SOFTWARE ENGINEERING

NOTES

SERIES -1

LinkedIn ATUL KUMAR
7777888822

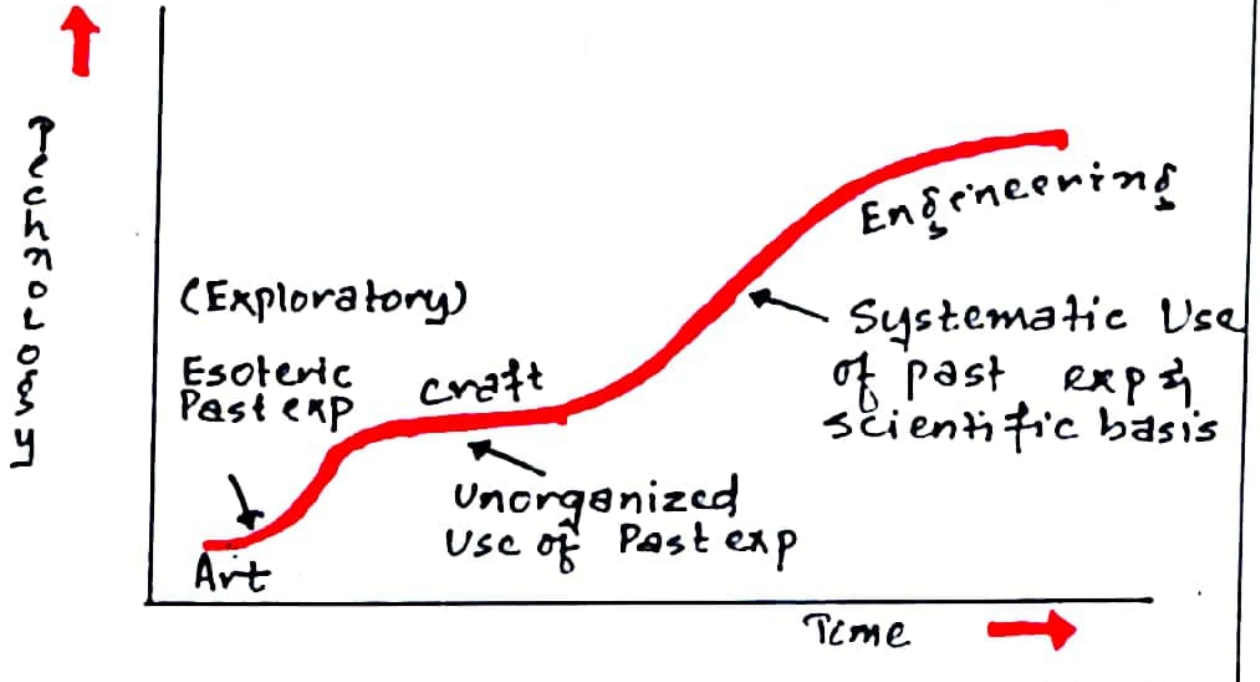
CHAPTER -1 INTRODUCTION

- What is Software Engineering?
- Program Vs S/W Products?
- Why study software Engg?
- Evolution.
- Notable changes in Practices?
- Computer Systems Engineering - Introductn to Life-cycle models.
- Summary.

WHAT IS SOFTWARE ENGINEERING?

- Engineering approach to devlop S/W. (Building construction analogy)
- Systematic collectn of past experience, techniques, methodologies & guidelines.

- Technology development pattern.



• Software crisis - solution

✔ Software products
- fail to meet user requirements.
- frequently crash.
- are expensive.
- difficult to alter, debug & enhance.
- are often delivered late.
- Uses resources non-optimally.

✔ Reason behind the crisis
- Larger Problems
- Lack of adequate training in s/w engg.
- Increasing skill shortage.
- Low productivity improvements.

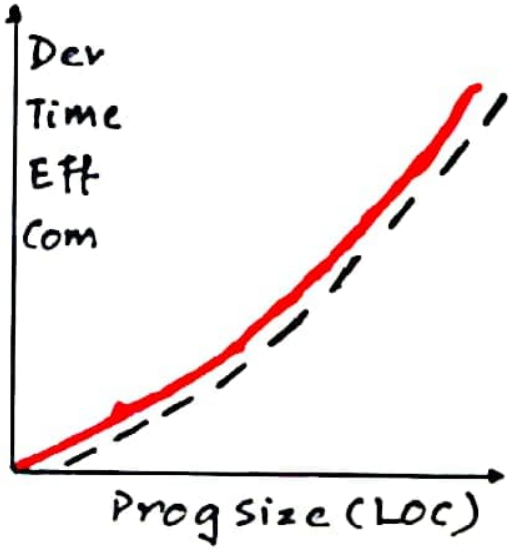
PROGRAM VS SOFTWARE PRODUCT?

