- : Introduction to Software Engineering: -

- **What is Software Engineering:**

In most of the software development organizations, there is no systematic methodology for the development. Software development is considered as coding, through coding forms a very small percentage of the total development effort. Before starting the coding, one has to systematically approach the problem. One has to understand the requirements (what the software is supposed to do), carry out the design, do the coding, carry out a rigorous (precise) testing and if the software is as per the requirements, release the software to the customer. Subsequently, if the customer wants some changes refinements or enhancements – then the software has to be modified.

"Software Engineering can be defined as systematic development of software."

IEEE (Institute of Electrical and Electronics Engineering) software engineering as "The application of a systematic, disciplined, quantitative approach to the development, operation and maintenance of software".

The software engineering discusses systematic and cost – effective techniques to software development. These techniques have resulted from innovations as well as lessons learnt from past mistakes. Alternatively, we can view software engineering as the engineering approach to develop software.

Software engineering is an **engineering discipline** that is concerned with all aspects of **software production** from the early stages of system specification to maintaining the system after it has gone into use.

Suppose you want to build a large commercial complex. If failure might come in several forms – the building might collapse during the construction stage itself due to his ignorance of the theories concerning the strengths of materials; the construction might get delayed, since he may not prepare proper estimations and detailed plans regarding the types and quantities of raw materials required, the times at which these are required, etc. In short, to be successful in constructing a building of large magnitude, one needs a through understanding of civil and architectural engineering techniques such as analysis, estimations, prototyping, planning, designing and testing, etc. Similar things happen in case of software development projects as well.

- **why software engineering**

1. To deliver quality product
2. The competition in thee software industry.

3. The development cost
4. Time to market
5. Fast changing technology
6. Sensitivity in software products.

- **History of Software Engineering:**

The establishment and use of software engineering principals in order to obtain economically software, that is reliable and works effectively on real machines (Prof. Fritz Bouer) at the NATO conference on Software Engineering in 1969.

- **Goals of Software Engineering:**

1. **Usability:**
   The usability for the end user to easily & effectively put the software to proper use.

2. **Efficiency:**
   The ability for software to use computing resources effectively (space and time).

3. **Maintainability**
   The ability to easily make changes enhancements or implements.

- **Principals of Software Engineering:**

1. **Encapsulation:**
   Hide the implementation.

2. **Modularity:**
   Divide and conquer data of complex program.

3. **Localization:**
   Collect similar things together.

4. **Abstraction:**
   Provide an Illusion.

5. **Uniformity:**
   Make everything look similar.

Software Crisis:

The expenses that organization all around the world are running on software purchases as compare to the hardware. At present, many customers are actually spending much more buying software than on buying hardware. There are many factors to software crisis. The first one is lack of training in software engineering, increasing skill shortage, lack of knowledge of software engineering. Software crisis can be classified in the following measure areas.

- ✓ **From Programmers point of view:**
  - Problem of portability
  - Problem of documentation
  - Problem of co-ordination of work with different people
  - Problem that arise during actual runtime in organization some time the errors are not detected during the run of program.
  - Problem of piracy of software
  - Problem of Implementation
  - Problem of maintenance in proper manner.

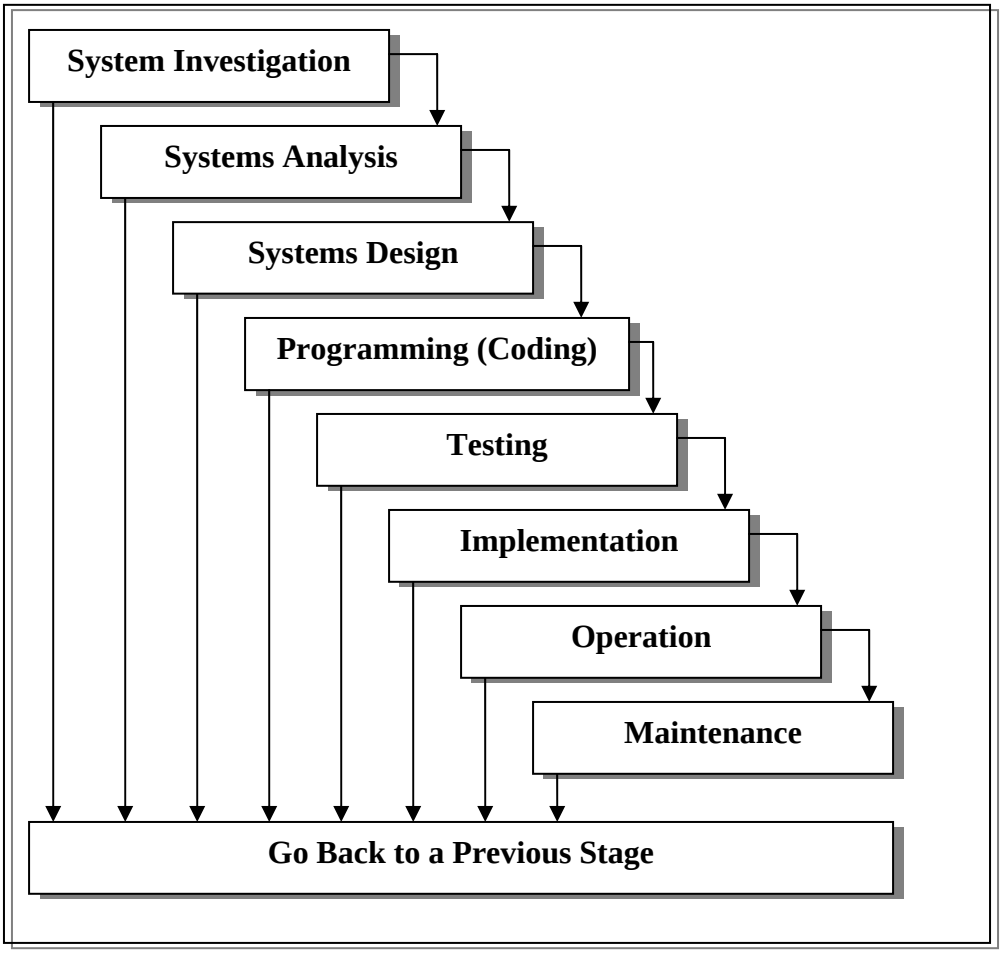
- ✓ **From User's point of view:**
  - How to choose a software from market availability
  - Hardware specification
  - Problem of virus
  - Problem of software bugs
  - Certain software runs only on specific Operating System environment.
  - Problem of disk managements
  - Problems for protected data in software
  - Problem of different versions of software.

- : Software Development Process Models: -

Software Development Life Cycle:

Large project structured problems, such as accounting and payroll systems and enterprise software applications, are usually conceived, planned, development and maintained within a framework called systems development life cycle (SDLC). Thus we can say that systems development life cycle (SDLC) is a structured framework used for large IT projects that consists of sequential processes by which information systems are developed.

This approach is also called "waterfall" development, because it consists of several distinct phases that are followed methodically, and the developers complete the phases sequentially. SDLC is a classical though of as the set of activities that analyst, designers and users carry out to develop and implement an information system. SDLC consist of eight development stages.



1. <u>System Investigation:</u>

Systems investigation begins with the business problem or business opportunity. In this phase, investigation is carried out regarding the need to develop a new system or what are the modification that are required to modify the old system. The next task in the systems investigation stage is the feasibility study that judges the probability of success of a proposed project and provides rough assessment of the project's feasibility. Feasibility study consists of four different aspects they are:

- **Technical Feasibility:**

  Assessment of whether hardware, software and communications components can develop and / or acquired to solve a business problem.

- **Economic Feasibility:**
  Assessment of whether a project is an affordable by organization? And what is the time needed to complete it.

- **Operational Feasibility:**
  Assessment of the system whether, the system be used appropriately by its user and will the system be used to its full capacity?

- **Behavioral Feasibility:**
  Assessment of the human issues involved in a proposed project, including resistance to change and skills and training needs.

2. Systems analysis:

This stage examines the business problem that the organization plans to solve with an information system. This stage defines the business problem, identifies its causes, specifies the solution and identifies the information requirements that the solution must satisfy. Organizations have three basic solutions to any business problem relating to an information system:

- Do nothing and continue to use the existing system unchanged,
- Modify or enhance the existing system,
- Develop a new system

The analysis is carried out by system analyst. For there the system analyst has to work with variety of persons to gather data about business process. As the data are gathered, analyst studies the requirement that identifies features of the new system. After analysis is completed, system analysis produces the following information:

- Strengths and weaknesses of the existing system.
- Functions that the new system must have to solve the business problem.
- User information requirements for the new system.

3. Systems design:

System analysis describes what a system must do to solve the business problem and systems design describes how the system will accomplish this task. The deliverable of the systems design phase is the technical design that specific the following:

- System outputs, inputs and user interfaces.
- Hardware, software, databases, telecommunications, personnel and procedures
- How these components are integrated
  System design has two major aspects of the new system:

- Logical system design: states what the system will do?
- Physical system design: states now the system will perform its functions?

4. Programming (Coding):

The translation of a system's design specification into computer code is done in this phase. This process can be lengthy and time consuming. In order to achieve some uniformity to the programming process, programmers use structured programming techniques. These techniques improve logical flow of the program by decomposing the computer code into modules.

5. Testing:

Testing check to see if the computer code will produce the expected and desired results under certain conditions. Testing is designed to delete errors (bugs) in the computer code. These errors are of two types. Syntax errors (e.g. misspelled word or misplaces comma) and Logic errors that permit the program to run but result in incorrect output.

6. Implementation:

Implementation or deployment is the process of converting from the old system to the new system. Organizations use four major conversion strategies; parallel, direct, pilot and phased.

- **Parallel conversion:** Implementation process in which the old system and the new system operate simultaneously for a period of time**.**
- **Direct conversion:** Implementation process in which the old system is cut off and the new system turned on at a certain point in time.
- **Pilot conversion:** Implementation process that introduces the new system in one part of the organization on a trial basis, when new system is working property it is introduced in other parts of the organization.
- **Phased conversion:** Implementation process that introduces components of the new system in stages, until the entire new system is operational.

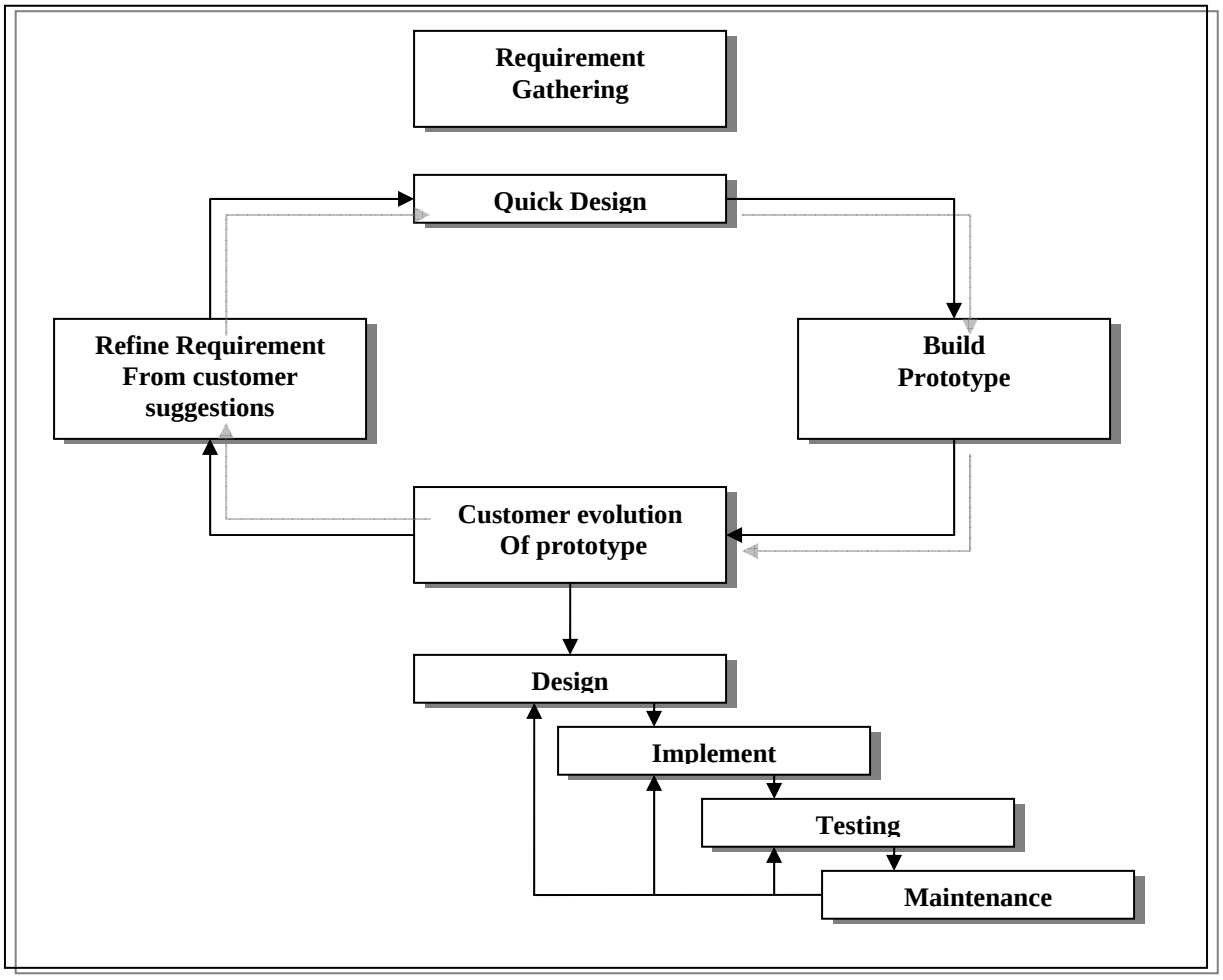
7. Operation and Maintenance:

Systems need several types of maintenance;

- Debugging: A process that continues throughout the life of the system.
- Updating: Updating the system to accommodate changes in business conditions
- Maintenance: That adds new functionality to the system – adding new features to the existing system without disturbing its operation.
- Corrective, Adaptive and Perfective

Prototype Model:

- The prototyping model requires that before caring out the development of the actual software, a working prototype of the system should be built. A prototype is a toy implementation of the system.
- A prototype is usually built using several short cuts. The short cuts might involve using in efficient, in accurate or dummy functions.
- The short cut implementation of a function may produce the desire result by using a table look up rather than by performing the actual computations.
- A prototype usually turned out to be a very complex version of the actual system. Possibly it's using limited functions capabilities, low reliability and inefficient performance as compare to the actual function of the software.
- In prototype model we have to develop Graphical User Interface (GUI) of a particular system for the user.
- It becomes much easier to form his opinion by experimental with a working model.
- Rather then trying to imaging a working of complex system.



Advantage of Prototype Model:

- It saves time in system development.
- Encourage the communication between the developer and user.
- Missing customer services can be detected.
- Confusing or difficult to use user services can be identified and refine**.**
- System can be delivering with in proper time & there for delays are minimize**.**
- If proper tools are used low cost is required for development of prototype.

Tools Available For Prototyping:

- Screen Generator
- Report Generator
- Application Generator

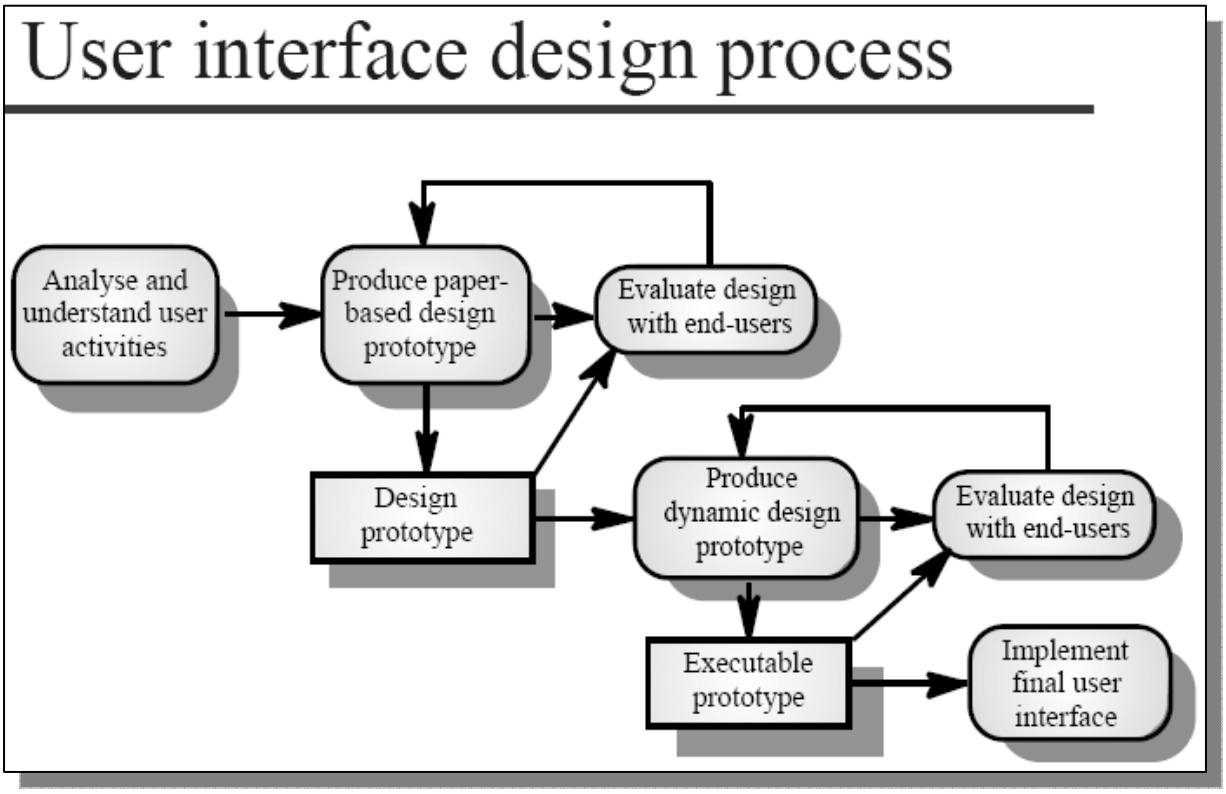
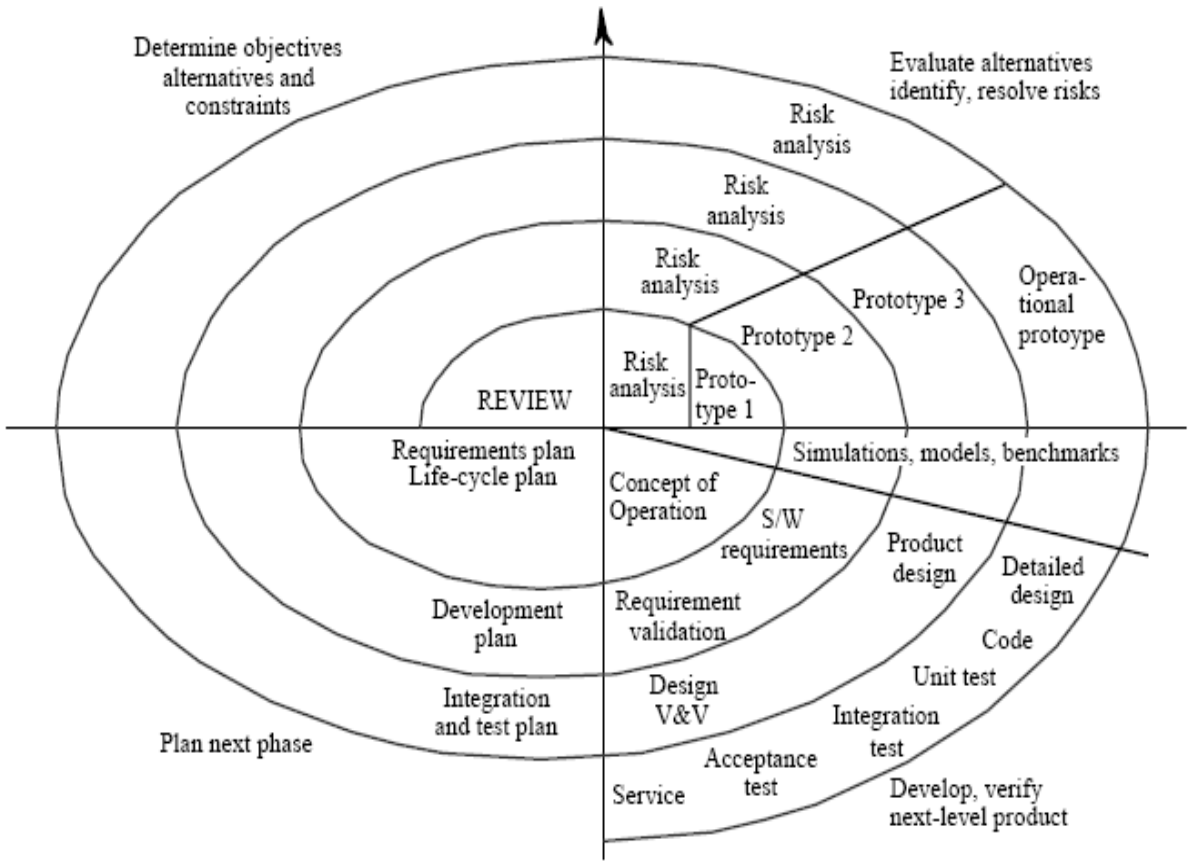


Figure: GUI process with prototype.

