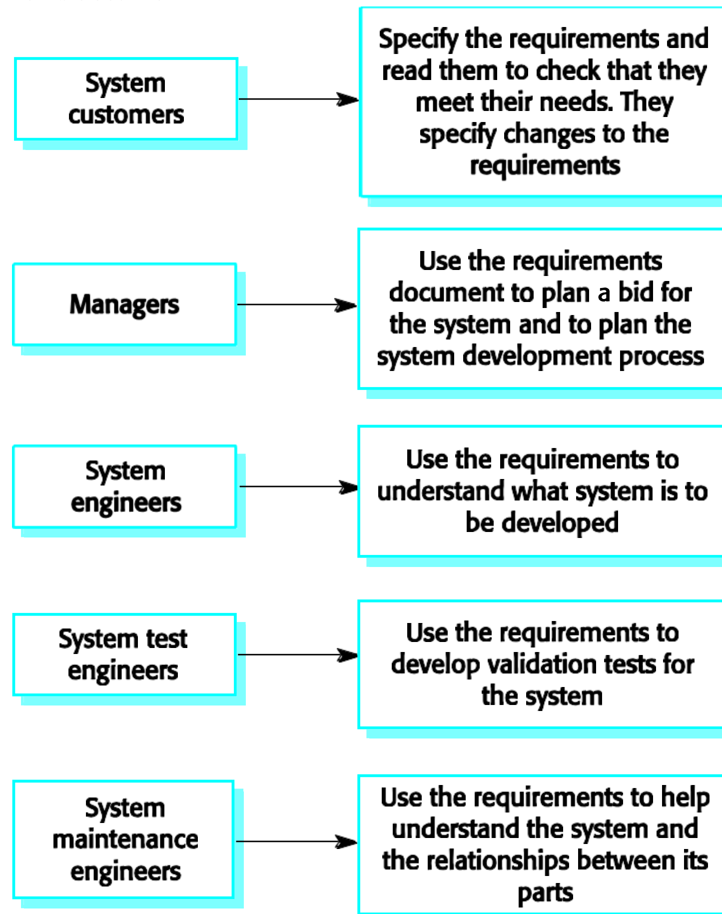
2) INTERFACE SPECIFICATION

- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- Three types of interface may have to be defined
  - ***Procedural interfaces*** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Applicatin Programming Interfaces (APIs)
  - ***Data structures that are exchanged*** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description
  - ***Data representations*** that have been established for an existing sub-system
- Formal notations are an effective technique for interface specification.

3) THE SOFTWARE REQUIREMENTS DOCUMENT:

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Users of a requirements document:



IEEE requirements standard defines a generic structure for a requirements document that must be instantiated for each specific system.

1. Introduction.
   i) Purpose of the requirements document
   ii) Scope of the project
   iii) Definitions, acronyms and abbreviations
   iv) References
   v) Overview of the remainder of the document
2. General description.
   i) Product perspective
   ii) Product functions
   iii) User characteristics
   iv) General constraints
   v) Assumptions and dependencies
3. Specific requirements cover functional, non-functional and interface requirements. The requirements may document external interfaces, describe system functionality and performance, specify logical database requirements, design constraints, emergent system properties and quality characteristics.
4. Appendices.
5. Index.

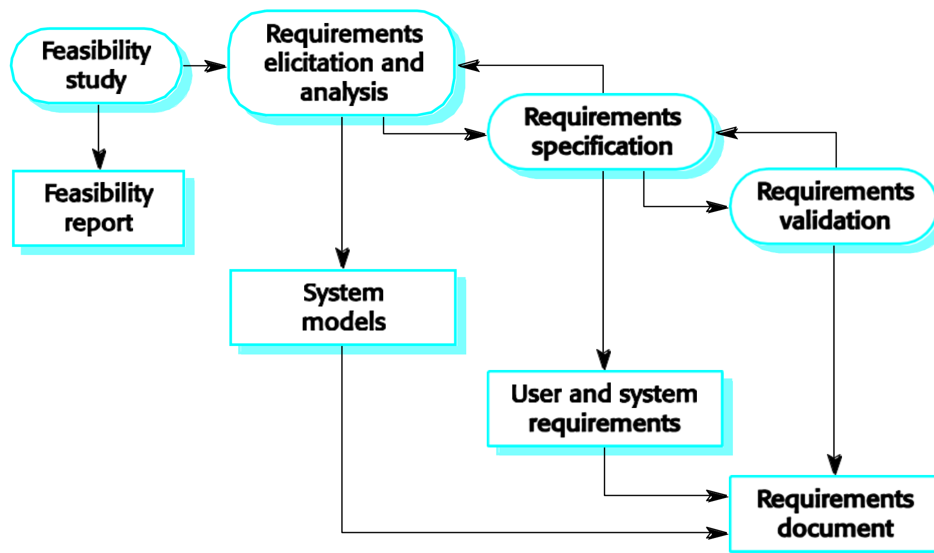
REQUIREMENTS ENGINEERING PROCESSES

The **goal** of requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirement engineering sub-processes. These are concerned with

- ✓ Assessing whether the system is useful to the business(feasibility study)
- ✓ Discovering requirements(elicitation and analysis)
- ✓ Converting these requirements into some standard form(specification)
- ✓ Checking that the requirements actually define the system that the customer wants(validation)

The process of managing the changes in the requirements is called **requirement management**.