→ Usually small in size	→ Large
→ Author is the user	→ Large no of users
→ single developer	→ Team of dev's.
→ Lacks proper user interface.	→ well-designed interface
→ Lacks proper documentation.	→ well-documented & user-manual prepared.
→ Ad-hoc development	→ Systematic development

" An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. "

WHY STUDY SOFTWARE ENGINEERING?

- To acquire skills to develop large programs.
- Ability to solve complex programming problems.
- Learn standardized techniques.
- To acquire skills to be a better programmer.



EMERGENCE OF SOFTWARE ENGINEERING

- Early computer programming. (50's)
  - programs were written in assembly language.
  - Limited to a few hundred lines of assembly code.
  - Every programmer developed his own style of writing programs (intution) also called exploratory programming.
- High-level language (Early 60's)
  - High level language such as FORTRAN, COBOL were introduced which reduced software developments efforts greatly.
  - Development style was still explora-tory. Typical program sizes limited to a few thousand lines of source code.

- control- flow Based Design

- size and complexity of programs increased further and exploratory style proved to be insufficient.

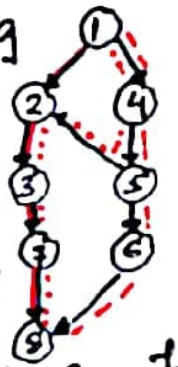
- programmers found it difficult to write correct programs and understand programs written by others.

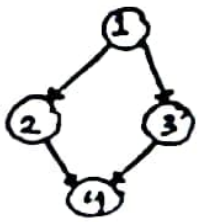
- To cope up with the problem it was advised to pay particular attention to the design of the program's ctrl. structure.

- A programs control structure indicates the sequence in which the programs instruction are executed.

- This can be achieved using flow-charting technique, where one can represent and design a program's ctrl. structure.

- But a program having a messy flow-chart-representation is difficult to understand and debug





- GOTO statements were found to make the control structure messy and alter the flow arbitrarily. So there was need to restrict GOTO

- Soon it was proved that only 3 constructs were sufficient to express any programming logic.

- This formed the basis of structured programming, where a program uses only 3 types of constructs.
  - * sequence (a = 0; b = b++;)
  - * select^n (if (c = true) k = 5 else m = 5;)
  - * iterat^n (while (a > 0) a = a - 1;)

- Reasons to use structured programming
* Unstructured control flows are avoided
* Consist of neat set of modules.
* Use single entry, exit constructs.
* Easier to maintain, read & understand.
* Require less effort & time
* commit less no of errors.
* sometimes premature loop exit is allowed to support exception handling.

- Data structure Oriented Design

Eventually it was discovered that it is important to pay more attention to the design of data structure than its control structure

Ex: Jackson Structured Programming.

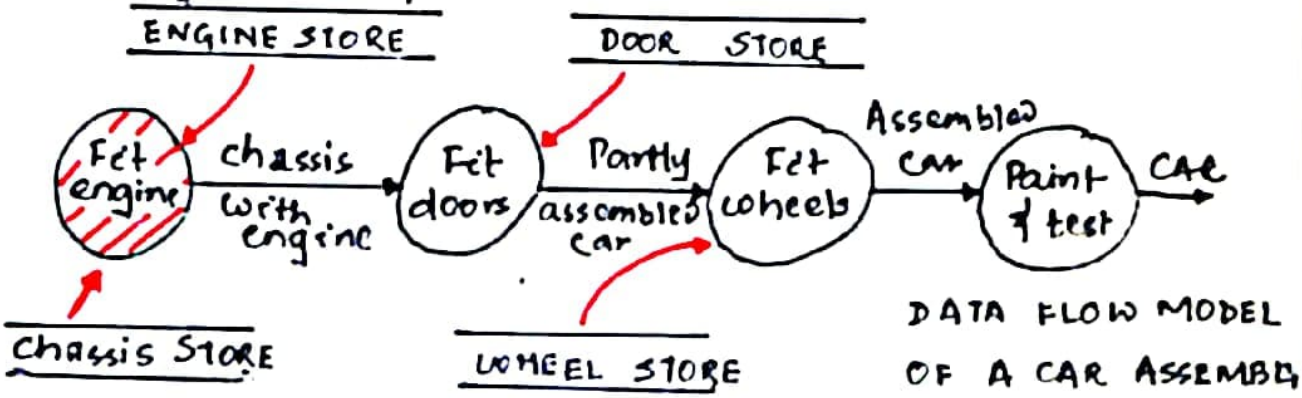
But these techniques are rarely used.

- Data Flow Oriented Design

- Here the data items input to a system must first be identified.

- processing required should be determined.