Spiral Model:

- The spiral model is a software development process combining which elements of both design & prototyping in stages. In an effort to combine advantages of top-down and bottom-up concepts.
- This model of development combines the features of prototyping model and for the waterfall model.
- The spiral model is only for large expensive and complicated project.
- The spiral model was defined by **Barry. W. Boehm** in his **1986** article "**A spiral model of software development & enhancement**".
- A preliminary design is created for the new system. This phase is the most important part of spiral model. In this phase all possible alternatives which can held in developing a cost effective project are analyzed & strategies to use them are decided this phase has been added specially in order to identify & resolve all he possible risk in the project development.
- In risk identify and indicate and any kind of a certainly requirement, prototyping may be used to proceed with the available data and find out possible situation in order to deal with the potential changes in the requirements.

Spiral model of the software process



Spiral model sectors

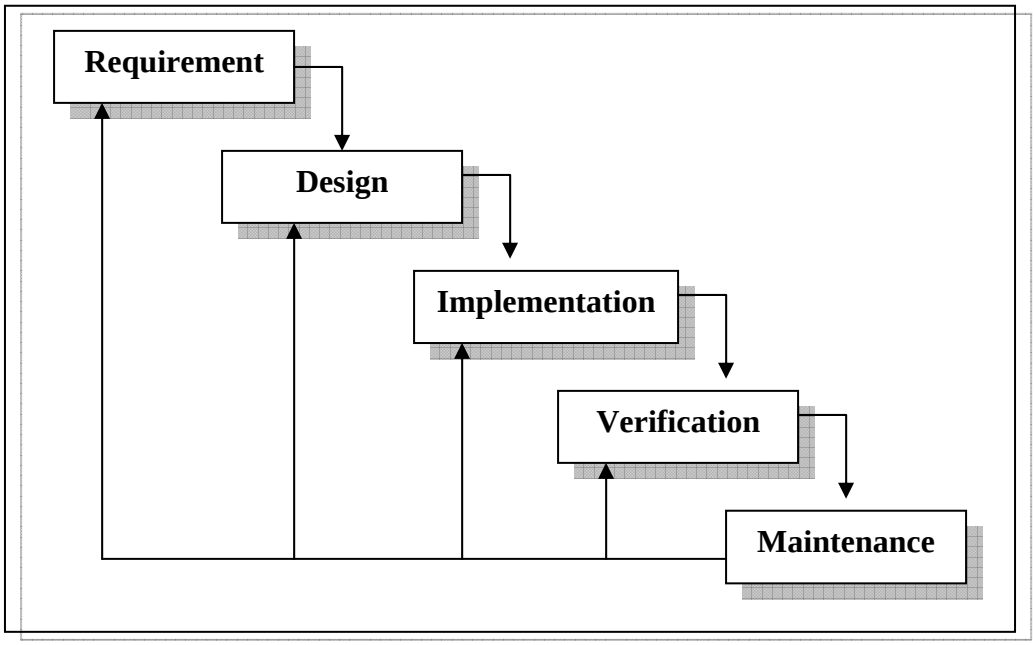
- Objective setting
  - Specific objectives for the phase are identified
- Risk assessment and reduction
  - Risks are assessed and activities put in place to reduce the key risks
- Development and validation
  - A development model for the system is chosen which can be any of the generic models
- Planning
  - The project is reviewed and the next phase of the spiral is planned

Applications of Spiral Model:

- Game development is the main area when the spiral model is used and needed. That is because of the size & the constantly shifting goals of those large projects.
- The U.S. Military has adopted the spiral model for it future combat system (FCS) program.

Waterfall Model:

The waterfall model is a sequential software development process in which the progress is flowing to the downwards through the phases of requirement, design, implementation, verification and maintenance.

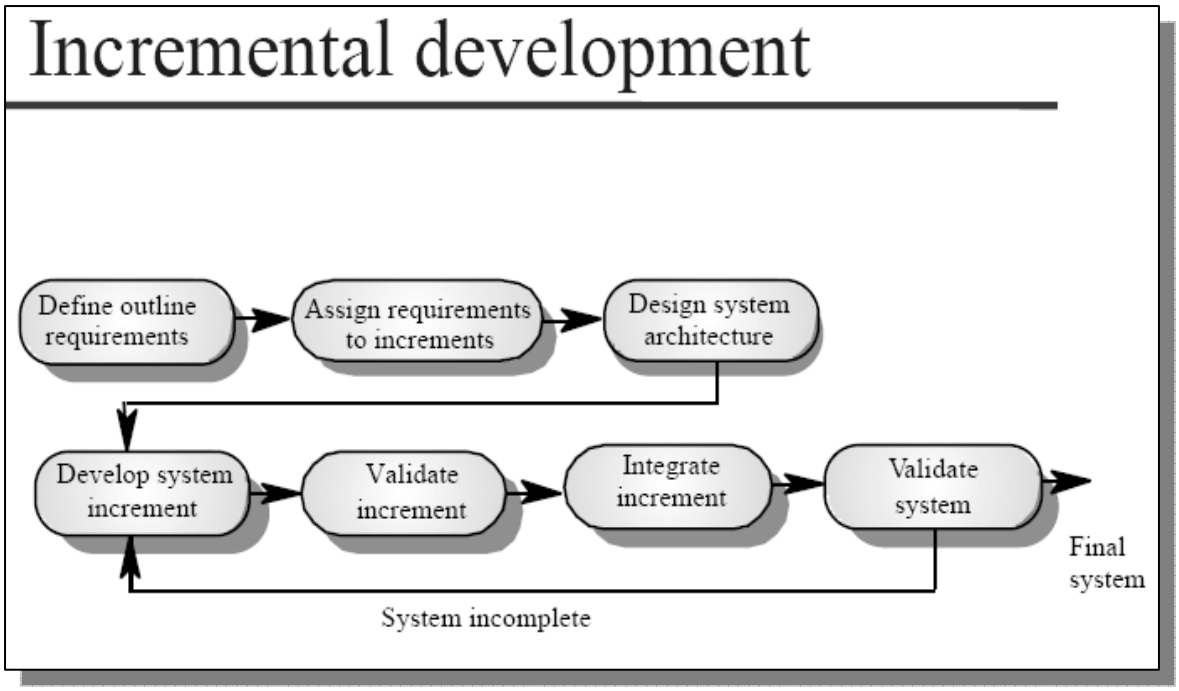


The first formal description of the waterfall model is published in **1970** by **Winston Royce.** The waterfall development model has its origin in manufacturing & construction industry, highly structured physical environment in which some fact changes are costly. If not impossible since no formal software development methodology exits at the time. This hardware oriented model was simply adapted for software development. The waterfall model is generally used to develop a complex software development process.

Iterative & Incremental Model:

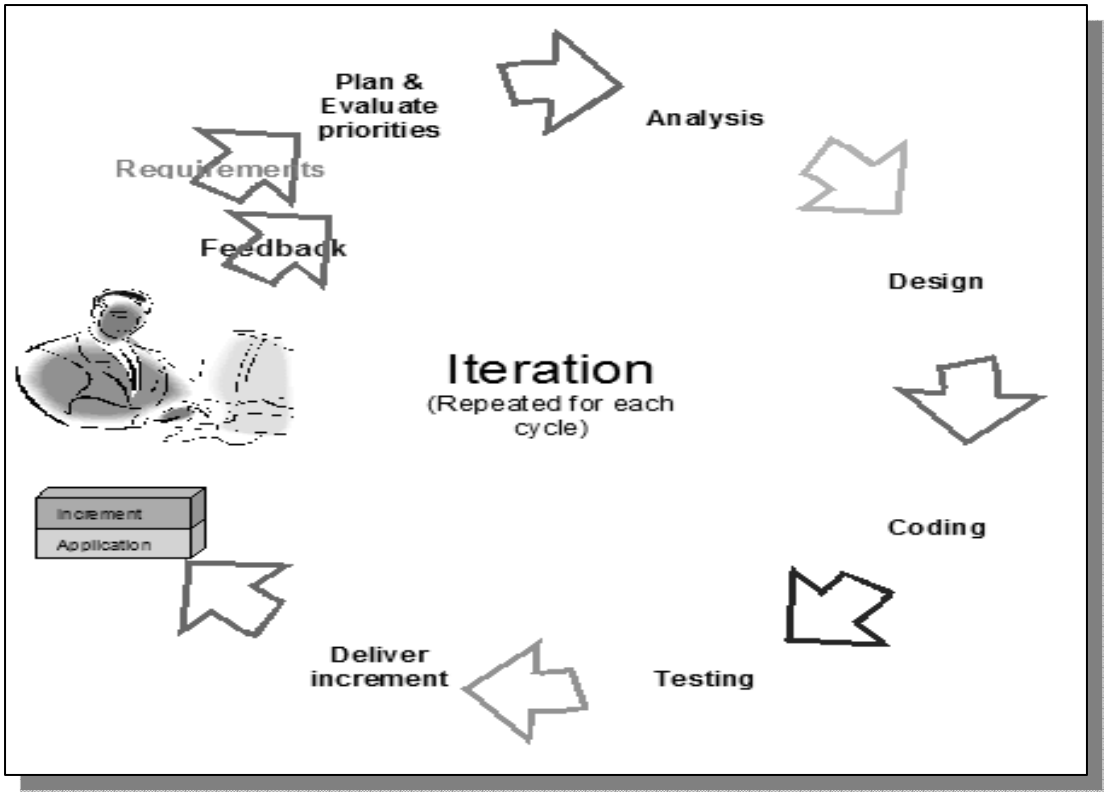
- Iterative & incremental development is cyclic software development processes develop in response to the weaknesses of the waterfall model.
- It starts with an initial planning and ends with deployment with the cyclist interaction.

- The iterative & incremental development is an essential part of the **Rational Unified Process**, **Extreme programming** & generally the **Agile software development** frame works.
- Iterative development slices the business value (system functionality) into iterations. In each iteration a slice of functionality is delivered through cross discipline work. Starting from the requirement through to the deployment.



Incremental development

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality
- User requirements are prioritised and the highest priority requirements are included in early increments
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve



Incremental development advantages

- Customer value can be delivered with each increment so system functionality is available earlier
- Early increments act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing

- In iteration development identify project, scope, risk and requirements transition delivers the system into the production operating environment.
- Construction incrementally with the produce from analysis, design, implementation and testing of the functional requirements.
- The iterative models deliver a working architecture that help the top risk & full fill the non functional requirements.

Agile Software Development Model:

- Agile software development refers to a group of software development methodology based on iterative development.
- Where requirement & solution involve through collaboration between self organizing cross functional teams. The term was introduced in the year 2001 by Agile manifesto.
- Agile methods generally promote a project management process that encourages frequent inspection and adaptation a leadership philosophy that encourages team work, self organization & co-ordination, a set of engineering practices to allow for rapid delivery of high quality software & a business approach that agile development with customer needs and company goals.

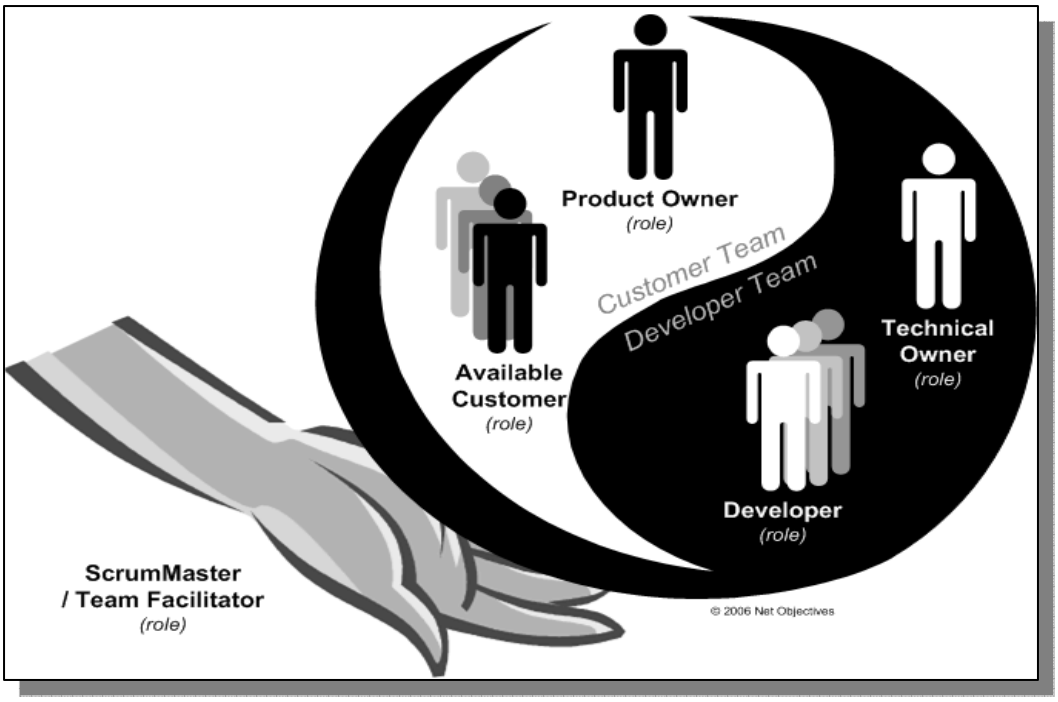


Figure: Agile Process Development.

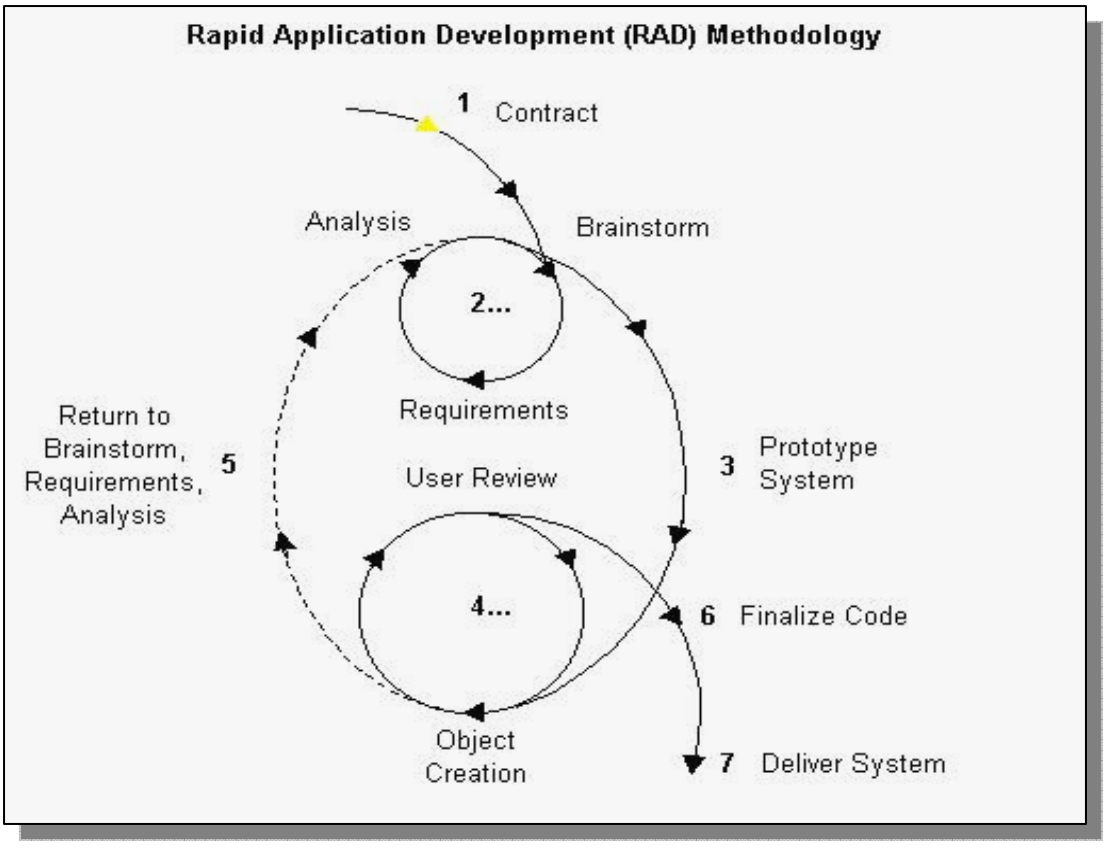
- Customer satisfaction by rapid continues delivery of useful software.
- Working software is delivered frequently (week's rather then months). Working software is running principal of process major. Late changes in requirements are also welcome.
- Close, daily co-operation between business people & developers. Face to face conversion is the best form of communication.
- Continues attention to technical & good design, simplicity, and self organizing teams. Regular adaptation to changing new requirements.

Rapid Application Development (RAD):

Rapid Application Development (RAD) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Rapid Application Development is a software development methodology that involves techniques like iterative development and software prototyping. According to Whitten (2004), it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.

In Rapid Application Development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems". RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.



History

Rapid Application Development is a term originally used to describe a software development process introduced by James Martin in 1991. Martin's methodology involves iterative development and the construction of prototypes. More recently, the term and its acronym have come to be used in a broader, generic sense that encompasses a variety of techniques aimed at speeding application development, such as the use of web application frameworks and other types of software frameworks.

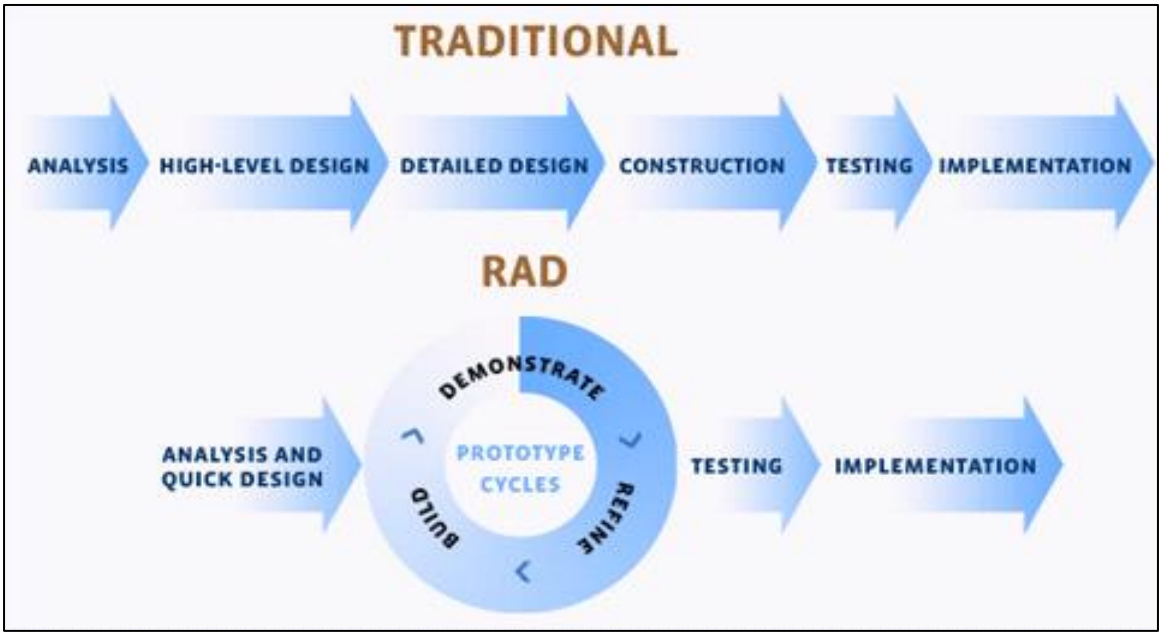


Figure: Rapid Application Development Model Vs Traditional Model

Advantage:

Promotes strong collaborative atmosphere and dynamic gathering of requirements. Business owner actively participates in prototyping, writing test cases and performing unit testing.