The requirements engineering process

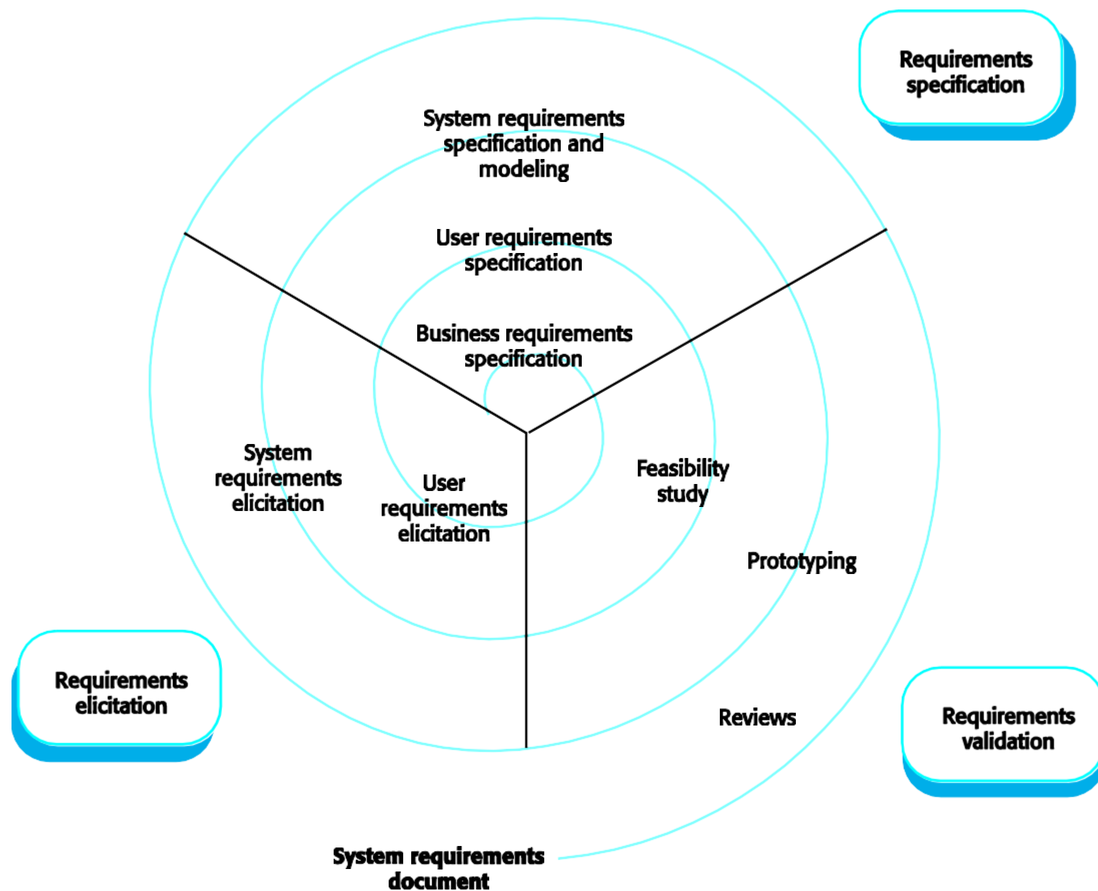


Requirements Engineering: The alternative perspective on the requirements engineering process presents the process as a **three-stage activity** where the activities are organized as an iterative process around a spiral. The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements and the user requirements. Later in the process, in the outer rings of the spiral, more effort will be devoted to system requirements engineering and system modeling.

This spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.

Some people consider requirements engineering to be the process of applying a structured analysis method such as object-oriented analysis. This involves analyzing the system and developing a set of graphical system models, such as use-case models, that then serve as a system specification. The set of models describes the behavior of the system and are annotated with additional information describing, for example, its required performance or reliability.

Spiral model of requirements engineering processes



1) **FEASIBILITY STUDIES**

A feasibility study decides whether or not the proposed system is worthwhile. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it worth carrying on with the requirements engineering and system development process.

- A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be engineered using current technology and within budget;

– If the system can be integrated with other systems that are used.

Feasibility study implementation:

- A feasibility study involves information assessment, information collection and report writing.
- Questions for people in the organisation
  - What if the system wasn't implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed? What skills?
  - What facilities must be supported by the proposed system?

In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system. They should try to complete a feasibility study in two or three weeks.

Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

2) REQUIREMENT ELICITATION AND ANALYSIS:

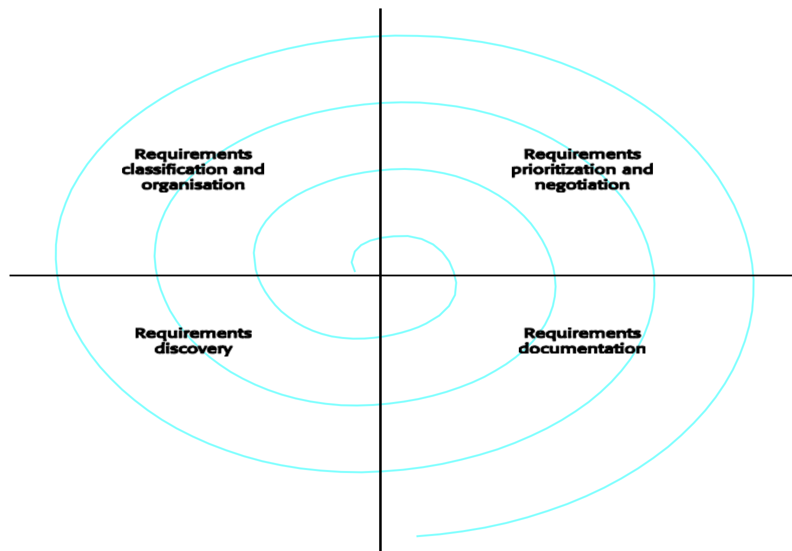
The requirement engineering process is requirements elicitation and analysis.

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*
-

Problems of requirements analysis

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

The requirements spiral



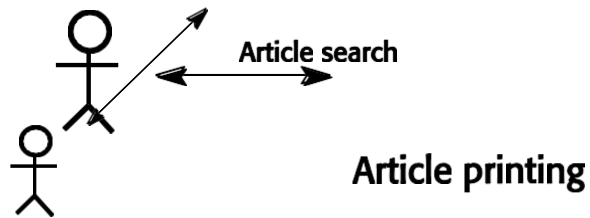
Process activities

1. Requirements discovery
   - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
2. Requirements classification and organisation
   - Groups related requirements and organises them into coherent clusters.
3. Prioritisation and negotiation
   - Prioritising requirements and resolving requirements conflicts.
4. Requirements documentation
   - Requirements are documented and input into the next round of the spiral.

Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.
-

Article printing use-case



3) REQUIREMENTS VALIDATION

- Concerned with demonstrating that the requirements define the system that the customer really wants.

User

- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking:

- ***Validity***: Does the system provide the functions which best support the customer's needs?
- ***Consistency***: Are there any requirements conflicts?
- ***Completeness***: Are all functions required by the customer included?
- ***Realism***: Can the requirements be implemented given available budget and technology
- ***Verifiability***: Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirement.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews:

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks:

- ***Verifiability*****:** Is the requirement realistically testable?
- ***Comprehensibility*****:** Is the requirement properly understood?
- ***Traceability*****:** Is the origin of the requirement clearly stated?
- ***Adaptability*****:** Can the requirement be changed without a large impact on other requirements?

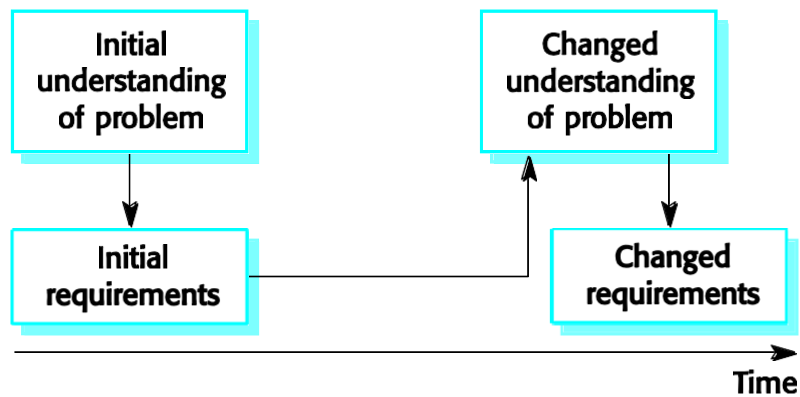
4) REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably incomplete and inconsistent
  - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.

Requirements change

- The priority of requirements from different viewpoints changes during the development process.
- System customers may specify requirements from a business perspective that conflict with end-user requirements.
- The business and technical environment of the system changes during its development.

Requirements evolution:



Requirements management planning:

- During the requirements engineering process, you have to plan:
  - Requirements identification
    - How requirements are individually identified;
  - A change management process
    - The process followed when analysing a requirements change;
  - Traceability policies
    - The amount of information about requirements relationships that is maintained;
  - CASE tool support
    - The tool support required to help manage requirements change;

Traceability:

Traceability is concerned with the relationships between requirements, their sources and the system design

- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability - Links from the requirements to the design;

CASE tool support:

- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management
  - Automated retrieval of the links between requirements.

Requirements change management:

- Should apply to all proposed changes to the requirements.
- Principal stages
  - Problem analysis. Discuss requirements problem and propose change;
  - Change analysis and costing. Assess effects of change on other requirements;
  - Change implementation. Modify requirements document and other documents to reflect change.

Change management:

SYSTEM MODELLING

- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.
- Different models present the system from different perspectives
  - Behavioural perspective showing the behaviour of the system;
  - Structural perspective showing the system or data architecture.

Model types

- Data processing model showing how the data is processed at different stages.
- Composition model showing how entities are composed of other entities.
- Architectural model showing principal sub-systems.
- Classification model showing how entities have common characteristics.
- Stimulus/response model showing the system's reaction to events.

1) CONTEXT MODELS:

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

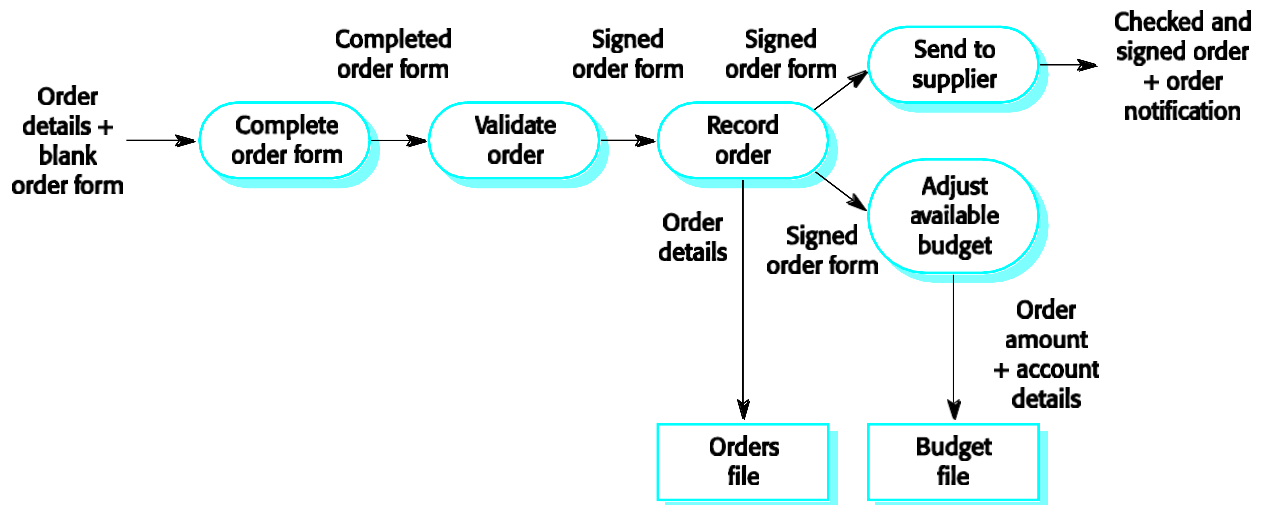
2) BEHAVIOURAL MODELS:

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
  - Data processing models that show how data is processed as it moves through the system;
  - State machine models that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system's behaviour.

Data-processing models:

- Data flow diagrams (DFDs) may be used to model the system's data processing.
- These show the processing steps as data flows through a system.
- DFDs are an intrinsic part of many analysis methods.
- Simple and intuitive notation that customers can understand.
- Show end-to-end processing of data.