- The processes and the data items that are exchanged between different processes are represented using a Data Flow Diagram or a DFD.
- Adv is can be used to model the working of any system not just S/w System is very simple.



DATA FLOW MODEL OF A CAR ASSEMBLY PLANT

- **Object Oriented Design**

- Here the natural objects are first identified and then relationship among objects such as inheritance are determined.
- widely used b'coz of- there simplicity, Reuse possibilities, lower dev cost, and robustness and easier maintenance.

- **Evolution**

Adhoc exploratory ➡ control-flow based ➡ Data structure based ➡ Data Flow based ➡ Object oriented based

Evolution of software design techniques

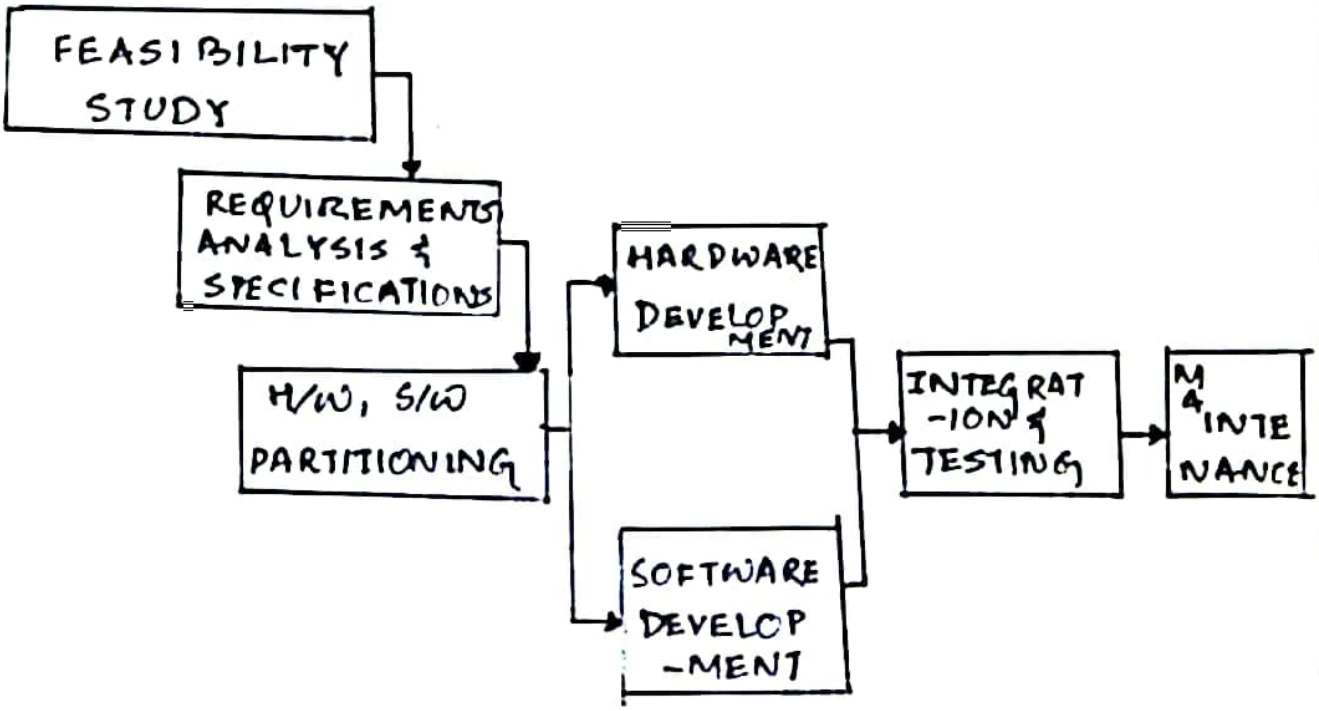
NOTABLE CHANGES IN S/W DEVELOPMENT Practices

i,e between exploratory & modern practices.

- Emphasis has shifted from error-correctn to error-preventn. or detectn of errors as close to their point of introduction, as possible.
- In exploratory style coding valsn is synonymous with program development, but now coding is considered only a small part of the whole effort.
- A lot of attentn is being paid to requirements specificatn and we have a distinct design phase.
- Periodic reviews are being carried out and systematic testing techniques are being used.
- There is better visibility of design & code. visibility means production of good quality, consistent and st. andard documents. Earlier very little attention was given to this.
- B'coz. of good documentation fault diagnosis and maintenance are smother
- Projects are being thorough planned through scheduling, estimation and monitoring mechanisms.
- Nowadays CASE tools are also being used.

COMPUTER SYSTEM ENGINEERING:-

addresses development of such systems that require development of both s/w and specific h/w to run the system.



Computer Systems Engineering..

Important Questions:

(1) Difference betn program and product.

(2) Major differences between exploratory style and modern techniques.