Limitation:

Dependency on strong cohesive teams and individual commitment to the project. Decision making relies on the feature functionality team and a communal decision-making process with lesser degree of centralized PM and engineering authority.

- : Software Project Planning (Management): -

- Once a project is found to be physical, software project manager under take project planning.
- Project planning is under taken and completed even before any development activities starts. Planning of project consist of the following essential activities.

Figure: Software Project Management process.

1. **Estimation:**

   - **Cost:** How much is it going to develop the software?
   - **Duration:** How long is it going to take a development the project?
   - **Effort:** How much effort would be require developing the product?

The effectiveness of all other planning activities such as scheduling & staffing are based on the accuracy of this estimation.

2. **Scheduling:**

After the estimates are made the schedules for man power & other resources have to be develop this scheduling display the project status & project schedule.

3. **Staffing:**

Staff organization and staffing plans have to be made.

4. **Risk Management:**

Risk identification, Risk analysis and planning have to be made.

5. Miscellaneous (Others):

Several other plans such as **Quality Assurance Plan**, **Configuration Management** plan have to be made.

Types of project plan

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

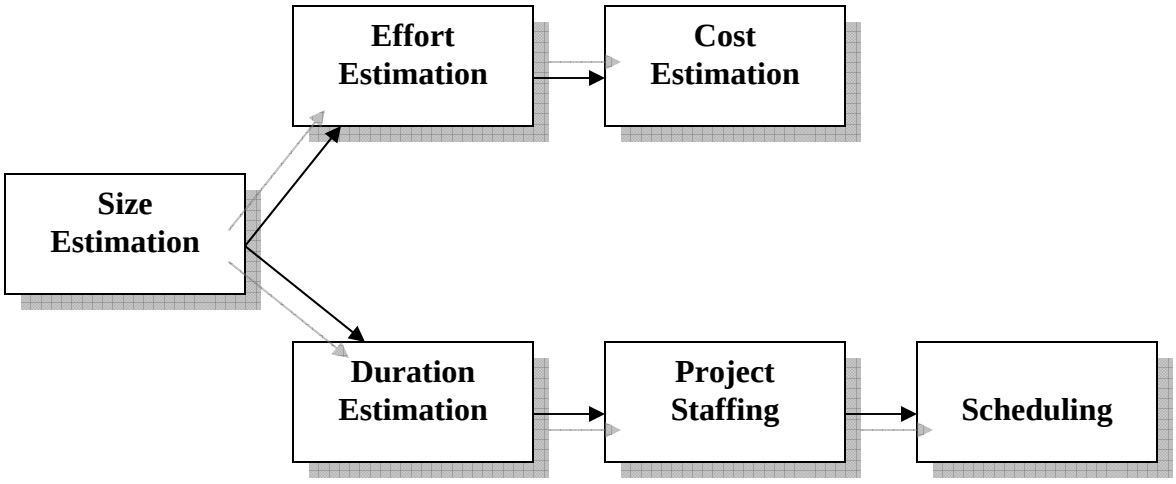


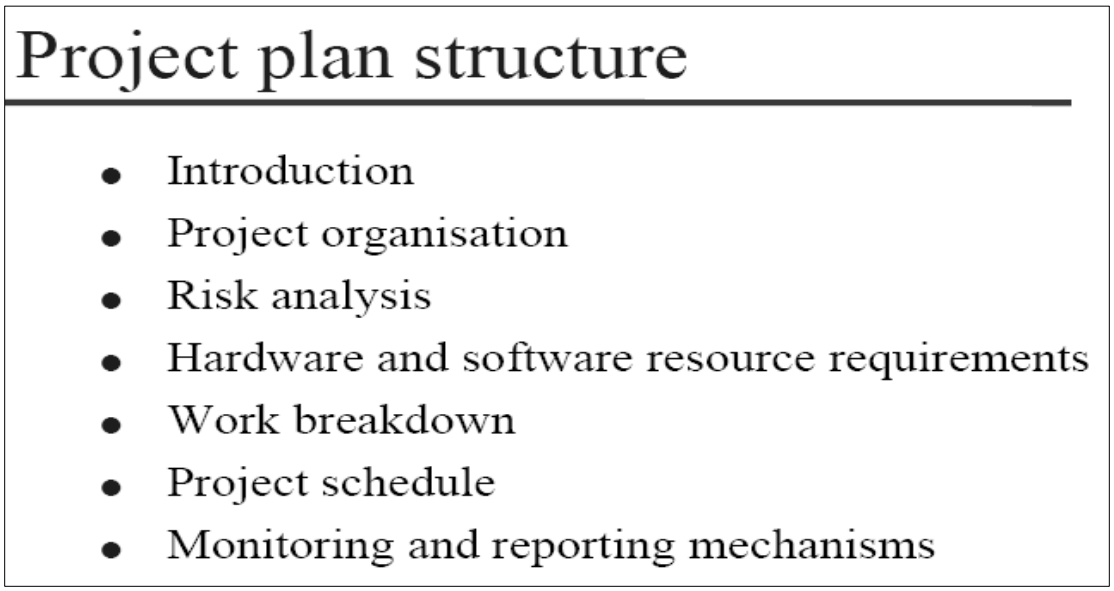
Figure: Software Project Planning

The Software Project Management Plan (SPMP) Document:

1. **INTRODUCTION:**
    - Objectives
    - Major Functions
    - Performance Issue
    - Management & Technical constraint
2. **PROJECT ESTIMATES:**
    - Historical data used
    - Estimation techniques i.e. Delphi , COCOMO
    - Effort, Resources, Cost and project duration estimates
3. **SCHEDULES:**
    - Work Breakdown structure
    - Task network representation
    - Gantt Chart representation
    - PERT Chart representation
4. **PROJECT RESOURCES:**
    - People
    - Hardware & Software
    - Special Resources
5. **STAFF ORGANIZATION:**
    - Team Structure
    - Management Reporting
6. **RISK MANAGEMENT PLAN:**
    - Risk Analysis
    - Risk Identification
    - Risk Estimation

7. MISCELLANEOUS PLAN:

- Quality Assurance Plan
- Configuration Management Plan
- Validation & Verification
- System Testing Plan
- Delivery, Installation & Maintenance plan



Software Project Estimation (Cost Estimation):

- **Introduction:**

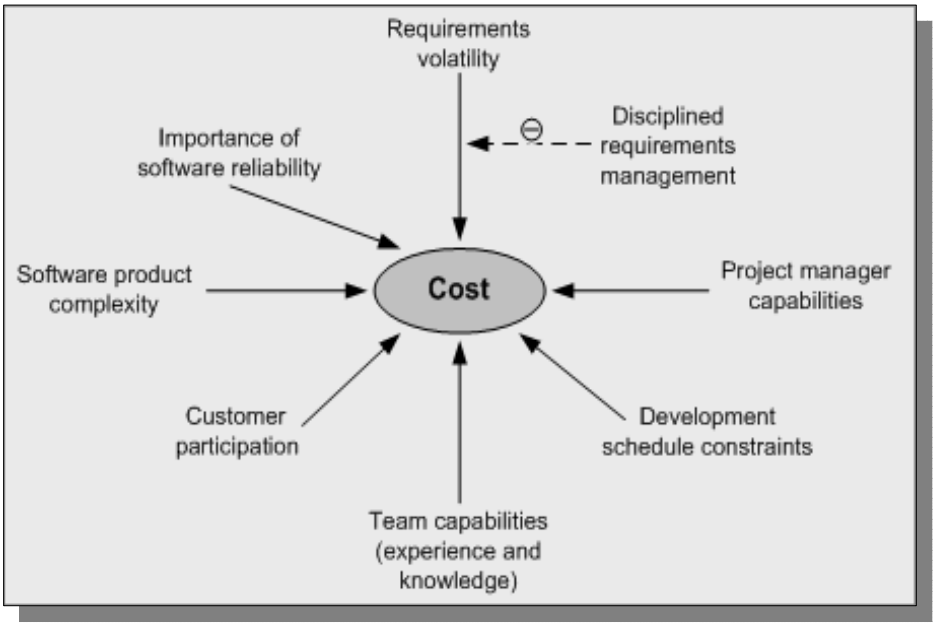
  Software project estimation focus of cost and effort required to develop the software. Different types of cost estimation technique such as Delphi cost estimation & COCOMO model can be used.

- **Project Estimation:**

In early way of computing, software cost of a small percentage of the overall computer-based system cost. An order of error in estimates of software cost had relatively little impact. Today, software is the most expansive development of computer based system. Software cost and effort estimation will never be an exact science. To many variables human, technical, environmental, political can affect the ultimate cost of the software and effort applied to develop it. However, software project estimation can be transformed from a black art to series of systematic steps that provide estimates with acceptable risk.

To achieve reliable cost and effort estimates, a number of options arise.

1. Delay estimation until late in the project.
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition technique to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.



- **Software cost factors:**

There are many factors that influence the cost of a software product.
The effects of most of this factors and hence the cost of a development or maintenance effort, are difficult to estimate. Major factors that influence software cost are:

1. Programmers Ability:

On very large product, the differences in individual programmer ability will tend to average out, but on projects utilizing five or fewer programmers, individual differences in ability can be significant.

2. Programmers Ability:

There are three generally acknowledge categories of software products:

I. Application Programs:

Include data processing and scientific programs. Application programs are developed in the environment provided by a language complier. Interactions with the operating system are limited to job control statements and run-time support facilities provided by the language processor.

II. Utility Programs:

Such as compilers, linkage editors, and inventory systems. Utility programs are written to provide user-processing environment and make sophisticated use of operating system.

III. System Programs:

Such as database management systems, operating systems and real-time systems. System programs interact directly with the hardware. They typically involve concurrent processing and timing constraints.

3. Product size:

A large software product is obviously more expensive to develop that a small one. Boehm's equation indicates that the rate of increase in required effort grows with the number of instructions at an exponential rate slightly greater than 1.

4. Available Time:

Total project effort is sensitive to the calendar time available for project completion. Several investigators have studied the question of optimal development time, and most of compressed or expanded from the optimal time.

5. Level of Technology:

The programming language, the abstract machine, the programming practices, and the software tools used reflect the levels of technology in a software development project. It is well known that the number of source instruction written per day is largely independent of the language used.

- **Different methods of Estimation:**

Cost estimation can be made either.

1. Top-Down Estimation:

First focuses on system-level costs such as the computing resources and personnel required to develop the system, as well as the costs of **Configuration Management, Quality Assurance, System Integration, Training, and Publications, Personnel cost** are estimated by examining the cost of similar past projects.

2. Top-Down Estimation:

First estimate the cost of develop each module of subsystem. Those costs are combined to arrive at an overall estimate. It emphasizes the costs associated with the developing individual system components, but may fail to account for system-level costs, such as configuration management and quality control.

Delphi Cost Estimation Model:

The Delphi cost technique was developed at the Rand Corporation in 1948 to gain expert consensus without introducing the adverse side effects of group meetings. The Delphi technique can be adapted to software cost estimation in the following manner.

1. A coordinator provides each estimator with the system definition document and a form for recording a cost estimate.
2. Estimators study the definition and complete their estimates anonymously. They may ask questions to the coordinator, but they do not discuss their estimates with one another.
3. The coordinate prepares and distributes a summary of the estimator's responses, and includes any unusual rationales noted by the estimators.
4. Estimators complete another estimate, again anonymously, using the results from the previous estimate. Estimator's whose estimate differs sharply from a group may be asked, anonymously, to provide justification for their estimates.
5. The process is iterated for as many rounds as required. No group discussion is allowed during the entire process

The following approach is a variation on the standard Delphi Technique increase communication while preserving anonymity.

1. The coordinator provides each estimator with the system Definition and estimation form.

2. The estimators study the definition and the coordinator call a group meeting so that estimators can discuss estimation issues with the coordinator and one another.
3. Estimators complete their estimates anonymously.
4. The coordinator prepares a summary of the estimates, but does not record rational.

5. The coordinator calls a group meeting to focus on issues where the estimates vary widely.

6. Estimation completes another estimate, again anonymously. The process iterated for as many rounds as necessary.

COCOMO Model:

- **History Of COCOMO Model:**

The COCOMO (**C**onstructive **C**ost **M**odel) is a software cost estimation model developed by Barry W. Boehm. The model uses a basic regression formula, with parameters that are derived from historical project data and account project characteristics.

COCOMO was first published in 1981 Barry W. Boehm's book "Software Engineering economics". As a model for estimating effort, cost and schedule for software projects. This model of study TRW aerospace to project ranging in size from 2000 to 10,000 lines of code, and programming languages ranging from assembly to PL/I. These projects were based on the waterfall model of software development process in 1981.

In 1997 COCOMO II was developed and finally published in 2000 in the book software cost estimation with COCOMO II. COCOMO II is the successor of COCOMO 81 and his better suite for estimating modern software development projects.

COCOMO consist of a hierarchy of three increasing retail and accurate from the first level basic COCOMO is good for quick early rough order of estimates of software cost by its accuracy is limited due to its lack of factors to account for different in project attributes (Cost Drivers). Intermediate COCOMO take this cost driver into account and detail COCOMO additionally accounts for the individual project phases.

- **COCOMO Hierarchy:**

Barry W. Boehm introduce a hierarchy of software estimation models that name COCOMO (Constructive Cost Model)

- ✓ **Model 1 (BASIC):**

The Basic COCOMO model computes software development effort as a function of program size expressed in estimated line of code.

- ✓ **Model 2 (INTERMEDIATE):**

The Intermediate COCOMO model computes software development effort as a function of program size and a set of Cost Drivers that includes subjective assessment of products, hardware, personal and project attributes.

- ✓ **Model 3(ADVANCED):**

The advanced COCOMO model in corporate all characteristic of the intermediate version with and assignment of cost drivers impact on each step of the software engineering process.

BASIC COCOMO (MODEL 1):

Basic COCOMO software development effort and cost as a function of program size. Program size is expressed in estimated Thousand of Line of Code (KLOC). COCOMO applies to three classes' software projects.

- ✓ **Organic Projects:**

Relatively small, simple software projects in which small teams with good application experience work to a set of less than rigid requirement.

- ✓ **Semi – Detached Projects:**

An intermediate software project in which teams with mixed experience levels must meet a mix of rigid and less than rigid requirements.

- ✓ **Embedded Projects:**

A software project that must be developed within a set of tight hardware, software and operational constrains.

The basic COCOMO equations take from the
Effort applied – a_b (KLOC) b_b [Man – Months]

Development Time – c_b (effort applied) d_b [Months]
People Required – Effort applied / Development Time [Count]

The coefficient $a_{b,}$ $b_{b,}$ $c_{b,\ and}$ d_b are given in the following table.

Software Projects	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick estimate of software cost. However it doesn't account for differences for hardware constrains, personal quality and experience use of modern tools & techniques and so on.

INTERMEDIATE COCOMO (MODEL 2):

Intermediate COCOMO computes software development effort as function of program size and set of "Cost Drivers" that include subjective assessment of product, hardware, personal and project attributes. This extension considers a set of for cost drivers.

Each with a no of attributes:

- **Product attributes**:
  - ✓ Required Reliability
  - ✓ Data-base size
  - ✓ Product Complexity
- **Hardware attributes (Computer):**
  - ✓ Runtime performance constrains
  - ✓ Memory Constrain
  - ✓ Virtual machine environment of volatility
  - ✓ Required turnaround time
- **Personnel attributes:**
  - ✓ Analyst capability
  - ✓ Programmer (System Engineer) capability
  - ✓ Applications experience
  - ✓ Virtual machine experience
  - ✓ Programming language experience

- **Project attributes:**

  - ✓ Use of software tools
  - ✓ Application of software engineering method
  - ✓ Required development schedule

Example:

Each of the fifteen attributes received a ratting on a side point scale that ranges from "Very Low" to Extra High" (In important or value). An effort multiplayer from the table, applies to the rating the product of all effort multipliers result in an "Effort Adjustment Factor" (EAF). The value for EAF range from 0.9 to 1.4 (Standard as per Effort).

Cost Drivers	Low Cost	Low	Normal	High	Very High	Extra High
1. **Product Attributes:**						
✓ Required Reliability	0.75	0.60	1.00	1.75	1.40	-
✓ Data-base size	-	0.94	1.00	1.08	1.16	-
✓ Product Complexity	0.70	0.85	1.00	1.15	1.30	1.65
2. **Hardware Attributes:**						
✓ Runtime performance constraints	-	-	1.00	1.11	1.30	1.66
✓ Memory Constrain	-	-	1.00	1.06	1.21	1.56
✓ Virtual machine environment of volatility	-	0.87	1.00	1.15	1.30	-
✓ Programming language experience	-	0.87	1.00	1.07	1.15	-
3. **Personnel Attributes:**						
✓ Analyst capability	1.46	1.90	1.00	0.86	0.71	-
✓ Programmer (System Engineer) capability	1.42	1.70	1.00	0.86	0.70	-
✓ Applications experience	1.29	1.10	1.00	0.91	0.82	-
✓ Virtual machine experience	1.14	1.10	1.00	0.95	-	-
✓ Programming language experience	1.21	1.10	1.00	0.90	-	-
4. **Project attributes:**						
✓ Use of software tools	1.24	1.10	1.00	0.21	0.83	-
✓ Application of software engineering method	1.24	1.10	1.00	0.91	0.82	-
✓ Required development schedule	1.23	1.08	1.00	1.04	1.10	-

The intermediate COCOMO formula now taken from

$$E = a_l (KLOC)^{b_l} . EAF$$

Where E is the effort applied in person, month. KLOC is the estimated number of thousand of delivered line of code for the project and EAF is the factor calculated from the above. The coefficient a_l and exponent b_l are given in the table.

Software Project	a_l	b_l
Organic	3.2	1.05
Semi detached	3.0	1.12
Embedded	2.8	1.20

The development time of calculation used E in the same way as in the basic COCOMO.