Order processing DFD:



Data flow diagrams:

- DFDs model the system from a functional perspective.
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.
- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

Data dictionaries

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
    - Support name management and avoid duplication;
    - Store of organisational knowledge linking analysis, design and implementation;
- Many CASE workbenches support data dictionaries.

3) OBJECT MODELS:

- Object models describe the system in terms of object classes and their associations.
- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.
- Various object models may be produced
    - Inheritance models;
    - Aggregation models;
    - Interaction models.
- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

Inheritance models:

- Organise the domain object classes into a hierarchy.
- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes. these may then be specialised as necessary.

- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

Object models and the UML:

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
    - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
    - Relationships between object classes (known as associations) are shown as lines linking objects;
    - Inheritance is referred to as generalisation and is shown 'upwards' rather than 'downwards' in a hierarchy.

UNIT-IV

A strategic Approach for Software testing

- Software Testing
  - One of the important phases of software development
  - Testing is the process of execution of a program with the intention of finding errors
  - Involves 40% of total project cost
  - Testing Strategy
  - A road map that incorporates test planning, test case design, test execution and resultant data collection and execution
  - **Validation** refers to a different set of activities that ensures that the software is traceable to the customer requirements.
  - V&V encompasses a wide array of Software Quality Assurance
  - Perform Formal Technical reviews(FTR) to uncover errors during software development
  - Begin testing at component level and move outward to integration of entire component based system.
  - Adopt testing techniques relevant to stages of testing
  - Testing can be done by software developer and independent testing group
  - Testing and debugging are different activities. Debugging follows testing
  - Low level tests verifies small code segments.
  - High level tests validate major system functions against customer requirements

Software Testing :

- Two major categories of software testing
  Black box testing
  White box testing

Black box testing

Treats the system as black box whose behavior can be determined by studying its input and related output
Not concerned with the internal structure of the program

Black Box Testing

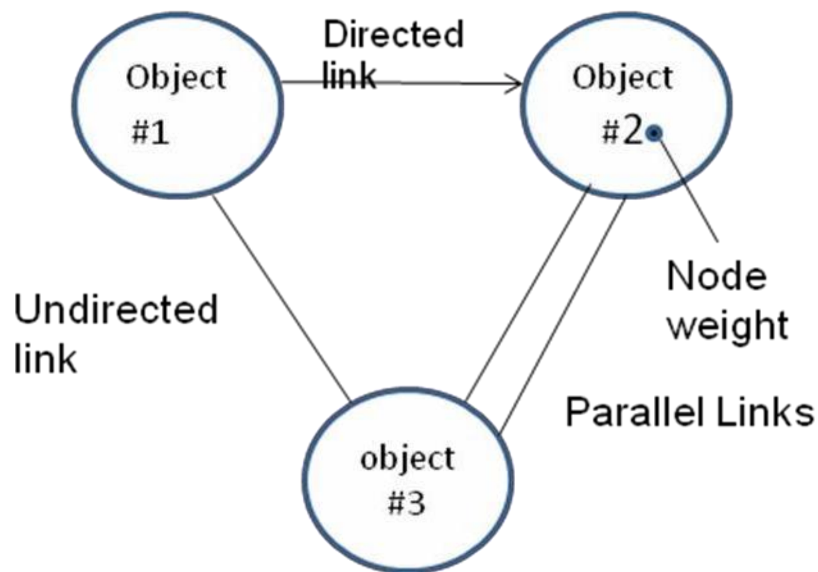
- It focuses on the functional requirements of the software ie it enables the sw engineer to derive a set of input conditions that fully exercise all the functional requirements for that program.
- Concerned with functionality and implementation

1)Graph based testing method

2)Equivalence partitioning

Graph based testing

- Draw a graph of objects and relations
- Devise test cases t uncover the graph such that each object and its relationship exercised.



Equivalence partitioning

- Divides all possible inputs into classes such that there are a finite equivalence classes.
- Equivalence class

-- Set of objects that can be linked by relationship

- Reduces the cost of testing
- Example
- Input consists of 1 to 10
- Then classes are n<1,1<=n<=10,n>10

Boundary Value analysis

- Select input from equivalence classes such that the input lies at the edge of the equivalence classes
- Set of data lies on the edge or boundary of a class of input data or generates the data that lies at the boundary of a class of output data

White Box testing

- Also called glass box testing
- Involves knowing the internal working of a program
- Guarantees that all independent paths will be exercised at least once.
- Exercises all logical decisions on their true and false sides
- Executes all loops
- Exercises all data structures for their validity
- White box testing techniques

1. Basis path testing
2. Control structure testing

Basis path testing

- Proposed by Tom McCabe
- Defines a basic set of execution paths based on logical complexity of a procedural design
- Guarantees to execute every statement in the program at least once
- Steps of Basis Path Testing
- Draw the flow graph from flow chart of the program
- Calculate the cyclomatic complexity of the resultant flow graph
- Prepare test cases that will force execution of each path
- Three methods to compute Cyclomatic complexity number
- V(G)=E-N+2(E is number of edges, N is number of nodes
- V(G)=Number of regions
- V(G)= Number of predicates +1
- Control Structure testing

- Basis path testing is simple and effective
- It is not sufficient in itself
- Control structure broadens the basic test coverage and improves the quality of white box testing
- Condition Testing
- Data flow Testing
- Loop Testing

Condition Testing

- Exercise the logical conditions contained in a program module
- Focuses on testing each condition in the program to ensure that it does contain errors
- Simple condition

Data flow Testing

- Selects test paths according to the locations of definitions and use of variables in a program
- Aims to ensure that the definitions of variables and subsequent use is tested
- First construct a definition-use graph from the control flow of a program

Loop Testing

- Focuses on the validity of loop constructs
- Four categories can be defined

1. Simple loops
2. Nested loops
3. Concatenated loops
4. Unstructured loop.

Software Quality

- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- Factors that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects uncovered during testing)
2. Factors that can be measured only indirectly (e.g. usability or maintainability)