(3) Describe the evolution of software design techniques.

(4) What is computer systems engineering? How is it different from s/w engineering.

(5) what are symptoms of the s/w crisis and reasons behind it..

CHAPTER- 2

SOFTWARE LIFE CYCLE MODELS..

- Introduction
- Why use a LIFE Cycle model?
- Classical waterfall model?
- Iterative waterfall model?
- Prototyping model?
- Evolutionary model?
- Spiral model?
- Comparision of different Life-cycle models?

INTRODUCTION.

A software Life cycle model or process model is a :

- descriptive and diagrammatic model of software Life cycle.
- identifies all the activities required for product development.
- captures the order in wh these activities are to be undertaken.
- Divides the life-cycle into phases, where where several different activities may be carried out in each phase.

* A process covers all the activities beginning from product inception through delivery and retirement, but a methodology covers only a single activity or a few individual individual steps in the development.

WHY USE A LIFE CYCLE MODEL?

Primary advantage of having a life-cycle model is it helps in development of S/W in a systematic and disciplined manner.

- when a program is developed by a single programmer, he has the freedom to decide his exact steps.
- when the product is being developed by a team, there must be a precise understanding among team members as to "who to do what". Otherwise it would lead to chaos and project failure.

• Why document a life-cycle model?

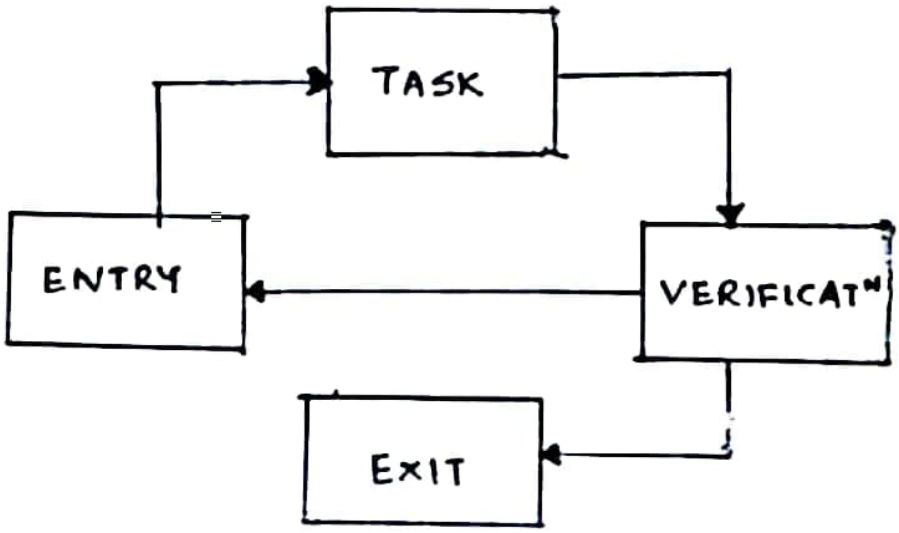
- Forms a common understanding of activities among the S/W developers.
- Helps in identifying inconsistencies, redundancies and omissions in the development process.
- Helps in tailoring a process model for specific projects.
- Most of the times, it's a mandatory requirement.

• Phase Entry & Exit Criteria..

- A life-cycle model defines entry & exit criteria for every phase
- A phase is considered to be complete only when all its exit criteria are satisfied

- A phase can start only if its phase-entry criteria have been satisfied.

- Ex: The phase criteria for the SRS phase is SRS document is complete, reviewed & approved by the customer.



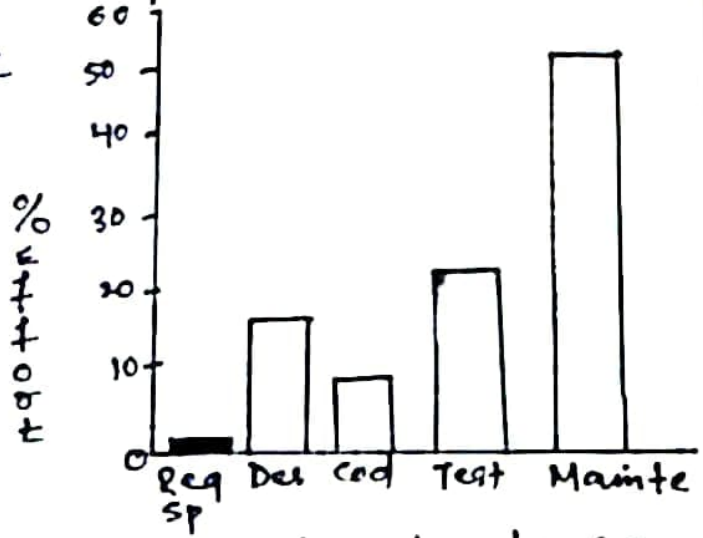
- with life-cycle model it becomes easier for project managers to monitor, the progress of the project.
- A project manager can at any time fairly accurately tell, at which stage of the life cycle, the project is. Other wise the project-manager would have to depend on the guesses of the team members, which usually leads to the problem known as 99% complete syndrome. Here the optimistic members feel that the project is 99% complete, when actually, the project is far from complete.