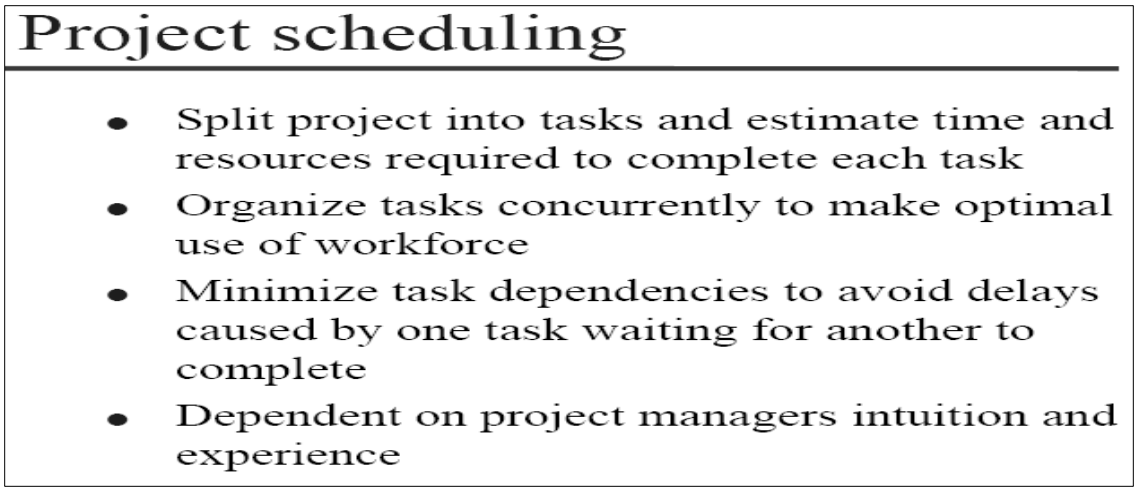
ADVANCE COCOMO (MODEL 3):

Detail COCOMO in corporate all characteristic of intermediate version with an assessment of the cost drivers impact on each step [Analyst, design etc] of the software engineering process the detail model uses different effort multipliers for each cost driver attributes this phase sensitive effort multiplier where each to determine to amount of effort required to complete each phase.

- **Advantage of COCOMO II:**

- Making investment or other financial decision involving a software development effort.
- Setting project budget and schedule as a basic for planning and contract.
- Making software cost and schedule for risk management decision.
- Decision deciding how to implement a process improvement strategy such as SEI CMM.

Software Project Scheduling Management:



Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard
- Productivity is not proportional to the number of people working on a task
- Adding people to a late project makes it later because of communication overheads
- The unexpected always happens. Always allow contingency in planning

- **PERT (Program or Project Evaluation & Review Technique):**

- PERT is a Model for project management design to analyze & represent the task invoke in completing in given project. It is commonly used in conjunction with the Critical Path Method (CPM).
- A PERT chart is a project management tool used to schedule, organize co-ordinate task with in a project.
- PERT stand for Program Evaluation & Review Technique in 1958 to manage the Polaris Submarine Missile project,
- A similar methodology the Critical Path Method (CPM) which is develops for project management in the private sector in the same time.

- **GANTT Chart:**

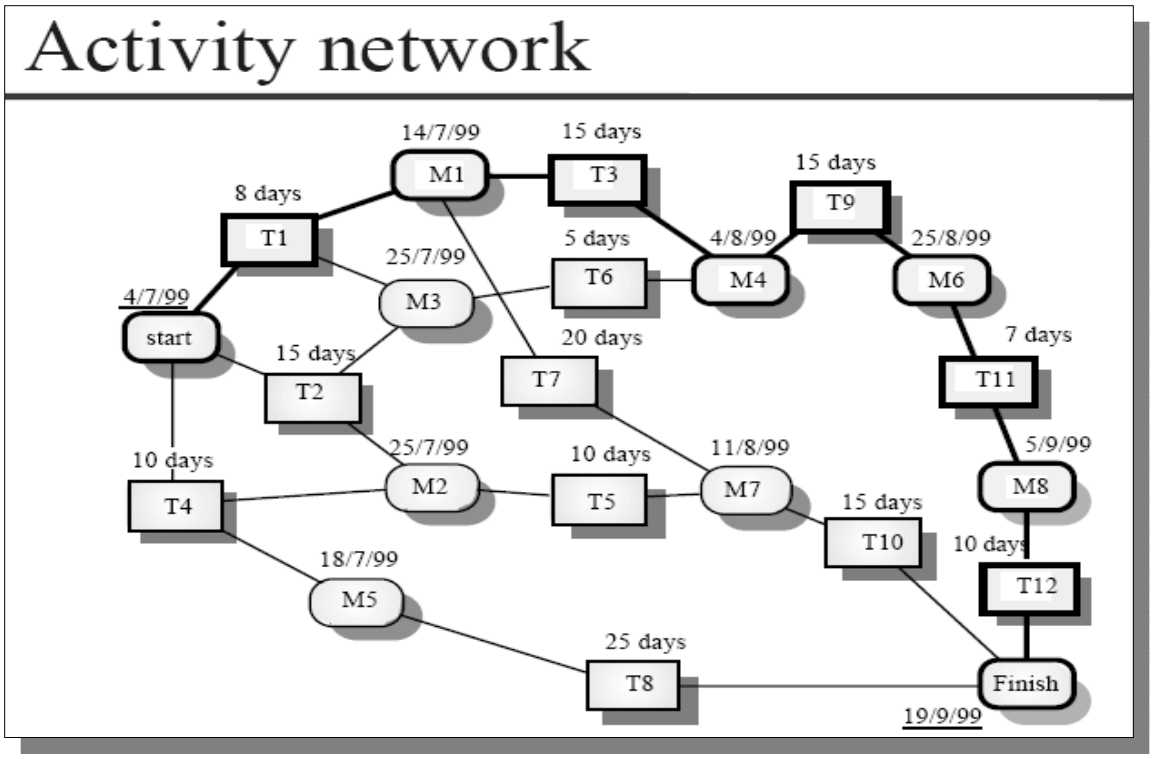
- A Gantt chart is a type of bar chart that illustrates a project schedule.
- Gantt chart illustrates the start & finish dates of the terminal element & summary element of project terminal elements & summary element used the work breakdown structure of the project.
- Some Gantt chart also shows the dependency relationship between activities.

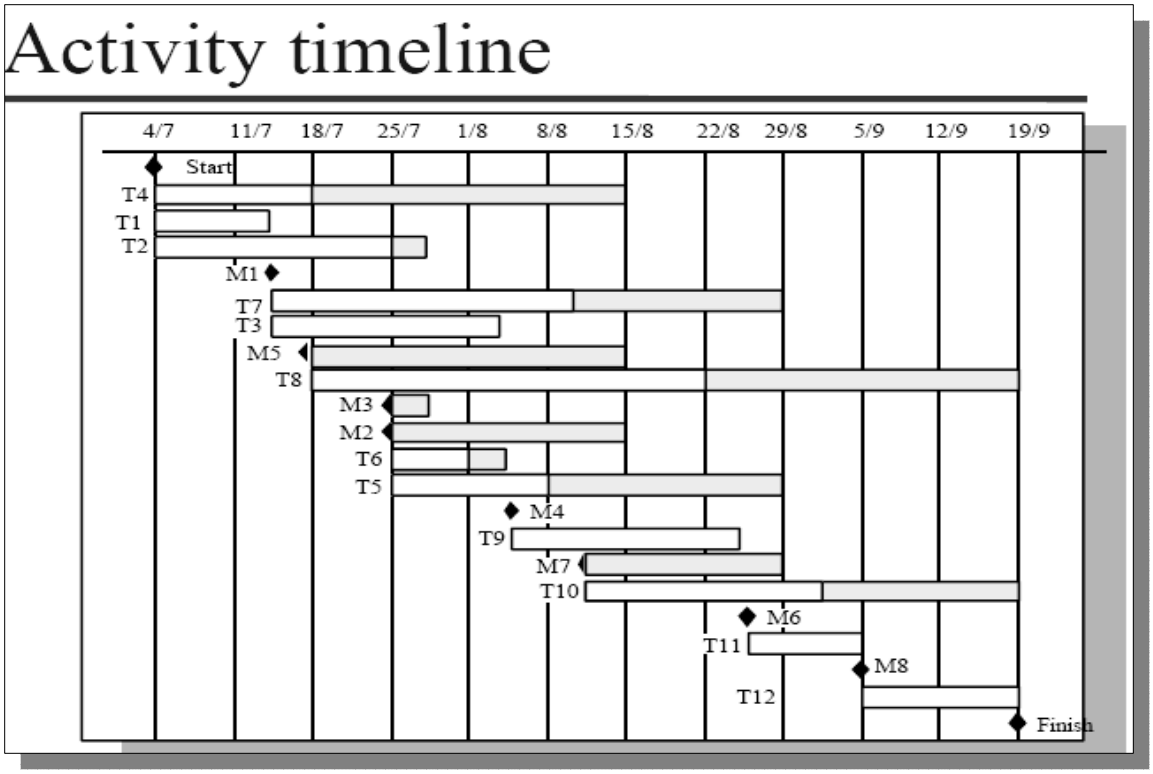
Bar charts and activity networks

- Graphical notations used to illustrate the project schedule
- Show project breakdown into tasks. Tasks should not be too small. They should take about a week or two
- Activity charts show task dependencies and the the critical path
- Bar charts show schedule against calendar time

Task durations and dependencies

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)


Activity network
14/7/99
15 days
15 days
8 days
5 days
4/8/99
25/8/99
25/7/99
4/7/99
7 days
20 days
15 days
25/7/99
10 days
10 days
11/8/99
5/9/99
15 days
10 days
18/7/99
25 days
19/9/99
M1
T3
T9
T1
T6
M4
M6
M3
start
T11
T7
T2
M8
T4
M2
T5
M7
T10
T12
M5
T8
Finish



Software Project Risk Management:

Risk management

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- A risk is a probability that some adverse circumstance will occur.
  - Project risks affect schedule or resources
  - Product risks affect the quality or performance of the software being developed
  - Business risks affect the organisation developing or procuring the software

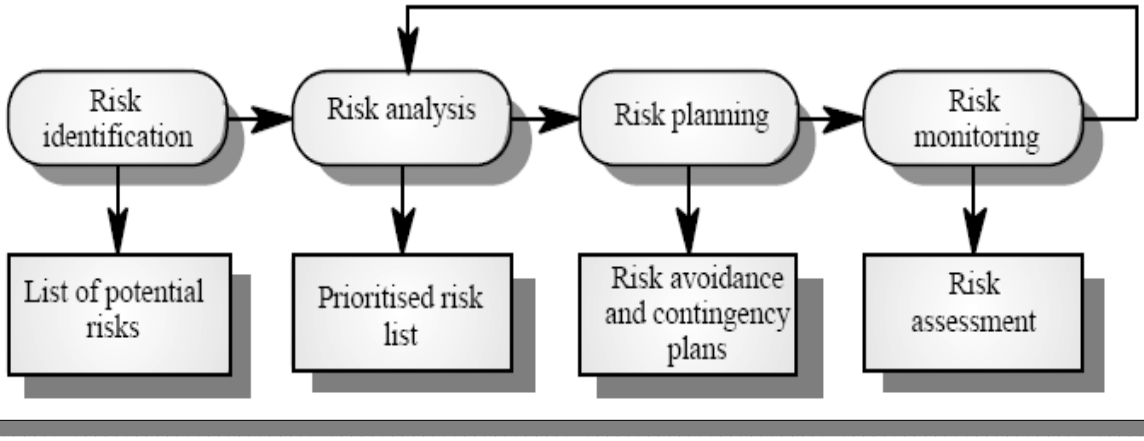
Software risks

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

The risk management process

- Risk identification
  - Identify project, product and business risks
- Risk analysis
  - Assess the likelihood and consequences of these risks
- Risk planning
  - Draw up plans to avoid or minimise the effects of the risk
- Risk monitoring
  - Monitor the risks throughout the project

The risk management process



Risk identification

- Technology risks
- People risks
- Organisational risks
- Requirements risks
- Estimation risks

Risk factors

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, job availability
Organisational	organisational gossip, lack of action by senior management
Tools	reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	many requirements change requests, customer complaints
Estimation	failure to meet agreed schedule, failure to clear reported defects

Key points

- Good project management is essential for project success
- The intangible nature of software causes problems for management
- Managers have diverse roles but their most significant activities are planning, estimating and scheduling
- Planning and estimating are iterative processes which continue throughout the course of a project

- : Quality Assurance: -

- **What is Software Quality:**

Quality: The American Heritage Dictionary Quality as "a characteristic or attribute to something".

Quality Assurance: Quality assurance is the process of defining how software quality can be achieved and how the development organization knows that the software has the required level of quality.

- Quality assurance refers to a program for the systematic monitoring and evaluation of the various process of the project, service or facility to the standards of quality.
- Two key principals characteristic quality assurance "Fit For Purpose" (the product should be suitable for particular purpose) and "Right First Time" (Mistake should be misplace and handled), quality assurance includes regulation of the quality of the raw materials, assemblies, products and components service related to the production and management, production and inspection process.

Software Quality:

- Quality is a customer's determination not an engineer's determination, not a marketing determination.
- It is a based on the customer's actual experience with the product or services, measure against his or her requirements.
- Product and service quality can be defined as "The total composite product and service characteristic of marketing, engineering, manufacture and maintenance through which the product & service and use in meet the acceptance of customers.
- Product can be designed & turned out to give satisfaction at a prize that user will pay. This is not easy & as soon as one fact feels fairly successful in the activities, he finds that the needs of the consumer have changed.
- Qualities that is defined as a matter of products & services whose measurable character satisfy a fixed specification.
- Quality that is identifying independent of any measurable characteristic, that is quality define as the products or services capability to meet customer's acceptance.

Software Quality Standards:

1. **Process standards:**

   Process standards define the process that should be followed during software development specification design and validation process.

2. **Product standards:**

   This standard is applied to the software product being they include document standards such as the structure of requirement document and coding standards that define how a programming language should be used

Process and product quality

- Process quality and product quality are closely related
- A good process is usually required to produce a good product
- For manufactured goods, process is the principal quality determinant
- For design-based activity, other factors are also involved especially the capabilities of the designers

Principal product quality factors

Quality factors

- For large projects with 'average' capabilities, the development process determines product quality
- For small projects, the capabilities of the developers is the main determinant
- The development technology is particularly significant for small projects
- In all cases, if an unrealistic schedule is imposed then product quality will suffer

Step for the quality assurance process:

1. Test previous article
2. Plan to improve
3. Design to include improvement and requirement
4. Manufacture with improvement
5. Review new item and improvement
6. Test new plan

- **Software Quality Models and Factors.**

1. Six Sigma:

It is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980's, the Six Sigma strategy "is a rigorous and disciplined methodology that used data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating 'detects' in manufacturing and service-related processes. The term "six sigma" is derived from six per million occurrences implying an extremely high quality standard.

1. **Define.**
2. **Measure**
3. **Analyze**
4. **Improve**
5. **Control**
6. **Design**
7. **Verify**

2. Total Quality Management (TQM)

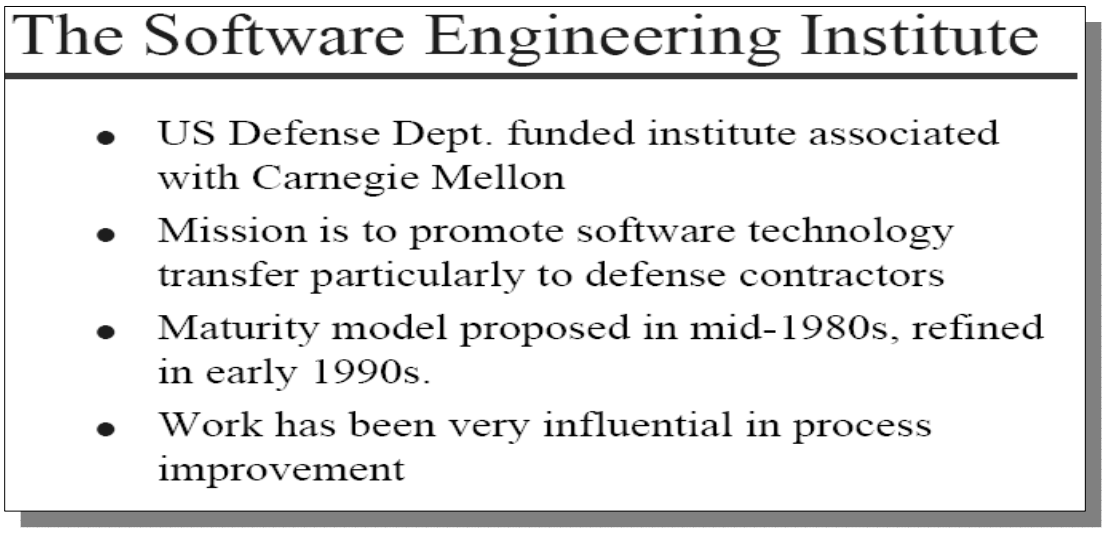
TQM is management concept providing by W. Edwards Deming. The basic of TQM is to reduce the errors produce during the manufacturing of services processes, increase customer satisfaction and handling supply chain management.

One of the principal aim's of TQM is to limit errors to one per millions units produce. TQM is often associated with the development deployment & maintaining of organization system that are requiring for various business processes.

3. The SEI CMM Model

The Software Engineering Institute (SEI) has developed a comprehensive model that is predicted on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity.

To determine an organization's current state of a process maturity, the SEI uses an assessment questionnaire and a five point grading scheme. This grading scheme compliance with a capability maturity model that defines key activities at different level of process maturity.



LEVELS OF CMM:

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimizing

The CMM and ISO 9000

- There is a clear correlation between the key processes in the CMM and the quality management processes in ISO 9000
- The CMM is more detailed and prescriptive and includes a framework for improvement
- Organisations rated as level 2 in the CMM are likely to be ISO 9000 compliant

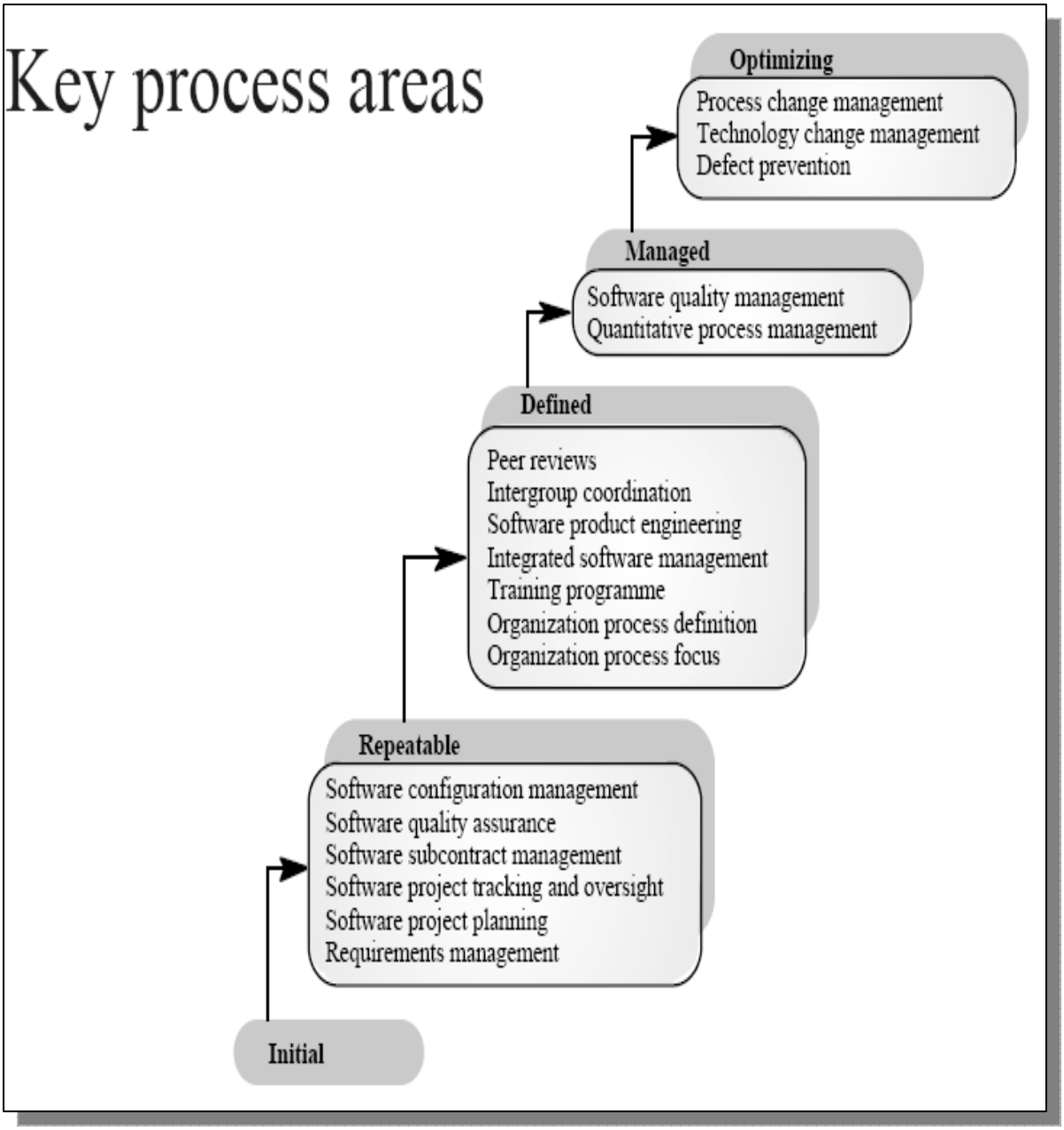


Figure: Process of SEI CMM

1. **Initial:**

The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort. Initial level is essentially uncontrolled.