- McCall's quality factors

1. Product operation
    a. Correctness
    b. Reliability
    c. Efficiency
    d. Integrity
    e. Usability
2. Product Revision
    a. Maintainability
    b. Flexibility
    c. Testability

3. Product Transition
    a. Portability
    b. Reusability
    c. Interoperability

ISO 9126 Quality Factors

1.Functionality

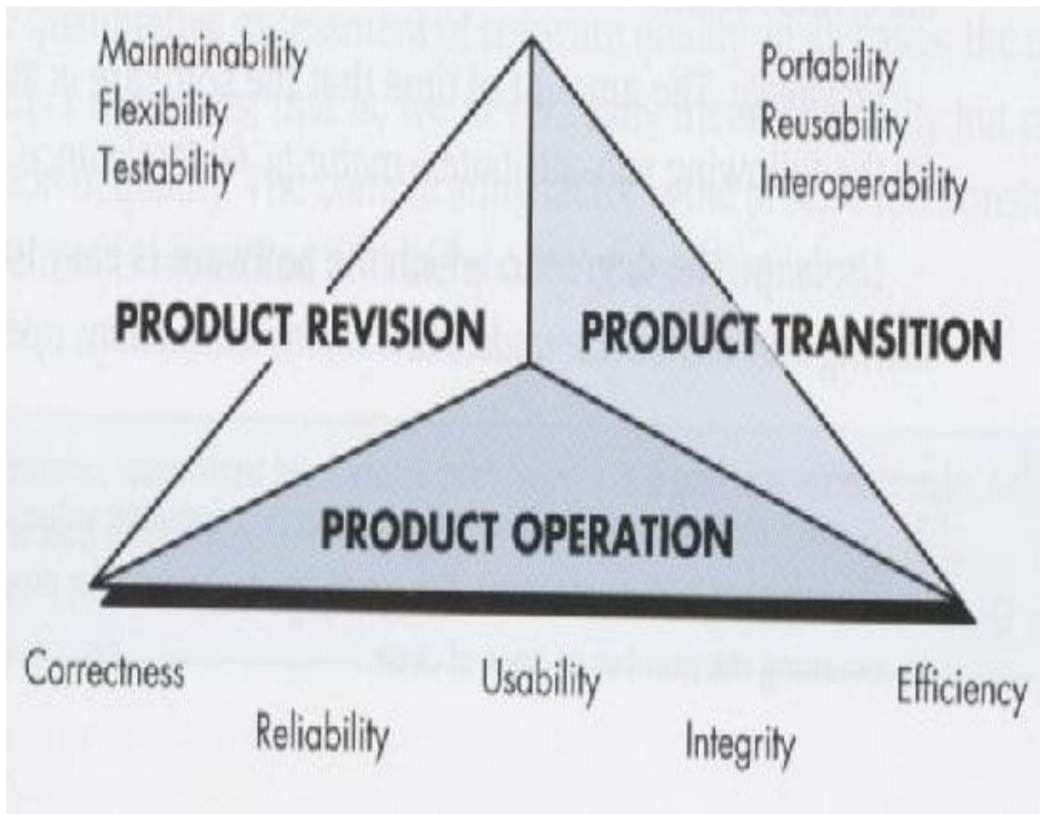
2.Reliability

3.Usability

4.Efficiency

5.Maintainability

6.Portability



Product metrics

- Product metrics for computer software helps us to assess quality.
- Measure

-- Provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process

- Metric(IEEE 93 definition)

-- A quantitative measure of the degree to which a system, component or process possess a given attribute

- Indicator

-- A metric or a combination of metrics that provide insight into the software process, a software project or a product itself

Product Metrics for analysis,Design,Test and maintenance

- **<u>Product metrics for the Analysis model</u>**

❖ Function point Metric

▪ First proposed by Albrecht
▪ Measures the functionality delivered by the system
▪ FP computed from the following parameters

1) Number of external inputs(EIS)
2) Number external outputs(EOS)
3) Number of external Inquiries(EQS)
4) Number of Internal Logical Files(ILF)
5) Number of external interface files(EIFS)

Each parameter is classified as simple, average or complex and weights are assigned as follows

Count	Simple	avg	Complex
	3	4	6
	4	5	7
	3	4	6
	7	10	15
	5	7	10

FP=Count total *[0.65+0.01*E(Fi)]

METRICS FOR PROCESS AND PROJECTS

1) SOFTWARE MEASUREMENT

Software measurement can be categorized in two ways.

(1) ***Direct measures*** of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced.

To develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- .

.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the *function point.* Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

2) METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

- **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- **Maintainability.** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is *mean-time-tochange* (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
- **Integrity.** Attacks can be made on all three components of software: programs, data, and documents.

QUALITY MANAGEMENT

1) QUALITY CONCEPTS:

Quality management encompasses
(1) a quality management approach,
(2) effective software engineering technology (methods and tools),
(3) formal technical reviews that are applied throughout the software process,
(4) a multitiered testing strategy,
(5) control of software documentation and the changes made to it,
(6) a procedure to ensure compliance with software development standards (when applicable), and
(7) measurement and reporting mechanisms.

Variation control is the heart of quality control.

Quality

- ✓ The *American Heritage Dictionary* defines *quality* as "a characteristic or attribute of something."
- ✓ ***Quality of design*** refers to the characteristics that designers specify for an item.
- ✓ ***Quality of conformance*** is the degree to which the design specifications are followed during manufacturing.

In software development, quality of design encompasses requirements, specifications, and the design of the system. Quality of conformance is an issue focused primarily on implementation. If the implementation follows the design and the resulting system meets its requirements and performance goals, conformance quality is high.
Robert Glass argues that a more "intuitive" relationship is in order:

User satisfaction = compliant product + good quality + delivery within budget and schedule

Quality Control

Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects produced.

Quality Assurance

Quality assurance consists of the auditing and reporting functions that assess the effectiveness and completeness of quality control activities. The **goal of quality** assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

Cost of Quality

The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities.