CLASSICAL WATERFALL MODEL

Feasibility Study
↓
Requirements Analysis & specs
↓
Design
↓
Coding & Unit testing
↓
Integration & Sys testing
↓
Maintenance

- Relative Effort for phases

- Phases between feasibility study & testing are known as development phase
- Among all life-cycle phases, maintenance takes maximum effort
- Among development-phases, testing takes the maximum effort.



- Most organizations usually standards on the outputs produced at the end of every phase.
- Entry & exit criteria are also defined.
- Specific methodologies are defined for each phase, which alongwith the guidelines are called organizations SOFTWARE DEVELOPMENT METHODOLOGY..

• Feasibility study

-Main aim is to determine technically & financially feasible projects.

- Understand the i/p, o/p, processing actions and constraints of the system.

- with the collected data we have analyzed, we can do the

✓ An abstract problem- difinition is a rough description of the problem,

✓ Formulation of different solution strate -gies.

✓ Analysis of alternative solutions to compare their benefits and shortcomings.

ITERATIVE WATERFALL MODEL

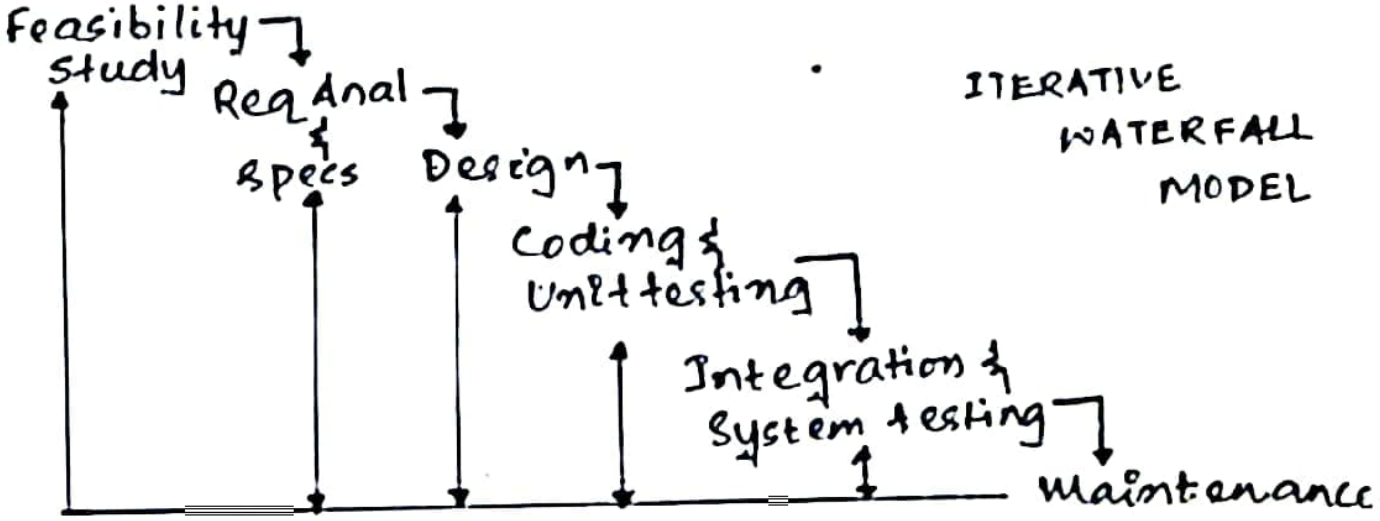
• Classical waterfall model assumes that

- No defect is introduced during any development activity, but in practice defects do get introduced in almost every phase of the life-cycle.

- Defects usually get detected much later in the life cycle.