2. **Repeatable:**

Basic project management processes are established to track cost, schedule and functionality. The necessary process discipline is in place to repeat earlier successes or projects with similar applications.

3. **Defined:**

The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of organization's process for developing and maintaining software. This level included all characteristics defined for level 2.

4. **Managed:**

Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level included all characteristics defined for level 3.

5. **Optimizing:**

Continuous process improvement is enabled by quantitative feedback from process and from testing innovative ideas and technologies. This level included all characteristics defined for level 4.

4. The ISO (International Standard Organization)

It is also called International organization for standardization (ISO).

History:

Formation	23 Feb 1947
Type	NGO
Purpose	International Standardization
Head Quarter	Geneva, Switzerland
Membership	165 member
Official Language	English, Russian, French

ISO 9000 Quality Factors:

- ISO 9000 is a family of standards for quality management system. ISO 9000 has some requirement likes a set of procedures that covers all key processes in the business, monitoring process to conduct they are is effective, keeping records, checking out for defects with appropriate for corrective action where necessary continue improvement.
- An international set of standards that can be used in the development of a quality management system in all industries is calling ISO 9000. ISO 9000 standards can be applied to a range of organizations form manufacturing to service industries.]
- ISO 9001 is most general of these standards and applies to a range of organizations concerned with the quality process in organizations that design, develop and maintain products. The ISO 9001 standards are not specifically aimed to software development but set outs general principals that can be applied to software.

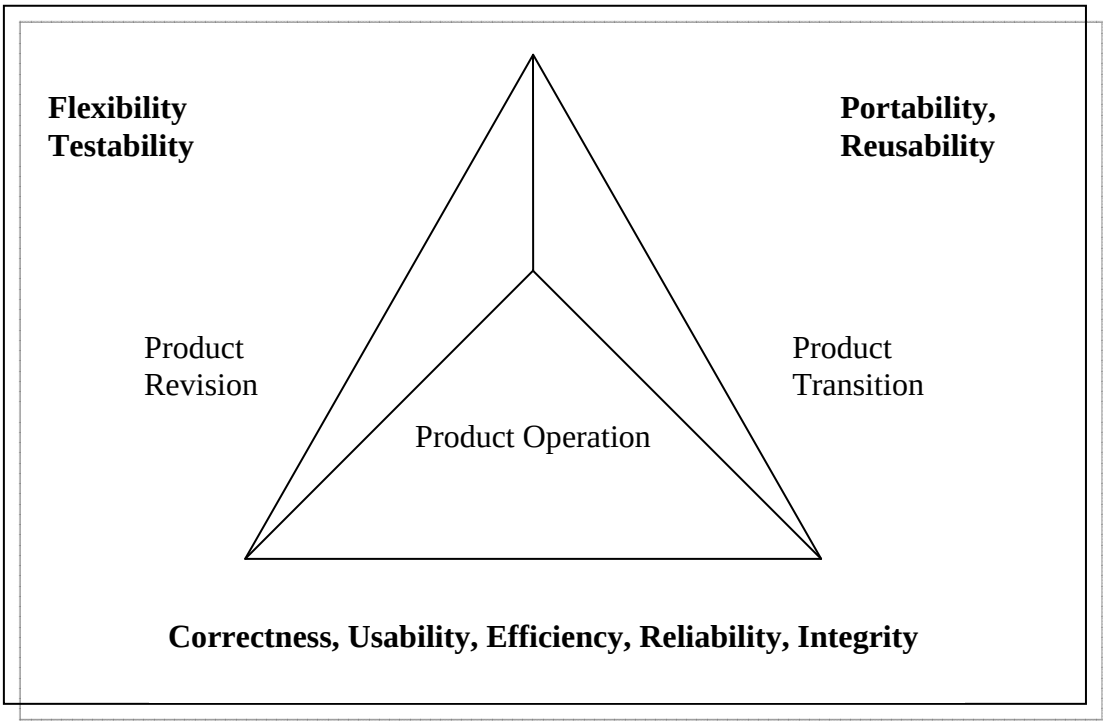
Advantages:

- Create a more efficient, effective
- Increase customers satisfaction
- Reduce audits
- Enhance marketing
- Improve employee motivation, awareness and moral
- Promote international thread
- Increase profit
- Reduce waste & increases productivity

McCall's Quality Factors:

McCall & his colleges' purpose a useful categorization of factor's that affect quality this software quality factors focus on three importance software products.

1. It's operational characteristic.
2. It's ability to undergo changes.
3. It's adaptability to the new environment.



1. **Correctness**

The extent to which a program satisfies its satisfaction and fail fill the customer's mission objectives.

2. **Reliability**

The extent to which a program satisfies its intended functions with required precision.

3. **Usability**

The effort required learning, operating, preparing input, and interpreting output of a program.

4. **Efficiency**

The amount of computing resources and code required by a program to perform its function.

5. **Integrity**

The extent to which access to software or data by unauthorized person can be controlled.

6. **Flexibility**

The effort required to modify an operational program.

7. **Testability**

The effort required to test a program to ensure that performs its intended function.

8. **Portability**

The efforts required to transfer the program to ensure that performs its intended function.

9. **Reusability**

The extent to which a program can be reused in another application related to the packaging and scope of the functions that the program performs.

ISO 9126 Quality factors:

The ISO 9126 standards were developed in attempt to identity quality attributes for computer software. The standard identifies six key attributes.

1. **Functionality:**

- The degree to which the software statistics need as indicated by the following sub attributes like suitability, accuracy, interoperatibility and security.

2. **Reliability:**

- The amount of time the software is available for use as indicated by the following sub attributes.
- Maturity, recoverability and fault tolerance.

3. **Usability:**

- The degree to which software is easy to use as indicated by the following sub attributes like Learn ability, operability.

4. **Efficiency:**

- The degree to which the software makes optimal use of system resources as indicated by the following sub attribute like Time behavior, resource behavior

5. **Maintainability:**

- The easy with which repair may be made the software as indicated by the following sub attributes. Like changeability, stability and testability.

6. **Portability:**

- The easy with which software can be transformed from one environment to another as indicated by following sub attributes. Like Adaptability, install ability and replace ability.

SQA Planning and Standards:

- Quality planning of developing a quality plan for a project. The quality plan should be set out the de desired software qualities and describe how these are to be assessed.

Quality Plan:

- Should select those organization standards that are appropriate to a particular product and development process. New standards may have to be defined if the project uses new methods and tools.

1. **Product Introduction:**

A description of the product, its intended market and the quality expectations for the product.

2. **Product Plans:**

The critical release dates and responsibility for the product along with plans for distributions and product servicing.

3. **Process Description:**

The development and service processes that should be used for product development and management.

4. **Quality Tools:**

The quality tools and plans for the product including an identification and justification of critical product quality attributes.

5. Risk and Risk Management:

The key risks that might affect product quality and the actions to address these risks.

IEEE - Software Quality Plan:

- The purpose and scope of the plan
- A description of all software engineering work product that fall within the preview of SQA.
- All applicable standards and practices that are applied during the software process.
- SQA actions and tasks and their placement throughout the software process.
- The tools and methods that support SQA actions and tasks.
- Software configuration management procedures for managing changes
- Methods for assembling, safeguarding, and maintaining all SQA – related records.
- Organizational roles and responsibilities relative to product quality.

-: Verification And Validation:-

Introduction:

Verification and validation is not the same thing, although they are often confused. Verification and Validation is an expensive process. You should start planning system validation and verification early in the development process. Software inspection is a static verification and verification process in which a software system is reviewed to find errors, omissions and anomalies. Static analyzers are software tools that scan the source text of a program and detect possible fault and anomalies. Mills, Dyer and Linger first proposed the cleanroom philosophy for software engineering during the 1980's. The cleanroom approach makes use of a specialized version of the incremental process model.

Verification and Validation:

Boehm succinctly expressed the difference between them
1. Verification: Are we building the right product?
2. Validation : Are we building the product right?

The ultimate goal of the verification and validation process is to establish confidence that the software system is '**fit for purpose**'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users and the current marketing environment for the system.

Planning Verification and Validation

Verification and Validation is an expensive process. You should start planning system validation and verification early in the development process.

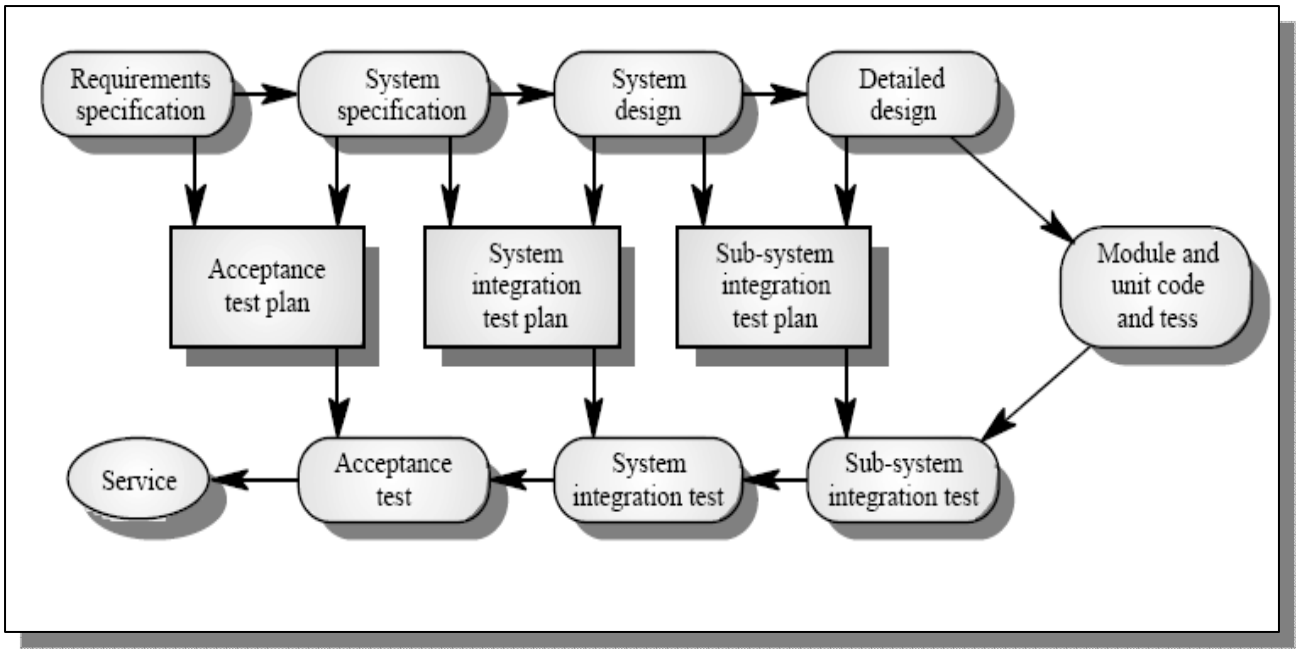


Figure: Test phase as a link between development and testing (V model).

You should start planning system validation and verification early in the development process. The software development process model shown above figure is sometimes called the V – Model. It is an instantiation of the generic waterfall model and shows that test plans should be derived from the system specification and design. This model also breaks down system verification and validation into a number of stages. Each stage is driven by tests that have been defined to check the conformance of the program with its design and specification.

Structure of a software test plan

The major components of a test plan for a large and complex system are:

1. **The Testing Process:**

A description of the major phases of the testing process.

2. Requirement Testability:

Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

3. Tester Items:

The product of the software process that is to be tested should be specified.

4. Testing Schedule:

An overall testing schedule and resources allocation for this schedule is, obliviously, linked to the more general project development schedule.

5. Test Recording Procedures:

It is not enough simply to run tests; the result of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

6. Hardware and Software Requirement:

This section should be set out the software tools required and estimated hardware utilization.

7. Constraints:

Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Test plans are not static documents but evolve during the development process. Test plans change because of delays at other stages in the development process.

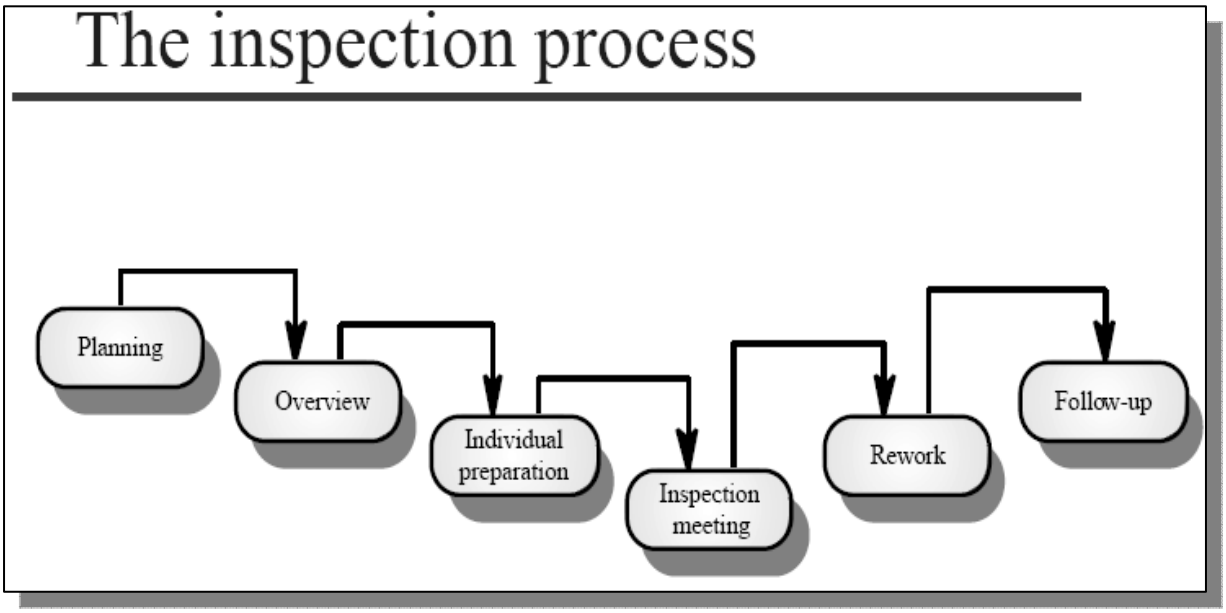
-: Software Inspections (Static V & V Process):-

Software inspection is a static verification and validation process in which a software system is reviewed to find errors, omissions and anomalies. Generally, inspections focus on source code, but any readable representation of the software such as its requirements or a design model can be inspected. When you inspect a system, you use knowledge of the system, its application domain and the programming language of design model to discover errors.

Advantages of Software Inspection over testing:

1. During testing, errors can mask other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.
2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obliviously adds to the system development costs.
3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program such as compliance with standards, portability and maintainability. You can look for interefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

The Program Inspection Process:



1. You have a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
2. The inspection team members are familiar with the organizational standards.
3. An up-to-date, compatible version of the code has been distributed to all team members. There is no point in inspecting code that is 'almost complete' even if a delay causes schedule.

Inspection Checks:

Possible checks that might be made during the inspection process are:

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

ClEANROOM Software Development:

Mills, Dyer and Linger first proposed the cleanroom philosophy for software engineering during the 1980's. Although early experiences with this disciplined approach to software work showed significant promise, it has not gained widespread usage. The cleanroom approach makes use of a specialized version of the incremental process model. Small independent software teams develop a "pipeline of software increments".

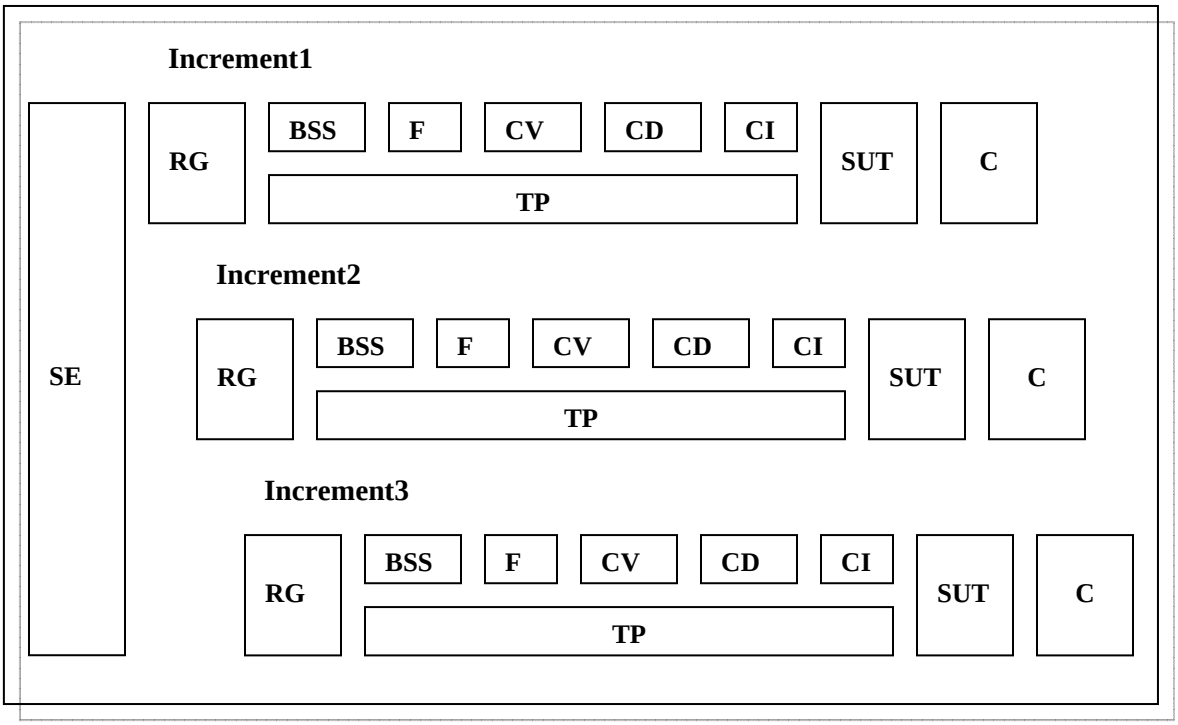


Figure: The Cleanroom process model

SE - System Engineering	**CG - Code Generation**
RG - Requirement Gathering	**CI - Code Inspection**
BSS - Box Structure Specification	**SUT - Statistical Use testing**
FD - Formal Design	**TP - Test Planning**
CV - Correctness Verification	**C - Certification**

The sequence of cleanroom tasks for each increment is illustrated in above figure. Overall system or product requirements are developed using the system engineering methods. Once functionality has been assigned to the software element of the system, the pipeline of cleanroom increments in initiated. The following task occurs:

1. **Increment Planning:**

   A project plan that adopts the incremental strategy is developed. The functionality of each increment, its project size, and a cleanroom development schedule are created. Special care must be taken to ensure that certified elements will be integrated in a timely manner.

2. **Requirement Gathering:**

   A more detailed description of customer-level requirements is developed.

3. Box Structure Specification:

A specification method that makes use of box structure is used to describe the functional specification. Conforming to the operational analysis principles, box structures "isolate and separate the creative definition of behavior, data, and procedures at each level of refinement".