Quality costs may be divided into costs associated with prevention, appraisal, and failure.
Prevention costs include

- quality planning
- formal technical reviews
- test equipment
- training

Appraisal costs include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include

- in-process and interprocess inspection
- equipment calibration and maintenance
- testing

Failure costs are those that would disappear if no defects appeared before shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs.

Internal failure costs are incurred when we detect a defect in our product prior to shipment. Internal failure costs include

- rework
- repair
- failure mode analysis

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are

- complaint resolution
- product return and replacement
- help line support
- warranty work

2) SOFTWARE QUALITY ASSURANCE

Software quality is defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

The definition serves to emphasize three important points:

1) Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2) Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3) A set of implicit requirements often goes unmentioned (e.g., the desire for ease of use and good maintainability). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

SQA Activities

Software quality assurance is composed of a variety of tasks associated with two different constituencies—

- ✓ the software engineers who do technical work and
- ✓ an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

The Software Engineering Institute recommends a set of SQA activities that address quality assurance planning, oversight, record keeping, analysis, and reporting. These activities are performed (or facilitated) by an independent SQA group that conducts the following activities.

Prepares an SQA plan for a project. The plan is developed during project planning and is reviewed by all interested parties. Quality assurance activities performed by the software engineering team and the SQA group are governed by the plan. The plan identifies

- ✓ evaluations to be performed
- ✓ audits and reviews to be performed
- ✓ standards that are applicable to the project
- ✓ procedures for error reporting and tracking
- ✓ documents to be produced by the SQA group
- ✓ amount of feedback provided to the software project team

3) SOFTWARE REVIEWS

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.*

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review

A formal technical review is the most effective filter from a quality assurance standpoint. Conducted by software engineers (and others) for software engineers, the FTR is an effective means for improving software quality.

Cost Impact of Software Defects:

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

A number of industry studies indicate that design activities introduce between 50 and 65 percent of all errors during the software process. However, formal review techniques have been shown to be up to 75 percent effective] in uncovering design errors. By detecting and removing a large percentage of these errors, the review process substantially reduces the cost of subsequent steps in the development and support phases.

To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects Assume that an error uncovered

- ✓ during design will cost 1.0 monetary unit to correct.
- ✓ just before testing commences will cost 6.5 units;

- ✓ during testing, 15 units;
- ✓ and after release, between 60 and 100 units.

4) FORMAL TECHNICAL REVIEWS

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are

(1) to uncover errors in function, logic, or implementation for any representation of the software;
(2) to verifythat the software under review meets its requirements;
(3) to ensure that the software has been represented according to predefined standards;
(4) to achieve software that is developed in a uniform manner; and
(5) to make projects more manageable.

The Review Meeting

Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

The focus of the FTR is on a work product.
The individual who has developed the work product—the *producer*—informs the project leader that the work product is complete and that a review is required.

- ✓ The project leader contacts a *review leader,* who evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation.
- ✓ Each reviewer is expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- ✓ The review meeting is attended by the review leader, all reviewers, and the producer. One of the reviewers takes on the role of the *recorder;* that is, the individual who records (in writing) all important issues raised during the review.

At the end of the review, all attendees of the FTR must decide whether to
(1) accept the product without further modification,
(2) reject the product due to severe errors (once corrected, another review must be performed), or
(3) accept the product provisionally.
The decision made, all FTR attendees complete a sign-off, indicating their participation in the review and their concurrence with the review team's findings.

Review Reporting and Record Keeping

At the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A ***review summary report*** answers three questions:
1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?
The review summary report is a single page form.
It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

Review Guidelines

The following represents a minimum set of guidelines for formal technical reviews:

1. ***Review the product, not the producer***. An FTR involves people and egos. Conducted properly, the FTR should leave all participants with a warm feeling of accomplishment.
2. ***Set an agenda and maintain it***. An FTR must be kept on track and on schedule. The review leader is chartered with the responsibility for maintaining the meeting schedule and should not be afraid to nudge people when drift sets in.

3. ***Limit debate and rebuttal***. When an issue is raised by a reviewer, there may not be universal agreement on its impact.
4. ***Enunciate problem areas, but don't attempt to solve every problem noted***. A review is not a problem-solving session. The solution of a problem can often be accomplished by the producer alone or with the help of only one other individual. Problem solving should be postponed until after the review meeting.
5. ***Take written notes***. It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. ***Limit the number of participants and insist upon advance preparation***. Keep the number of people involved to the necessary minimum.
7. ***Develop a checklist for each product that is likely to be reviewed***. A checklist helps the review leader to structure the FTR meeting and helps each reviewer to focus on important issues. Checklists should be developed for analysis, design, code, and even test documents.
8. ***Allocate resources and schedule time for FTRs***. For reviews to be effective, they should be scheduled as a task during the software engineering process
9. ***Conduct meaningful training for all reviewers.*** To be effective all review participants should receive some formal training.
10. ***Review your early reviews***. Debriefing can be beneficial in uncovering problems with the review process itself.

Sample-Driven Reviews (SDRs):

SDRs attempt to quantify those work products that are primary targets for full FTRs.To accomplish this the following steps are suggested...

- Inspect a fraction a_i of each software work product, *i.* Record the number of faults, f_i found within a_i.
- Develop a gross estimate of the number of faults within work product *i* by multiplying f_i by $1/a_i$.
- Sort the work products in descending order according to the gross estimate of the number of faults in each.
- Focus available review resources on those work products that have the highest estimated number of faults.

The fraction of the work product that is sampled must

- Be representative of the work product as a whole and
- Large enough to be meaningful to the reviewer(s) who does the sampling.

5) STATISTICAL SOFTWARE QUALITY ASSURANCE

For software, statistical quality assurance implies the following steps:

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., non-conformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the For software, statistical quality assurance implies the following steps:

.

The application of the statistical SQA and the pareto principle can be summarized in a single sentence: *spend your time focusing on things that really matter, but first be sure that you understand what really matters*.

Six Sigma for software Engineering:

Six Sigma is the most widely used strategy for statistical quality assurance in industry today.
The term “six sigma” is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- ***Define*** customer requirements and deliverables and project goals via well-defined methods of customer communication

- ***Measure*** the existing process and its output to determine current quality performance (collect defect metrics)
- ***Analyze*** defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps.