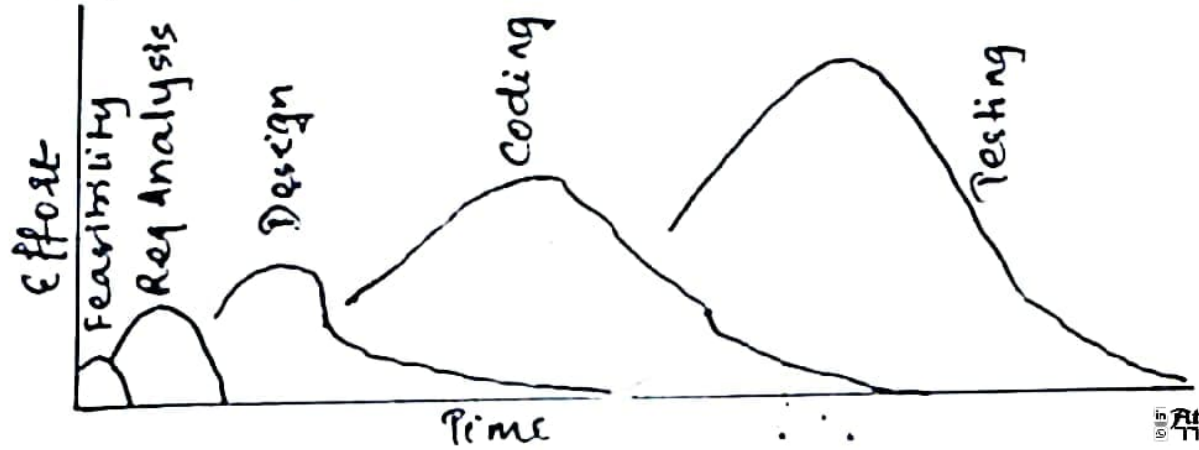
- Once a defect is detected, we need to go back to the phase where it was introduced. Redo some of the work in that phase and the subsequent phases.

c'o we need feedback paths in the classical waterfall model.



ITERATIVE WATERFALL MODEL

- Errors should be detected in the same phase in which they are introduced. For e.g. if a design problem is detected in the design phase itself, the prob can be taken care of much more easily.
- The principle of detecting errors as close to its point of introduction as possible is called PHASE CONTAINMENT OF ERRORS. This can be achieved by conducting reviews after every milestone.
- Almost all other models are derived from this model but the shortcomings are it assumes that req can be completely specified & adheres to strict phases.



PROTOTYPING MODEL

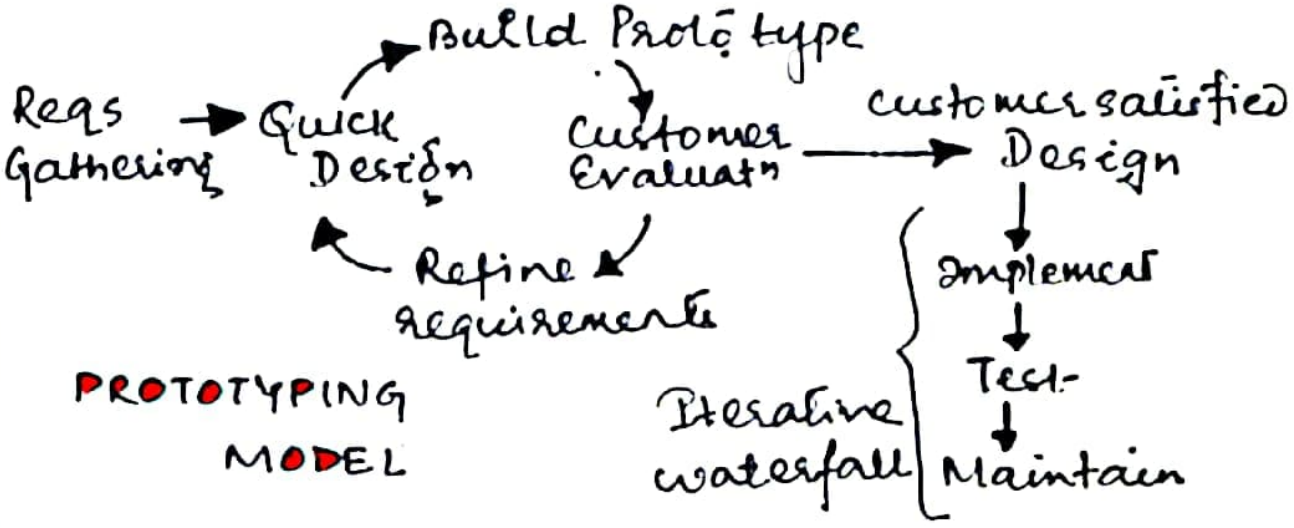
- A prototype is a toy implementation of a system with limited functional capabilities, low reliability and inefficient performance.
- B'fore starting actual development, a working prototype of the system should be built.

• Reasons for developing a prototype.

- Illustrate to the customer i/p data formats, messages, reports or interactive dialogs
- Examine tech issues like efficiency of a sorting algorithm, response time etc associated with hw development.
- It is impossible to get it right the 1st time.

• Process

- start with approx requirements
- carry out a quick design.
- Is done using several short-cuts, dummy functions and look-up tables.
- The developed prototype is submitted to the customer for evaluation and based on his feedback reqs are refined.
- This continues until the user approves the prototype...