4. Formal Design:

Using the box structure approach, cleanroom design is a natural and seamless extension of specification. Although it is possible to make a clear distinction between the two activities, specifications are iteratively refined to become analogous to architectural and component-level design.

5. Correctness Verifications:

The cleanroom team conducts a series of rigorous correctness verification activities on the design and then the code. Verification begins with the highest-level box structure and moves toward design detail and code, the first level of correctness verification occur by applying a set of "correctness questions". If these do not demonstrate that the specification is correct, more formal methods for verification are used.

6. Code Generation, Inspection and Verification:

The box structure specifications, represented in a specialized language, are translated into the appropriate programming language, standard walkthrough or inspection techniques are then used to ensure semantic conformance of the code and box structure and syntactic correctness of the code. Then correctness verification is conducted for the source code.

7. Statistical Test Planning:

The projected usage of the software is analyzed and a suite of test cases that exercise a "probability distribution" of usage is planned and designed. This cleanroom activity is conducted in parallel with specification, verification, and code generation.

8. Statistical Use Testing:

Statistical use techniques execute a series of tests derived from a statistical sample of all possible program execution by all users from a targeted population.

9. Certification:

Once verification, inspection and use testing have been completed, the increment is certified as ready for integration.

- : Software Testing: -

Introduction:

Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. Software Testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs. Software Testing can also be stated as the process of validating and verifying that a software program/application/product (1) meets the business and technical requirements that guided its design and development; (2) works as expected; and (3) can be implemented with the same characteristics.

Software Testing, depending on the testing method employed, can be implemented at any time in the development process, however the most test effort is employed after the requirements have been defined and coding process has been completed.

Software Testing Fundamentals:

1. Testability:

Software testability is simply how easily [a computer program] can be tested. Since Testing is so profoundly difficult, it pays to know what can be done to streamline it. Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them.

2. Operability:

"The better it works, the more efficiently it can be tested."

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).

3. Observability:

"What you see is what you test."

- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.

- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

4. Controllability:

"The better we can control the software, the more the testing can be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

5. Decomposability:

"By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- The software system is built from independent modules.
- Software modules can be tested independently.

6. Simplicity:

- "The less there is to test, the more quickly we can test it."
- Functional simplicity (e.g., the feature set is the minimum necessary to meet Requirements).
- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

7. Stability:

"The fewer the changes, the fewer the disruptions to testing."

- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

8. Understandability:

"The more information we have, the smarter we will test."

- The design is well understood.

- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible.
- Technical documentation is well organized.

Test Characteristics:

- A good test has a high portability of finding an error.
- A good test is not redundant.
- A good test should be “best of freed (Liberty)”.
- A good test should be neither too simple not to complex.

Software Faults and Failure:

1. Software Bug:



A software bug is the common term used to describe an error, flaw, mistake, failure, or fault in a computer program that prevents it from behaving as intended (e.g., producing an incorrect or unexpected result). Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be buggy. Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, change requests, and so forth.

2. Debugging

Finding and fixing bugs, or "debugging", has always been a major part of computer programming. Maurice Wilkes, an early computing pioneer, described his realization in the late 1940s that much of the rest of his life would be spent finding mistakes in his own programs. As computer programs grow more complex, bugs become more common and difficult to fix. Often programmers spend more time and effort finding and fixing bugs than writing new code.

Usually, the most difficult part of debugging is finding the bug in the source code. Once it is found, correcting it is usually relatively easy. Programs known as debuggers exist to help programmers locate bugs. However, even with the aid of a debugger, locating bugs is something of an art. It is not uncommon for a bug in one section of a program to cause failures in a completely different section, thus making it especially difficult to track (for example, an error in a graphics rendering routine causing a file I/O routine to fail), in an apparently unrelated part of the system.

Testing Pieces:

1. Test Case:

A test case consists of a test case identifier; an input description and an expected results definition. In addition, test case contains a pointer to the corresponding information source to maintain traceability between the test documents.

During test case execution, the tester records such information as actual results, pass/fail status, and version number of the software and hardware used for testing.

Typical written test case format

A test case is usually a single step, or occasionally a sequence of steps, to test the correct behavior/functionalities, features of an application. An expected result or expected outcome is usually given. Additional information that may be included:

- test case ID
- test case description
- test step or order of execution number
- related requirement(s)
- test category
- author
- pass/fail
- remarks

Example of Test Case:

Test Case #	Test case Description	Expected Result	Pass /Fail	Actual Result	Comment	Bug Id
Version Number: A1						
Functionality						
- Login & Logout						
1	Open a web browser and point to **http://url/IP** and click GO button.	Make sure **login page** should display	Pass	Display Login Page	No	------
2	Enter Email & Password then Click on "**Sign in"** Button	Make sure **homepage** should display	Pass	Display Home page	No	------

3	Click the "**Sign out"** Link	Make sure **login page** should display	Pass	Display Login page	No	------
4	Enter Invalid Email Field and Click on **"Sign In"**	Make sure error message should display as "Please Enter valid Email"	Pass	Display Message	No	------
5	With out providing an Email address click on **"Sign In"**	Make sure error message should display as "Enter email address"				
6	With out providing password click on **"Sign In"**	Make sure error message should display as "Enter Password"				
7	Enter wrong Email and Password then click **"Sign In"**	Make sure error message should display check email and password"				
8	Enter valid email in the email field with out entering password in password field	Make sure error message should display as "Enter the password"				
- User Interface						
9	Look at the top of the page	Make sure "XYZ System" banner should display at the top of the page				
10	Look at the bottom of the page	Make sure that the copyright© statement should display at the bottom				

- A test case is usually a single step and its expected result along with various additional piece of information.
- It can occasionally be a series of steps but with one expected result or expected output. The optional fields are a test case id, description, test step, expected results, test category, author is included in format of test case
- These steps can be stored in a **word processor document, spread sheet, database** or other common files.

2. Test Script:

A test script in software testing is a set of instructions that will be performed on the system under test to test that the system functions as expected. There are various means for executing test scripts.

- Manually
- Automated

Short program written in a programming language used to test part of the functionality of a software system. Test scripts written as a short program can either be written using a special automated functional GUI test tool (HP QuickTest Professional, Borland SilkTest or Rational Software) or in a well-known programming language (such as C++, C#, Java, PHP, Perl, Python, or Ruby).

The major **Advantage** of automated testing is that tests may be executed continuously without the need for a human intervention. Another advantage over Manual testing in that it is easily repeatable, and thus is favored when doing regression testing. It is worth considering automating tests if they are to be executed several times, for example as part of regression testing.

Disadvantages of automated testing are that automated tests may be poorly written and can break during playback. Since most systems are designed with human interaction in mind, it is good practice that a human tests the system at some point. Automated tests can only examine what they have been programmed to examine. A trained manual tester can notice that the system under test is misbehaving without being prompted or directed. Therefore, when used in regression testing, manual testers can find new bugs while ensuring that old bugs do not reappear while an automated test can only ensure the latter.

3. Test Plan

A test plan is a systematic approach to testing a system such as a machine or software. The plan typically contains a detailed understanding of what the eventual workflow will be.

A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from Test Engineers.

Depending on the product and the responsibility of the organization to which the test plan applies, a test plan may include one or more of the following:

- **Design Verification or Compliance test** - to be performed during the development or approval stages of the product, typically on a small sample of units.
- **Manufacturing or Production test** - to be performed during preparation or assembly of the product in an ongoing manner for purposes of performance verification and quality control.
- **Acceptance or Commissioning test** - to be performed at the time of delivery or installation of the product.
- **Service and Repair test** - to be performed as required over the service life of the product.....
- **Regression test** - to be performed on an existing operational product, to verify that existing functionality didn't get broken when other aspects of the environment are changed (e.g., upgrading the platform on which an existing application runs).

A complex system may have a high level test plan to address the overall requirements and supporting test plans to address the design details of subsystems and components.

Test plan document formats can be as varied as the products and organizations to which they apply, but there are three major elements of a test strategy that should be described in the test plan: Test Coverage, Test Methods, and Test Responsibilities.

4. Test Harness

In software testing, a test harness or automated test framework is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. It has two main parts: the Test execution engine and the Test script repository.

Test harnesses allow for the automation of tests. They can call functions with supplied parameters and print out and compare the results to the desired value. The test harness is a hook to the developed code, which can be tested using an automation framework.

A test harness should allow specific tests to run (this helps in optimizing), orchestrate a runtime environment, and provide a capability to analyze results.

The typical objectives of a test harness are to:

- Automate the testing process.
- Execute test suites of test cases.
- Generate associated test reports.

A test harness may provide some of the following benefits:

- Increased productivity due to automation of the testing process.
- Increased probability that regression testing will occur.
- Increased quality of software components and application.

5. Test Suite

In software development, a test suite, less commonly known as a validation suite, is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Collections of test cases are sometimes incorrectly termed a test plan, a test script, or even a test scenario.

Occasionally, test suites are used to group similar test cases together. A system might have a smoke test suite that consists only of smoke tests or a test suite for some specific functionality in the system. It may also contain all tests and signify if a test should be used as a smoke test or for some specific functionality.

An executable test suite is a test suite that can be executed by a program. This usually means that a test harness, which is integrated with the suite, exists. The test suite and the test harness together can work on a sufficiently detailed level to correctly communicate with the system under test (SUT).

Static Testing:

Static testing is a form of software testing where the software isn't actually used. This is in contrast to dynamic testing. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code and/or manually reviewing the code or document to find errors. This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used.

From the black box testing point of view, static testing involves reviewing requirements and specifications. This is done with an eye toward completeness or appropriateness for

the task at hand. This is the verification portion of Verification and Validation even static testing can be automated. A static testing test suite consists of programs to be analyzed by an interpreter or a compiler that asserts the programs syntactic validity. Bugs discovered at this stage of development are less expensive to fix than later in the development cycle. The people involved in static testing are application developers, testers, and business analyst.

Testing Methods (Technique):

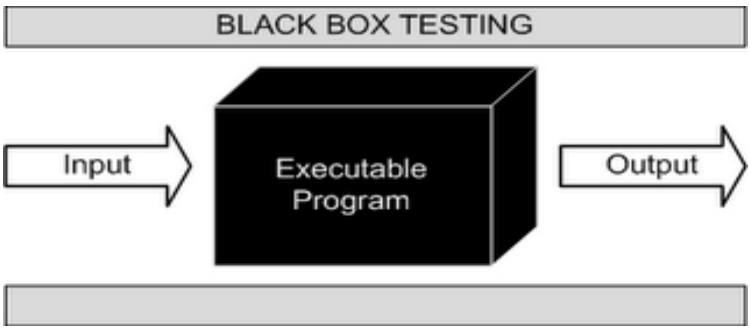
Software testing methods are traditionally divided into black box testing and white box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

- **Black box testing (Functional testing):**

  Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

  Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

  Specification-based testing is necessary, but it is insufficient to guard against certain risks.



  Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code must have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. But, on the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't

know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion," on the one hand, and the disadvantage of "blind exploring," on the other.

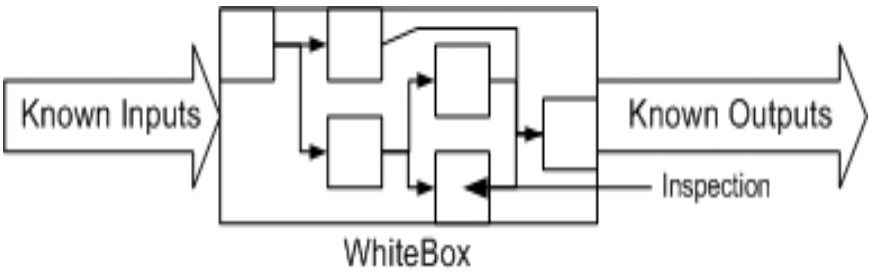
❖ **White box testing (Structural testing):**

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - Testing of the application using Public and Private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods
- Mutation testing methods
- Static testing - White box testing includes all static testing

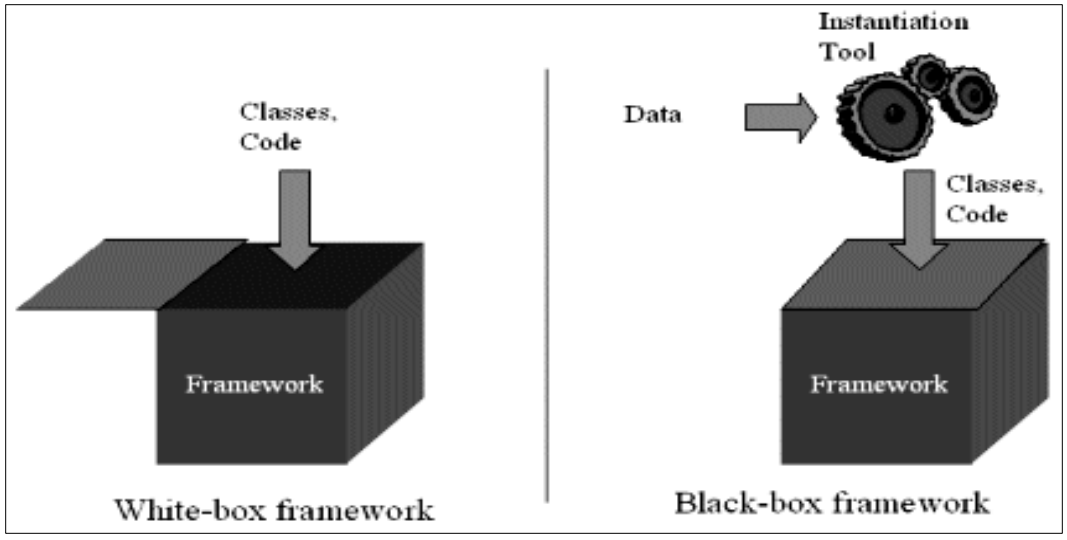


Code completeness evaluation

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested. Two common forms of code coverage are:

- Function coverage, which reports on functions executed
- Statement coverage, which reports on the number of lines executed to complete the test

They both return code coverage metric, measured as a percentage.



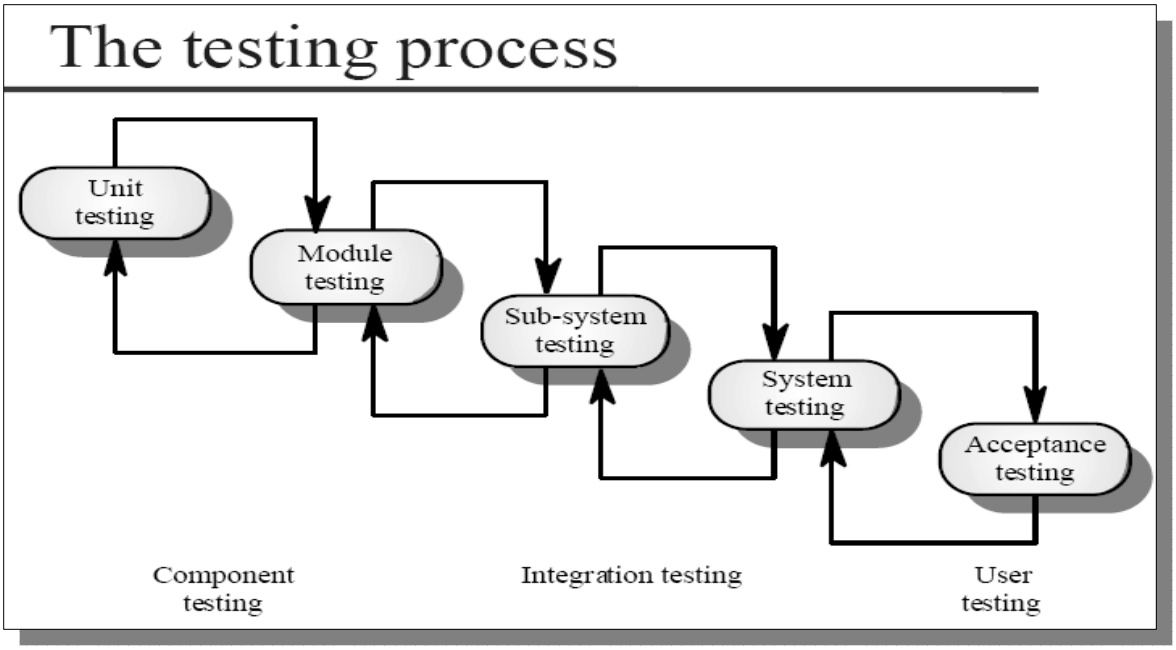
❖ **Grey box testing:**

Grey box testing involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing Process:

A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).



Level of Testing Process:

The system is tested in steps, with the plan and release strategy from individual units of code though integrated subsystem to the deployed released and to the final system. Testing levels include various physical levels of the application development life cycle each complicated level represented a milestone on the project plan & each stand represent an know level of physical integration & quality these stages of integration are known as test levels of testing includes the following.

- **Unit Testing**
- **Integration Testing**
- **System Testing**
- **Regression Testing**
- **Load Testing**
- **Performance Testing**

- **Usability Testing**
- **Storage Testing**
- **Recovery Testing**

1. Unit Testing:

In computer programming, unit testing is a software verification and validation method in which a programmer tests that individual units of source code are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a class, which may belong to a base/super class, abstract class or derived/child class.

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects, fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle.

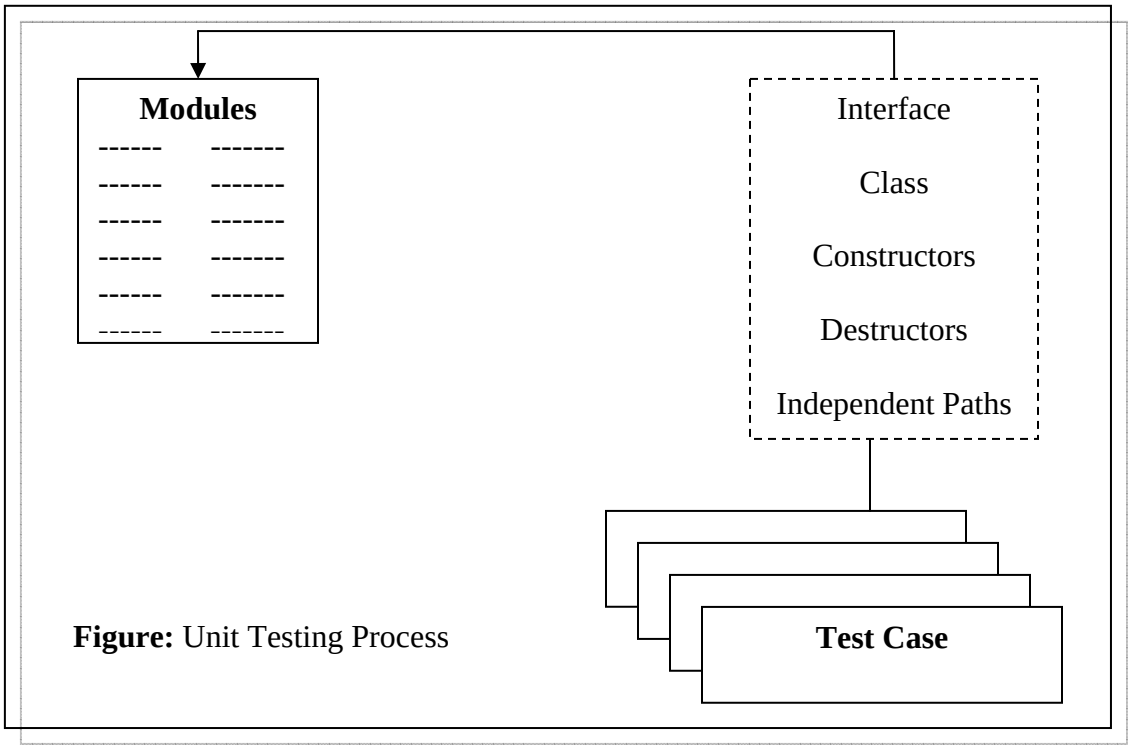


Figure: Unit Testing Process

2. Integration Testing:

Integration testing (sometimes called Integration and Testing, abbreviated "I&T") is the activity of software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing.

Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Purpose

The purpose of integration testing is to verify functional, performance and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using Black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing.

The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Some different types of integration testing are big bang, top-down, and bottom-up.

Limitations

Any conditions not stated in specified integration tests, outside of the confirmation of the execution of design items, will generally not be tested.

3. System Testing:

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are

integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

Testing the whole system

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing is an investigatory testing phase, where the focus is to have almost a destructive attitude and tests not only the design, but also the behavior and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).

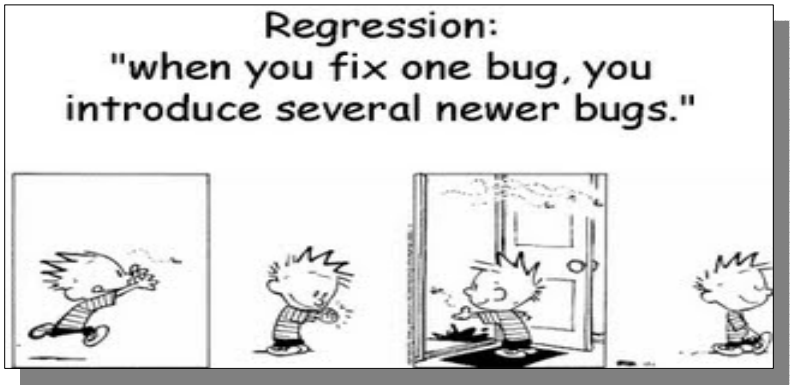
Types of system testing

The following examples are different types of testing that should be considered during System testing:

- **GUI software testing**
- **Usability testing**
- **Performance testing**
- **Compatibility testing**
- **Error handling testing**
- **Security testing**
- **Reliability testing**
- **Recovery testing**
- **Installation testing**
- **Maintenance testing**

4. Regression Testing:

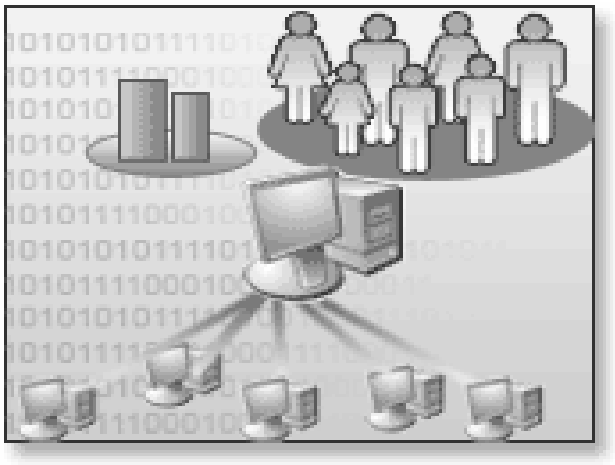
Focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended.



Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run test and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be low risk.

5. Peak Load Testing:

Peak load means to check the data and database and whole program which are used by many person or users in timely manner. For ex. in company all the employee are calling the same program and save their data in same database. It is possible for hardware and software to manage files and check the error.



6. Performance Testing:

When all system is tested then we test how it is work? Proper output is display or not? Work is done in time or not? , Means we check the all program or all system performance sequence by sequence. Performance covers a covers a broad range of engineering or functional evaluation when a material, product, system or person is not specified by detail material or component specification.

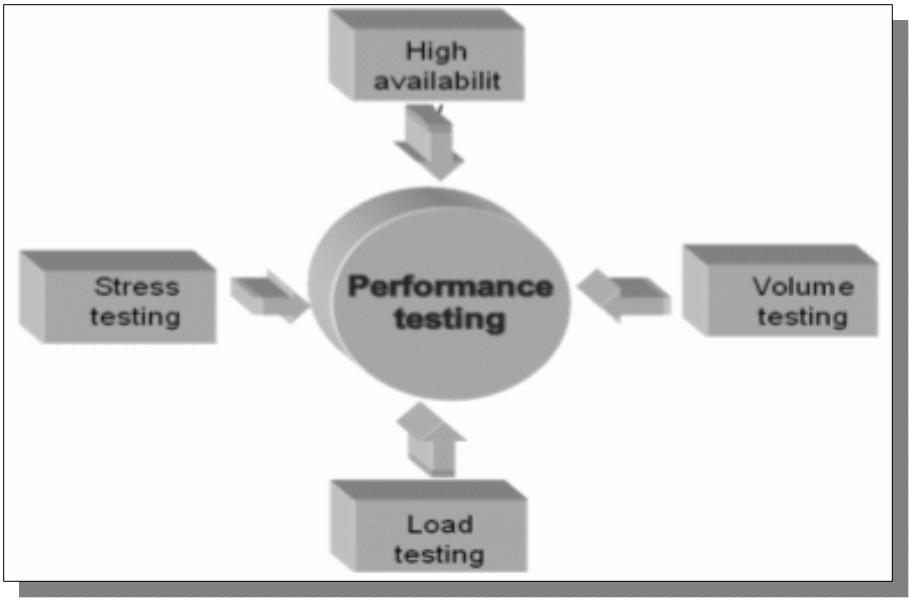


Figure: Performance Testing Process.

7. Usability Testing:

Usability testing is a technique use to evaluate a product by testing it on users. This can be seen as a repeatable usability practice since it gives direct input on real user use the system. This is a focus on measuring a human made products capacity to meet its indent purpose.

Example of products that commonly benefit from usability testing is customer products, website or web application, computer interfaces document & devices. Usability testing measures the usability or easy to use of a specific object or set of objects. Various general human computer interactions are going on.

8. Storage testing:

When a system design analyst specified its capacity & then measured in turn of the no of record that a disk will handled & file can contain. In short storage case means to check all store data for find the errors & correct it.

9. Recovery testing:

Recovery means to think that the system will fail and data will be damage or lost.

Software Walkthrough:

In software engineering, a walkthrough or walk-through is a form of software peer review "in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems".

"Software product" normally refers to some kind of technical document. As indicated by the IEEE definition, this might be a software design document or program source code, but use cases, business process definitions, test case specifications, and a variety of other technical documentation may also be walked through.

A walkthrough differs from software technical reviews in its openness of structure and its objective of familiarization. It differs from software inspection in its ability to suggest direct alterations to the product reviewed its lack of a direct focus on training and process improvement, and its omission of process and product measurement.

Objectives and participants

In general, a walkthrough has one or two broad objectives: to gain feedback about the technical quality or content of the document; and/or to familiarize the audience with the content.

A walkthrough is normally organized and directed by the author of the technical document. Any combination of interested or technically qualified personnel (from within or outside the project) may be included as seems appropriate.

- : Software Requirement Specification (SRS): -

- **Software System Requirement:**

The requirements for a system are the description of the series provides by the system & its operational constraints this requirements reflect the needs of customer for a system that helps solve some problem such as controlling a device placing & order or finding information the process of finding out analyzing, documenting & checking this services and constraints is called requirements engineering.

- **What is Software Requirement Specification:**

A software requirement specification (SRS) is a complete description of the behavior of the system to be developing. A software requirements specification (SRS) is a comprehensive description of the intended purpose and environment for software under development. The SRS fully describes what the software will do and how it will be expected to perform. There are many good definitions of SRS that provides as a good basic upon which we can both define a great specification & help us identify activity in out past efforts. We have to keep in mind that the goal is not to create specification but to create products & great software.

An SRS minimizes the time and effort required by developers to achieve desired goals and also minimizes the development cost. A good SRS defines how an application will interact with system hardware, other programs and human users in a wide variety of real-world situations. Parameters such as operating speed, response time, availability, portability, maintainability, footprint, security and speed of recovery from adverse events are evaluated. Methods of defining an SRS are described by the IEEE (Institute of Electrical and Electronics Engineers) specification 830-1998.

- **What are the benefits of a good SRS?**

The IEEE 830- 1998 standard defines the benefits of a good SRS:

Establish the basis for agreement between the customers and the suppliers on what the software product is to do: The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.
[NOTE: We use it as the basis of our contract with our clients all the time].

Reduce the development effort:

The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and

reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal maunder-standings and inconsistencies early in the development cycle when these problems are easier to correct.

Provide a basis for estimating costs and schedules:

The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates. [NOTE: Again, we use the SRS as the basis for our fixed price estimates]

Provide a baseline for validation and verification:

Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.
[NOTE: We use the SRS to create the Test Plan].

Facilitate transfer:

The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

Serve as a basis for enhancement:

Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

- **What are the characteristics of a good SRS?**

Again from the IEEE standard:
An SRS should be

a) Correct
b) Unambiguous
c) Complete
d) Consistent
e) Ranked for importance and/or stability
f) Verifiable
g) Modifiable
h) Traceable

Correct - Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.

Unambiguous - An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

Complete - A simple judge of this is that is should be all that is needed by the software designers to create the software.

Consistent - The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.

Ranked for Importance - Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS.

Verifiable - Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

Modifiable - Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.

Traceable - Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

- **FORMAT OF A SRS / GENERAL OUTLINE OF A SRS?**

❖ **Cover Page (Title Page):**

Software Requirements Specification

For

<Project>

Version 1.0 approved

Prepared by <author>

<Organization>

<Date created>

❖ **Table of Contents:**

Table of Contents ..... Error! Bookmark not defined.
Revision History **76**
1. Introduction **77**
1.1 Purpose 77
1.2 Document Conventions 77
1.3 Intended Audience and Reading Suggestions 77
1.4 Project Scope 77
1.5 References 77
2. Overall Description **77**
2.1 Product Perspective 77
2.2 Product Features 78
2.3 User Classes and Characteristics 78
2.4 Operating Environment 78
2.5 Design and Implementation Constraints 78
2.6 User Documentation 78
2.7 Assumptions and Dependencies 78
3. System Features **78**
3.1 System Feature 1 79
3.2 System Feature 2 (and so on) 79
4. External Interface Requirements **79**
4.1 User Interfaces 79
4.2 Hardware Interfaces 79
4.3 Software Interfaces 80
4.4 Communications Interfaces 80
5. Other Nonfunctional Requirements **80**
5.1 Performance Requirements 80
5.2 Safety Requirements 80
5.3 Security Requirements 80
5.4 Software Quality Attributes 81
6. Other Requirements **81**
Appendix A: Glossary **81**
Appendix B: Analysis Models **81**
Appendix C: Issues List **81**

Revision History

Name	Date	Reason For Changes	Version

Introduction

Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

Project Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here. An SRS that specifies the next release of an evolving product should contain its own scope statement as a subset of the long-term strategic product vision.>

References

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

Overall Description

Product Perspective

<Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

Product Features

<Summarize the major features the product contains or the significant functions that it performs or lets the user perform. Details will be provided in Section 3, so only a high level summary is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or a class diagram, is often effective.>

User Classes and Characteristics

<Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the favored user classes from those who are less important to satisfy.>

Operating Environment

<Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

Design and Implementation Constraints

<Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

User Documentation

<List the user documentation components (such as user manuals, on-line help, and tutorials) that will be delivered along with the software. Identify any known user documentation delivery formats or standards.>

Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include third-party or commercial components that you plan to use issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

System Features

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

System Feature 1

3.1.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

3.1.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

3.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1:
REQ-2:

System Feature 2 (and so on)

External Interface Requirements

User Interfaces

<Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

Hardware Interfaces

<Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported

device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

Software Interfaces

<Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

Communications Interfaces

<Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.>

Other Nonfunctional Requirements

Performance Requirements

<If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

Safety Requirements

<Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product's design or use. Define any safety certifications that must be satisfied.>

Security Requirements

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

Software Quality Attributes

<Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

Other Requirements

<Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

Appendix A: Glossary

<Define all the terms necessary to properly interpret the SRS, including acronyms and abbreviations. You may wish to build a separate glossary that spans multiple projects or the entire organization, and just include terms specific to a single project in each SRS.>

Appendix B: Analysis Models

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>

Appendix C: Issues List

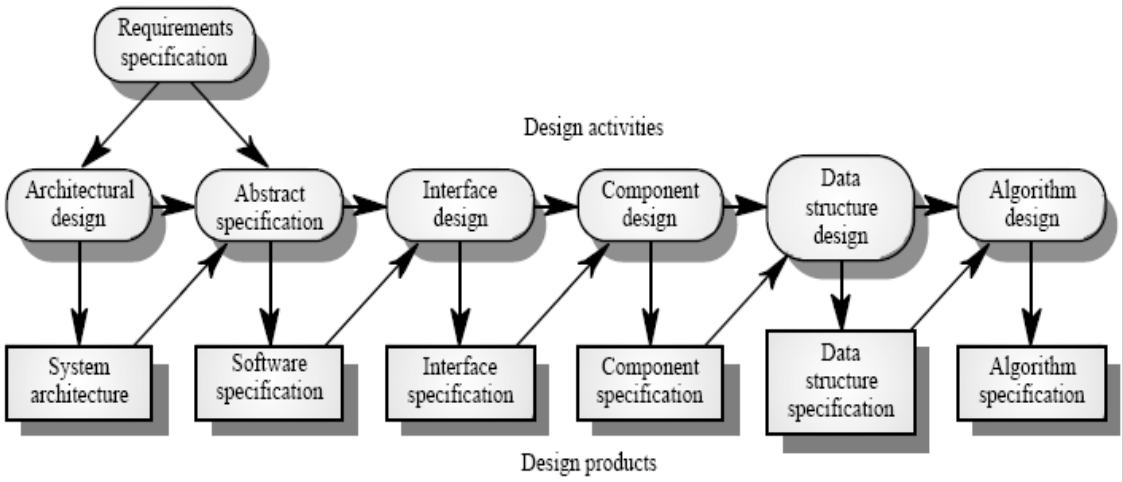
< This is a dynamic list of the open requirements issues that remain to be resolved, including pending decisions, information that is needed, conflicts awaiting resolution, and the like.>

- : Software Design: -

Design process activities

- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design

The software design process



Design methods

- Systematic approaches to developing a software design
- The design is usually documented as a set of graphical models
- Possible models
  - Data-flow model
  - Entity-relation-attribute model
  - Structural model
  - Object models

User Interface Design:

Objectives

- To suggest some general design principles for user interface design
- To explain different interaction styles
- To introduce styles of information presentation
- To describe the user support which should be built-in to user interfaces
- To introduce usability attributes and system approaches to system evaluation

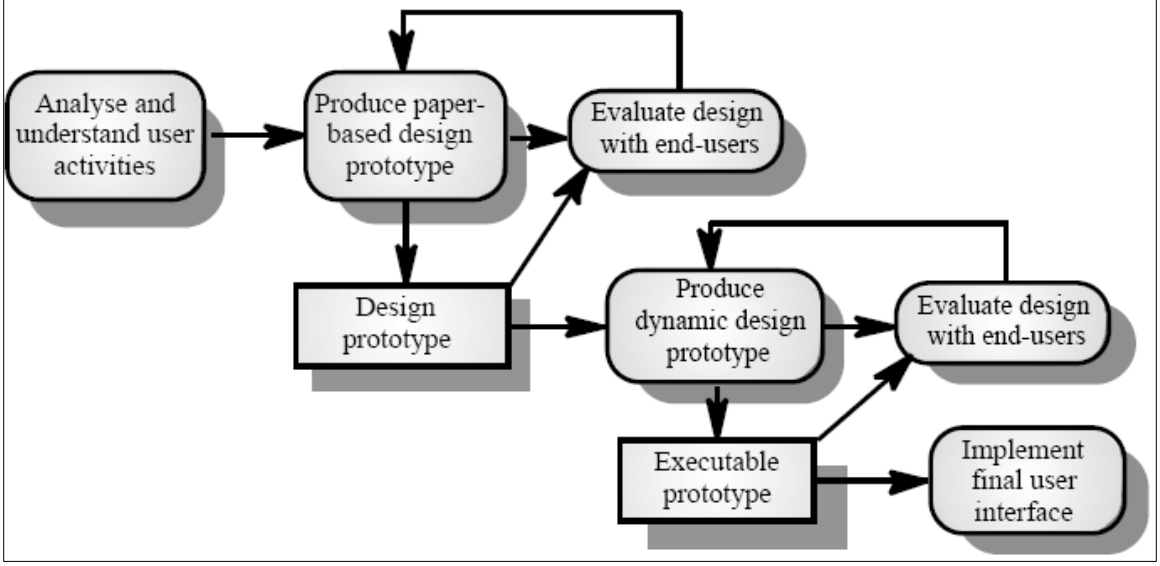
The user interface

- System users often judge a system by its interface rather than its functionality
- A poorly designed interface can cause a user to make catastrophic errors
- Poor user interface design is the reason why so many software systems are never used

GUI characteristics

Characteristic	Description
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files; on others, icons represent processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be mixed with text on the same display.

User interface design process



User interface design principles

Principle	Description
User familiarity	The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system user.

Input Interface Design:

- **Introduction:**

The input design is the link that with information system and user. Input design consists of procedures for data preparation as well data entry. Here Data procedure means by those steps necessary to put transaction data into a usable from for processing and data entry means by activity of putting the data into computer for processing. Data entry can be achieved by computer to read data from a written or printed document, or by having data directly into the system.

There are some ways to guide the design of Input/Output are:

Controlling amount of input:
Because data entry operations are dependent of people, if possible less data entry leads low cost. Second as data input is high leads much time consumption for processing.

Avoiding Delay:
This means by, how analyst can design input of the system so that time delay can be reduced for input.

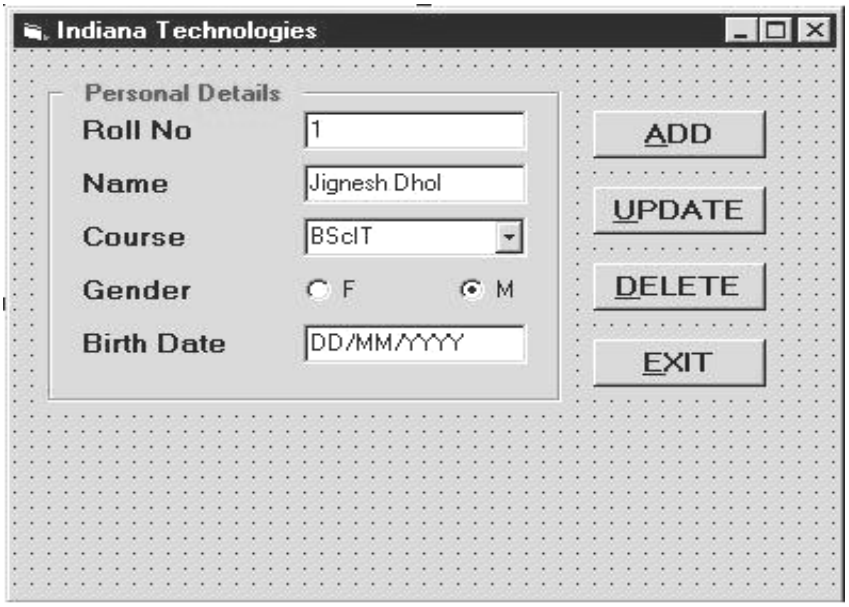
Avoiding Errors in Data:
The rate of errors directly depends on data input. The analyst can also affect the error rates of an operation through input design by the way data be entered.

Avoiding extra steps:
When the volume of transactions can not be reduced, the analyst must be sure the process is as efficient as possible. Perfect analyst will so avoid input designs that cause extra steps. This will save time as well less data entry.

Keeping the process simple:
The users always accept Simple input design. It is advisable to avoid complexity when there are simple alternatives for complex systems.

- **Caption and Data capture of input design.**

  Captions are static information available on the input design. This guides user for data entry. Captions tell user what data to provide and where they should be entered. For better and user friendly layout one should avoid using abbreviations for caption. Caption should be proper informative for respected data entry. For example, to insert date value in birth-date find one has to mention format of input data, like ”dd-Mon-yy‘, DD/MM/YY‘ or ”MM/DD/YYYY‘ Figure shows student data entry screen with respected finds and captions. Fields are mentioned with boxes and rests are captions.