- ***Improve*** the process by eliminating the root causes of defects.
- ***Control*** the process to ensure that future work does not reintroduce the causes of defects

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If any organization is developing a software process (rather than improving and existing process), the core steps are augmented as follows:

- ***Design*** the process to
  - avoid the root causes of defects and
  - to meet customer requirements
- ***Verify*** that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design and verify) method.

6) THE ISO 9000 QUALITY STANDARDS

A *quality assurance system* may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management

ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines have been developed to help interpret the standard for use in the software process.

The requirements delineated by ISO 9001 address topics such as

- management responsibility,
- quality system, contract review,
- design control,
- document and data control,
- product identification and traceability,
- process control,
- inspection and testing,
- corrective and preventive action,
- control of quality records,
- internal quality audits,
- training,
- servicing and
- statistical techniques.

In order for a software organization to become registered to ISO 9001, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.

SOFTWARE RELIABILITY

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

7.1 Measures of Reliability and Availability:

Most hardware-related reliability models are predicated on failure due to wear rather than failure due to design defects. In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. In fact, all software failures can be traced to design or implementation problems; wear does not enter into the picture.

A simple measure of reliability is *meantime-between-failure* (MTBF), where

MTBF = MTTF + MTTR

The acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

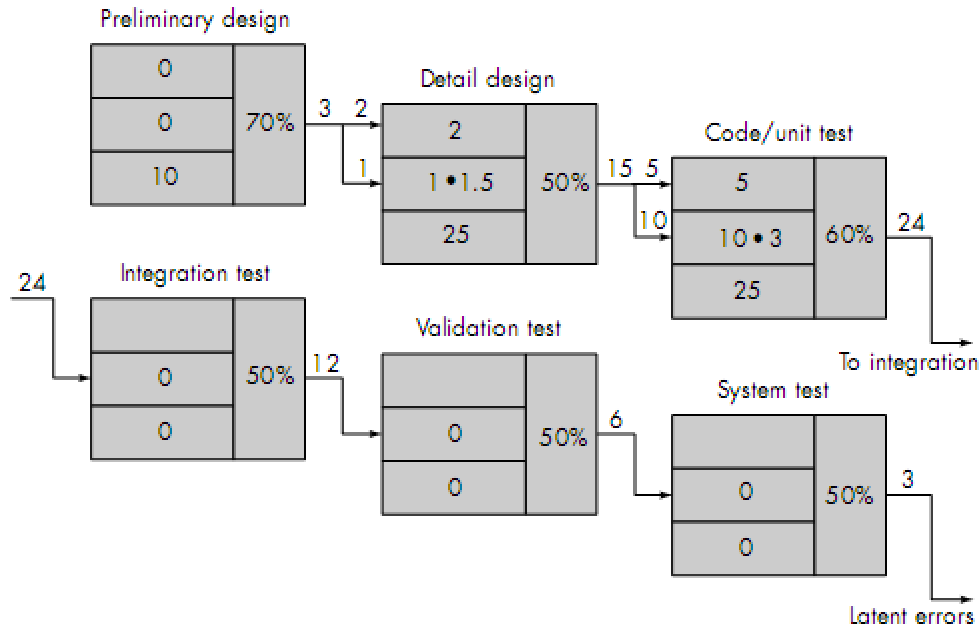
In addition to a reliability measure, we must develop a measure of availability. *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

Availability = [MTTF/(MTTF + MTTR)] 100%

The MTBF reliability measure is equally sensitive to MTTF and MTTR. The availability measure is somewhat more sensitive to MTTR, an indirect measure of the maintainability of software.

Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.



SOFTWARE ENGINEERING IMPORTANT QUESTIONS

Q1. Explain SDLC waterfall model? Also list its advantages and disadvantages?

Q2. Explain software requirement specification document in detail?

Q3. Difference between top-down and bottom-up design approaches?

Q4.Write short note on : information hiding,software verification?

Q5.What is software testing.explain different levels of software testing?

Q6.explain SEI CMM in detail?

Q7. Give the comparison of SEI CMM with ISO CMM?

Q8. Explain different software testing metrics?

Q9. Write short note on (a) CASE and its scope. (b) Rational software suite?

Q10.What is agile software development ? outline the agile process ?

Q11.What is requirement engineering?Why it is gaining importance?

Q12What is software crisis? Write its causes? Also discuss the solutions to overcome software crisis?

Q13What is software design? What are the principles of software design?

Q14.What is means by programming style?what are the important elements of programming style?

Q15.Difference between good and bad design?

Q16.What are the different software lifecycles models? Discuss the need of these models?

Q17.Difference between verification and validation?

Q18.Difference between PSP and TSP?

Q19. Explain flow and behavioural modelling?

Q20. Explain architectural design model?

Q22.Explain build and fix model?

Q24.Explain different software applications ?

Q25.Explain White box and black box testing?

Q26.What is debugging?

Q27. Explain different software testing strategies?

Q28. Explain structural testing?

ANSWERS

Q25.