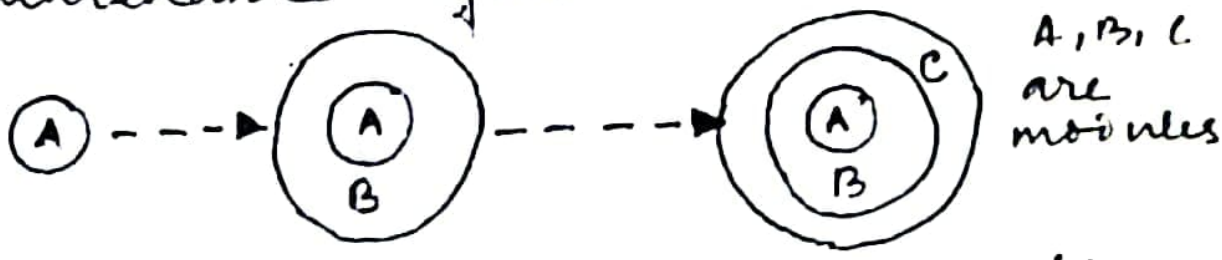
- Here Req analysis & specs become redundant
- Design & code of prototype is thrown away but the experience helps.
- Even though a prototype may seem to be expensive, the overall cost is low for projects with unclear user reqs & technical issues.

• Evolutionary model

Here the S/w is first broken down into several functional units which can be incrementally constructed & delivered.

- Rough Requirements specification.
- Identify the core and other parts to be developed incrementally.
- Develop the core part using an iterative waterfall model.
- Collect customer feedback and modify requirements.

- Develop the next identified features using an iterative model.
- After all features are complete, maintenance goes on.

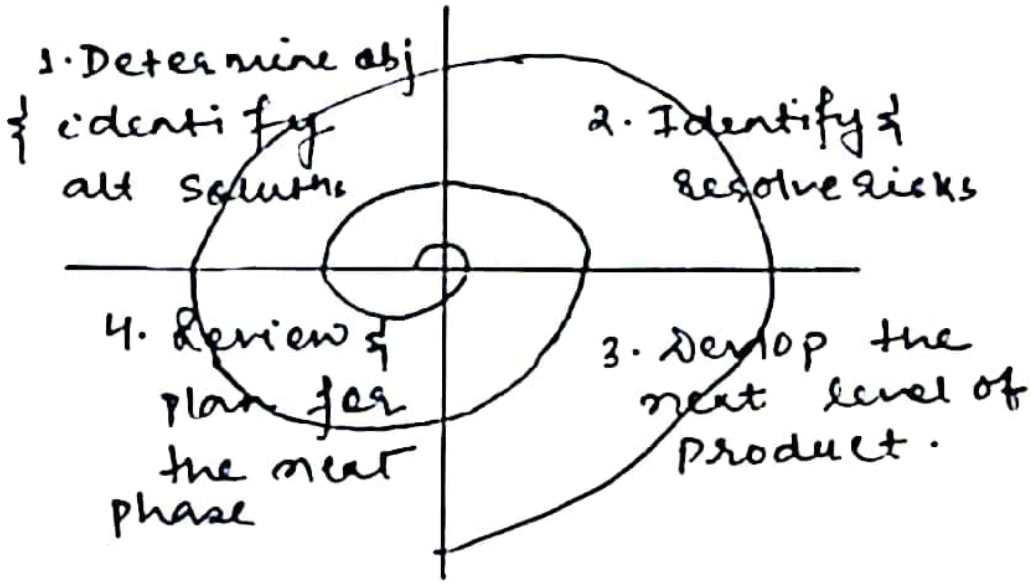


- **Advantages of Evolutionary model**
- Users get a chance to experiment with a partially developed system.
- Helps finding exact user requirements.
- Core modules get tested thoroughly
- Large resources are not required.

- **Disadvantage**
- For most practical problems it is difficult to divide problems into functional units

- **Useful**
- For large projects.
- Where customer prefers to receive the product in increments.
- Useful for - Object Oriented Software development projects b'coz it can be easily partitioned into stand-alone units.