A well-designed source document is easily completed and allows rapid data entry. Captions can be mentioned with variety of ways like before line, after line, above Line, below line, inside box, below box, ball box. This varies with requirement and display available on screen. Many a times due to space allocate for input these requirements may vary. Data capture facility is always attached with captions.

- **List the different coding techniques in input design**

Information system projects are always based on time limit, cost conscious and teamwork. Looking to these requirements system analyst has to design project guideline in a way that reduce development time, cost and employee. For these types of settlements system analyst can look forward to coding methods that fulfill above-mentioned goals. A code can be brief number, title or symbol instead of lengthier or describe coding techniques. With code, fewer details are necessary in input, but no loss of information results. Various coding methods are adopted for these are:

- Classification codes
- Function codes
- Sequence codes
- Significant digits
- Mnemonic codes

1. Classification Code:

Classification codes place separate entities, such as events, people or objects into distinct groups called classes. A code is used to distinguish one class from another. The code is recorded on the source document by the user or, in an online system. The user classifies the event into one of several possible categories and records the code.

For example, College management system can be classified by students, faculty or course. In this system, to maintain library information system can be classified using books issue or author to check book of interest.

2. Functional Code:

Function codes state the activities or work to be performed without spelling out all the detail n narrative statements. User use this type of coding method to process the data. The particular function code may determine the contents of the input record whether data in the code are keyed or scanned. For example to process today‘s total withdrawal amount in a bank analyst can design function, which accepts today‘s transaction with complete details of the transaction and data. A derived function can evaluate amount using input data. Another function can be derived to delete account information with different input values. For deletion of account, identical account number has to pass and rest function can be performed by function code.

3. Sequence Code:

Sequence codes are either numbers or letters assigned in series. For example roll no is a sequence in a class. Another example, a banking system must be able to keep track of the order of transactions so that it’s clear which transaction to process first, which second and so on. Therefore, a sequence number should be specified in the design to order the transactions.

Sequence codes are also used for identification purpose but are assigned in the order in which customers enter the system.

4. Significant Digit – Subset Code:

Suppose item number will be assigned to the different materials and products firm stocks or sells. One way to accomplish this is to assign number in sequence. Starting with the first and going through to the last. Or a prefix can be added to the identification numbers to further describe the type of item: steel has an S-prefix plastic a P and so on. The codes can be divided into subset or sub codes, characters that are part of the identification number and that have special meaning. The sub codes give the user additional information about the item.

5. Mnemonic Code:

Mnemonic does use letters and symbols from the product to describe it in a way that Communicates visually. For example, to describe a 21-inch color television set, useful code is TV-CL-21 (black and white is TV-BW-21). Universities frequently use mnemonic to code information: BIT (Bachelor of Information Technology) in general data and transaction coding reduces the volume of data for input and simplifies the process. Code selection depends on the nature of data and objectives of the analyst.

Output Interface Design:

The output applies to any information produced by an information system, whether printed or displayed. When analysts design computer output they first identify specific output needs of the system user. So methods of output vary across system. For some, such as an inventory report of the quantity of sale, the computer system, under program control, simply retrieves the data on hand from storage and assembles them into a presentable form. This type of output may be retrieved on printed form as per customer requirement, as stated above. Whether the output is a formatted report of a simple listing of the contents of a file a computer process will produce the output. System output may be,

1. A report
2. A document
3. A message

Depending on the circumstances and the contents, the output may be displayed or printed. Output contents originate from these sources:

1. Retrieval from a data store
2. Transmission from a process or system activity
3. Directly from an input source

Method of output suite to client flavor can be decided from asking few questions to the user of the system. Analyst can retrieve information on asking, who will receive the output? Answer of this question derives type of information and method of information. Analyst further checks for method of output by asking, by what method output is required? Should the output be printed or displayed? And rest depends on the user of the system for system method of output. There are three different sources of output. It may be the result of process or it may come from file or it may be direct input. The output presentation design basically concentrates on presenting more using less space.

There are various different formats to present the output data.

1. Tabular Format
2. Graphical Format
3. Use of Icons
4. Colored Presentation

The direct source of information for user is computer output. Effective, efficient and well-organized output can be more meaningful for user to operate information system. Below list of consideration has to be taken in any system output…

(1) Always give proper title output

(2) Specify various finds with boundaries or highlighted areas.

(3) If possible, make your output interactive

(4) Provide facility for online editing in output.

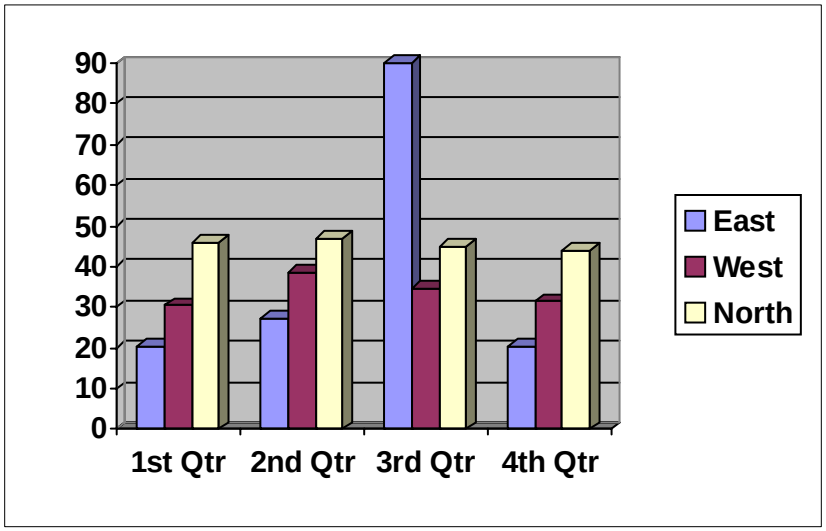
- **TABULAR FORMAT:**

The financial reports, the accounting details etc are generally presented in tabular format. Here one table is prepared showing various items row and column wise. The column shows the various categories of data. The data can be easily added in this tabular Format. Sometime data can also be sorted in table to get important data at the top. Thus, the Tabular Format enhances the readability of the data and more detail can be included in small space. Again at the end of table summaries and total gives extra understanding of output. For example,

Roll No	Name	Sub1	Sub2	Sub3	Sub4	Total

- **GRAPHICAL FORMAT:**

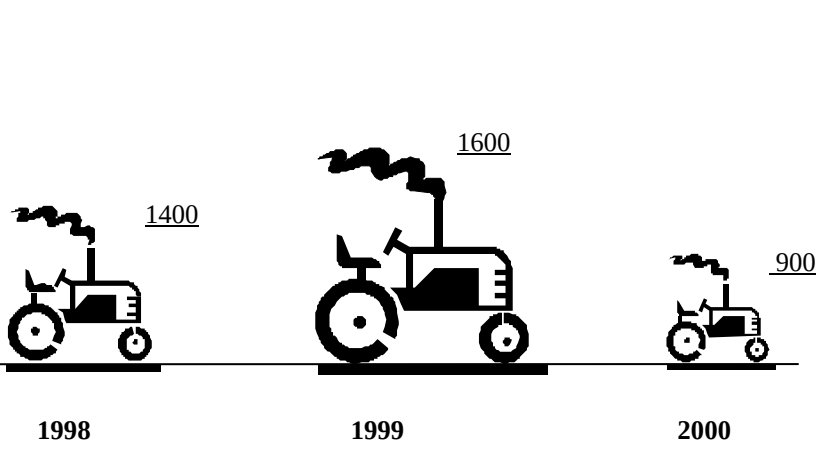
Now a day graphical technology has been improved. Thus it's very easy to present output in interactive and graphical way. High quality charts and diagrams can be easily created based on some data. For more understanding animation can also be used. There are various types of business graphics which are used now a day like Pie chart, Line-chart, bar chart, column-chart, map etc. as drawn below.



- **USE OF ICONS:**

Icon is a symbol used to describe about the data. For example picture of person can be used to report the no. Of student passed every year in BIT examination as given below. Thus, one can easily say that the no. Of student passed in 1999 is 1600 which is maximum.

Figure:



- **COLOURED PRESENTATION:**

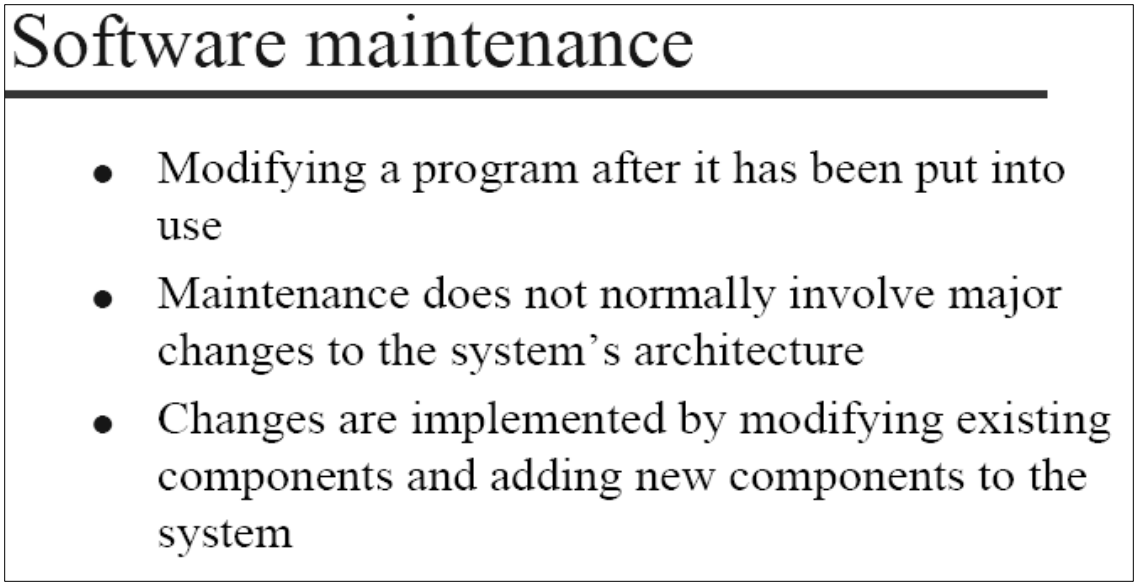
Sometime for more clarification of data and graph different colors are used to identify them easily. For highlighting some part of data again colors are used. The direct source of information for user is computer output. Effective, efficient and well-organized output can be more meaningful for user to operate information system. The major form of output type is printout (flare-copy). The various types of printers are from cheap to costly. The other important output media is CRT monitor. The following media devices are capable of giving computer-based output.

(1) MISR Recorder
(2) Dot Matrix printer
(3) Inkjet Laser printer
(4) Computer output micro-file
(5) CRT Screen Display
(6) Graph plotters
(7) Audio Response

Some time various different types of O/P forms are combined for proper and good combination. Some skills required.

(1) Always give proper title output
(2) Specify various finds with boundaries or highlighted areas.
(3) If possible make your output interactive.
(4) Provide facility for online editing in output.

- : Software Maintenance: -



Maintenance is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are to remain useful in an environment

There are four different types of software maintenance.

1. **Corrective Maintenance (Maintenance to require software fault).**

Codding error are usally relatively cheap to correct, design error are more expensive as they many involve rewriting some program components. Requirements error are most expensive system redesign that may be neccssory.

2. Adaptive Maintenance (Maintenance to adapt to software to the differenent government policies or operating environement).

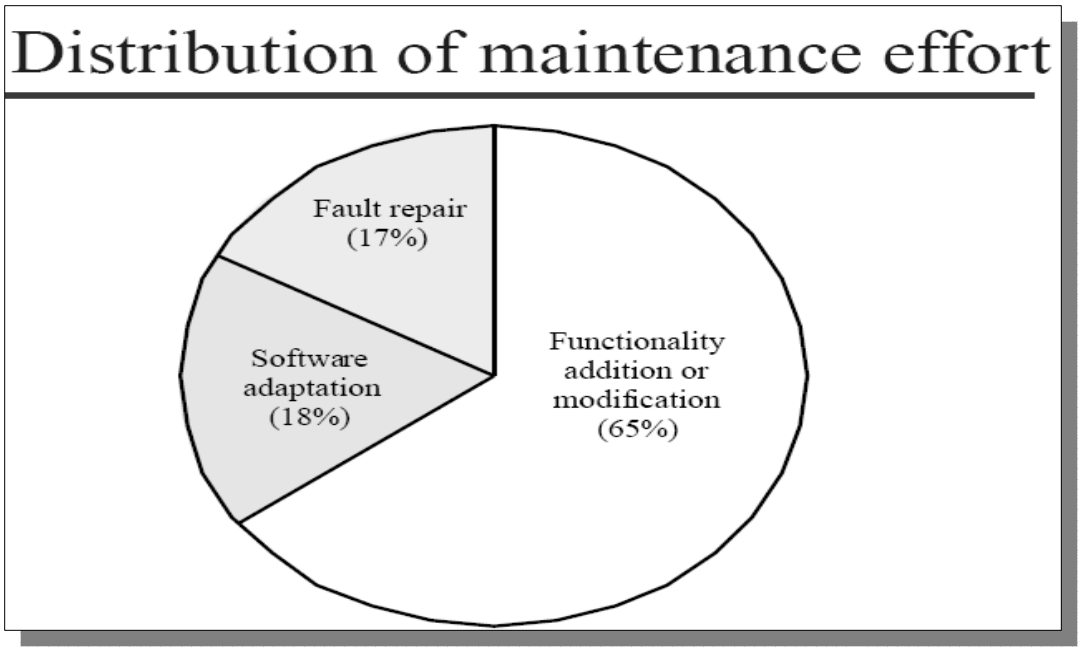
This type of maintenance is required when some aspect of the systems environment such as the hardware, the platform of operating system or other support software changes. The application system must be modified to adapt it.

2. **Perfective Maintenance (Maintenance to add or modify the system functionality).**

This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

3. Preventive Maintenance:

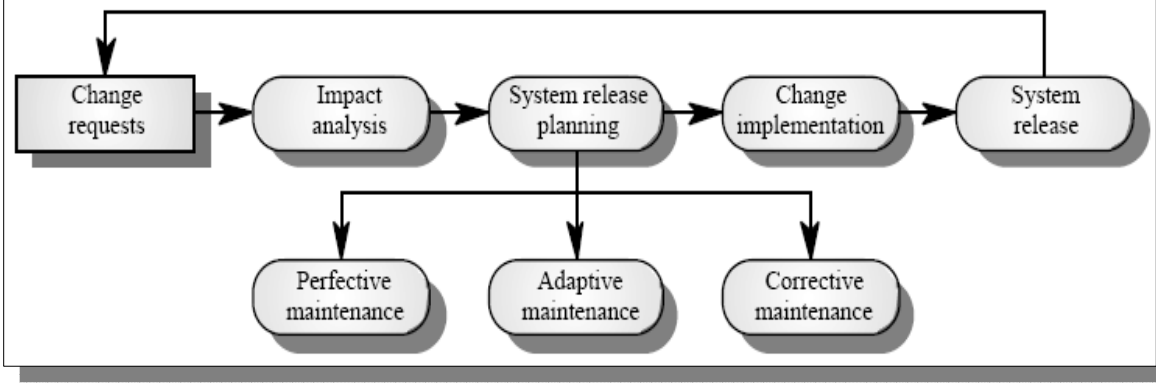
Maintenance to finds as actual or potential fault that has not yet becomr a failure & take action to correct the fault before damage is done.



Maintenance cost factors

- Team stability
  - Maintenance costs are reduced if the same staff are involved with them for some time
- Contractual responsibility
  - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change
- Staff skills
  - Maintenance staff are often inexperienced and have limited domain knowledge
- Program age and structure
  - As programs age, their structure is degraded and they become harder to understand and change

The maintenance process



Change requests

- Change requests are requests for system changes from users, customers or management
- In principle, all change requests should be carefully analysed as part of the maintenance process and then implemented
- In practice, some change requests must be implemented urgently
  - Fault repair
  - Changes to the system's environment
  - Urgently required business changes

System re-engineering

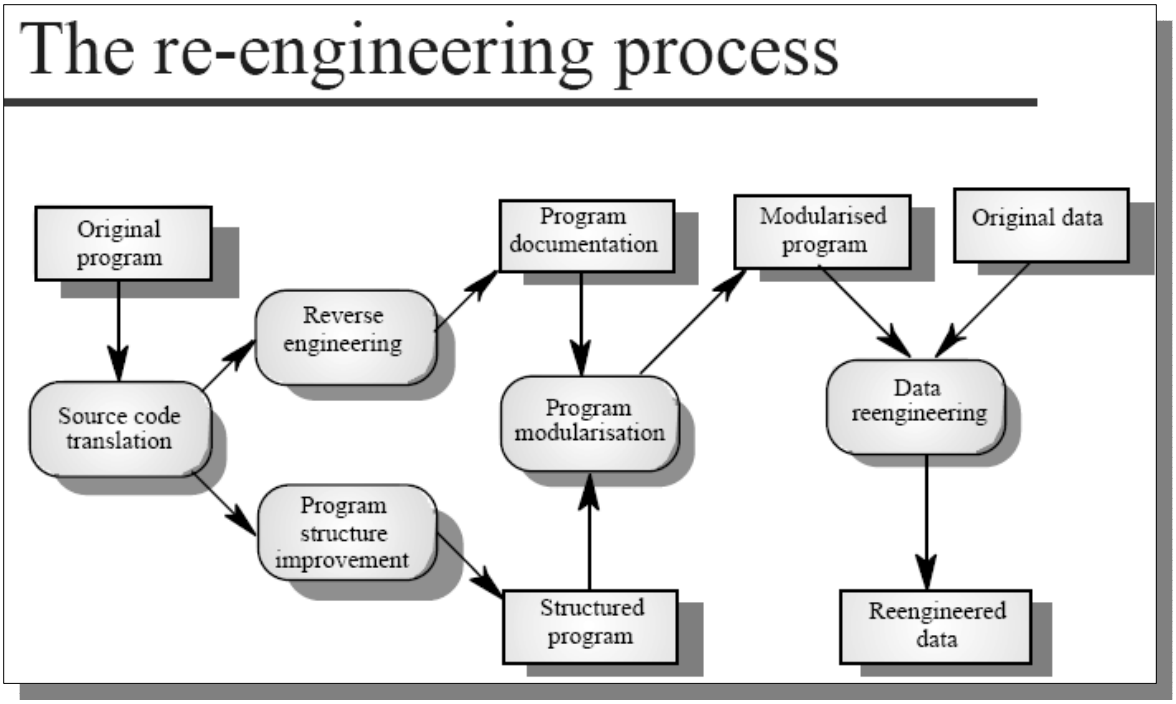
- Re-structuring or re-writing part or all of a legacy system without changing its functionality
- Applicable where some but not all sub-systems of a larger system require frequent maintenance
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

When to re-engineer

- When system changes are mostly confined to part of the system then re-engineer that part
- When hardware or software support becomes obsolete
- When tools to support re-structuring are available

Re-engineering advantages

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software



❖ **Estimation of maintenance cost:**

We had earlier known that maintenance efforts required about 60% of the total cycle cost for a software product. However maintenance cost very widely from one application domain to other for embedded system, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm, 1981 proposed a formula for estimating maintenance cost as part of his COCOMO cost estimation model. Boehm maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic [ACT]. Boehm defines ACT as the fraction of a software products source instruction which undergo change during a year either through addition deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{delete}}{KLOC_{total}}$$

Where:
KLOC added is the total kilo lines of source code added during maintenance. KLOC deleted is the total deleted lines during maintenance. Thus, the code that is changed should be counted in both the code added & for deleted. The ACT is

multiplied with the total development cost to arrive at maintenance cost for example.

Maintenance Cost = ACT * Development Cost

Most maintenance estimation mode, however give only approximate results because they do not take into account. Some factors such as experience levels of the engineers and know about the engineers with the product, hardware requirement, software complicity.

==========
****