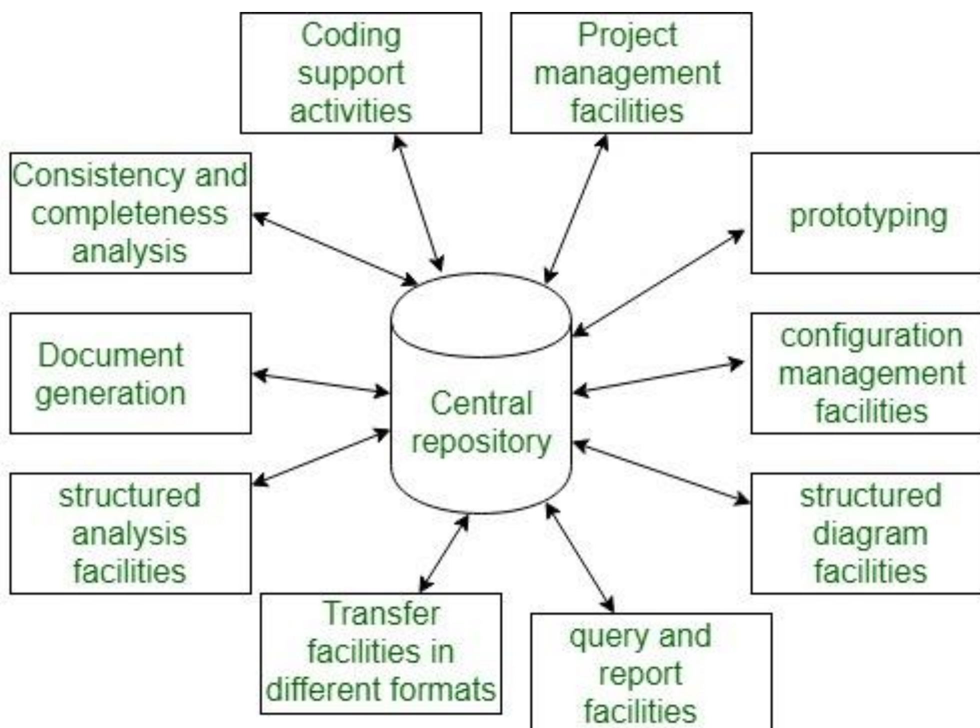
Criteria	**Black Box Testing**	**White Box Testing**
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
Levels Applicable To	Mainly applicable to higher levels of testing:Acceptance Testing System Testing	Mainly applicable to lower levels of testing:Unit Testing Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design

Q9. A CASE (Computer power-assisted software package Engineering) tool could be a generic term accustomed denote any type of machine-driven support for software package engineering. in a very additional restrictive sense, a CASE tool suggests that any tool accustomed automatize some activity related to software package development.
Several CASE tools square measure obtainable. A number of these CASE tools assist in part connected tasks like specification, structured analysis, design, coding, testing, etc. and other to non-phase activities like project management and configuration management.

Reasons for using CASE tools:
The primary reasons for employing a CASE tool are:

- to extend productivity
- to assist turn out higher quality code at a lower price

CASE environment:
Although individual CASE tools square measure helpful, the true power of a toolset is often completed only this set of tools square measure integrated into a typical framework or setting. CASE tools square measure characterized by the stage or stages of package development life cycle that they focus on. Since totally different tools covering different stages share common data, it's needed that they integrate through some central repository to possess an even read of data related to the package development artifacts. This central repository is sometimes information lexicon containing the definition of all composite and elementary data things.



A CASE environment

Q6. The Software Engineering Institute (**SEI**) **Capability Maturity Model** (**CMM**) specifies an increasing series of levels of a software development organization. The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.

What is CMM ?

- CMM stands for Capability Maturity Model.

- Focuses on elements of essential practices and processes from various bodies of knowledge.
- Describes common sense, efficient, proven ways of doing business (which you should already be doing) - not a radical new approach.
- CMM is a method to evaluate and measure the maturity of the software development process of an organizations.
- CMM measures the maturity of the software development process on a scale of 1 to 5.
- CMM v1.0 was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in Pittsburgh, USA.
- CMM was originally developed for Software Development and Maintenance but later it was developed for :
    - Systems Engineering
    - Supplier Sourcing
    - Integrated Product and Process Development
    - People CMM
    - Software Acquisition
    - CMM Examples:
- People CMM: Develop, motivate and retain project talent.
- Software CMM: Enhance a software focused development and maintenance capability.

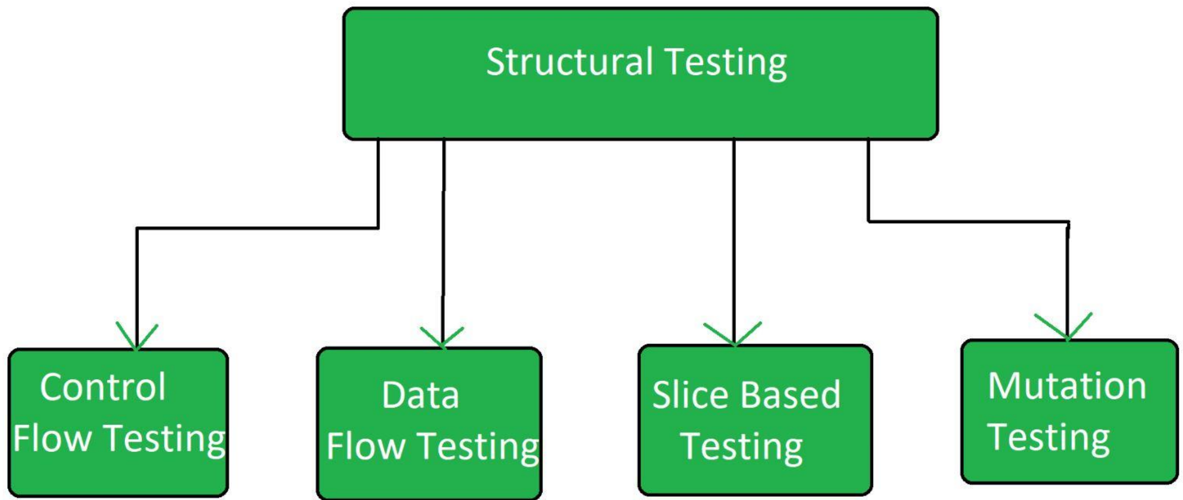
Q28. **Structural testing** is a type of software testing which uses the internal design of the software for testing or in other words the software testing which is performed by the team which knows the development phase of the software, is known as structural testing. Structural testing is basically related to the internal design and implementation of the software i.e. it involves the development team members in the testing team. It basically tests different aspects of the software according to its types. Structural testing is just the opposite of behavioral testing.

Types of Structural Testing:

There are 4 types of Structural Testing:

1. Control flow testing.
2. Data flow testing
3. Mutation testing
4. Slice based testing.

Diagram of structural testing


Structural Testing
Control Flow Testing
Data Flow Testing
Slice Based Testing
Mutation Testing