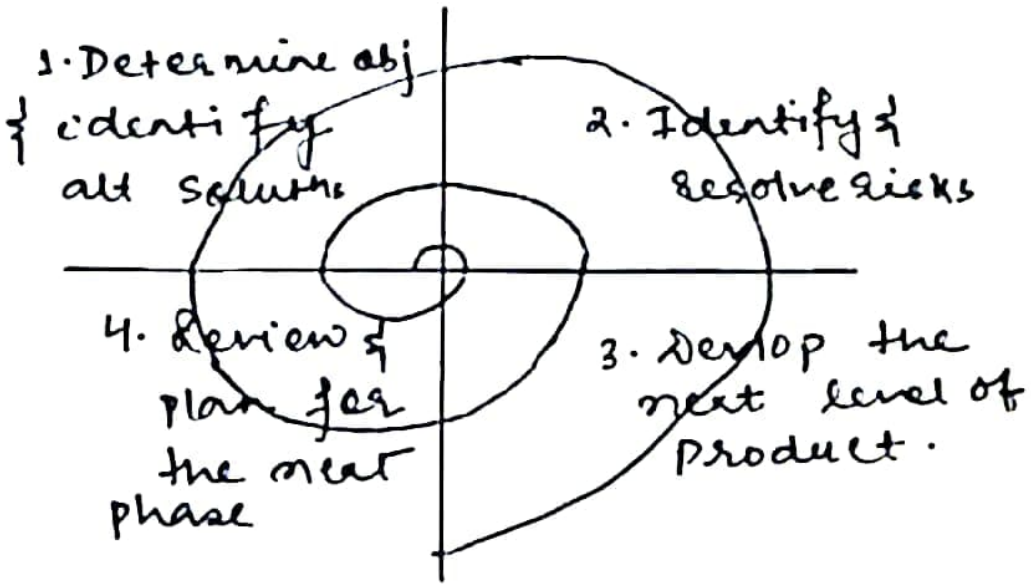
SPIRAL MODEL



- Each loop represents a phase of the S/w process
- Each loop in the spiral is split into four sectors (Quadrants).

- Spiral model as a meta model
  - A single loop spiral represents waterfall model.
  - Uses an evolutionary approach with successive loops.
  - Uses prototype as a risk reduction mechanism
  - Enables understanding and reacting to risks during each iteration.
  - with each iteration progressively more complete version of the S/w gets built.

SPIRAL MODEL



- Each loop represents a phase of the S/w process
- Each loop in the spiral is split into four sectors (Quadrants).

- Spiral model as a meta model
  - A single loop spiral represents waterfall model.
  - Uses an evolutionary approach with successive loops.
  - Uses prototype as a risk reduction mechanism
  - Enables understanding and reacting to risks during each iteration.
  - with each iteration progressively more complete version of the S/w gets built.

Comparision of different Life-Cycle Models

- **Iterative waterfall**
  - Most widely used.
  - But suitable only for well-understood problems.
- **Prototye model**
  - Suitable for projects not well-understood in terms of user requirements & technical aspects.
- **Evolutionary model is**
  - suitable for large problems which can be decomposed into a set of modules.
  - suitable for incremental delivery of the system.
- **Spiral Model:**
  - suitable for development of technically challenging & products that are subject to several kinds of risks.

Important Questions:

(1) what do you understand by a process? what is the difference between a process and a methodology. what problems will a firm face if it doesn't follow a systematic development process?

(2) what are the different life-cycle models? Compare them.

REQUIREMENT ANALYSIS & SPECIFICATION

- Introduction.
- Requirements Gathering & Analysis.
- Software Requirement- specificatn.
- Formal System Development Techniques.

INTRODUCTION

The goal of this phase is to clearly understand the customer requirements and to systematically organise the requirements into a specificatn document. This phase consists 2 activities:-

- Requirements gathering & analysis.
- Requirement Specificatns.

-The engineers who gather and analyze customer requirements and write the SRS are known as system analysts.

REQUIREMENT GATHERING & ANALYSIS

The two main activities involved in this phase are:-

(1) Reqs Gathering (2) Analysis.

- Requirements Gathering

- Analysts interview end-users and customers, study the existing documents to collect all possible information regarding the system. . .

• Analysis of Gathered Requirements

- The main purpose is to exactly understand the requirements of the customer. certain questions pertaining to the project should be clearly understood by the analyst. like

✓ what is the problem?
✓ why is it important to solve the problem?
✓ what are the possible solutions to the problem?
✓ what are the I/p data, complexities, external S/w or hardware?

- After understanding the exact reqs the analyst has to resolve the req problems.

- The most common requirement problems are anomaly, Inconsistency, Incompleteness

✓ Anomaly: Ambiguity in the interpretation of requirement.

✓ Inconsistency: If one of the requirements contradicts another.

✓ Incompleteness: some of the requirements have been overlooked.

After detecting these problems the analysts has to conduct further discussions and resolve the issues with the customer.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

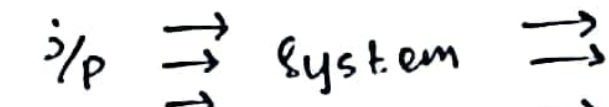
- **Different Categories of Users of SRS**

- Users, Customers, Marketing Personnel: The goal here is to ensure that the system will meet their needs.
- Sfw Developers: Ensure that they develop exactly what is required by the customer
- Test Engineers: ensure that the requirements are understandable from a functionality point of view.
- User documentation writers: understand the document well enough to be able to write user's manual.
- Project Managers: ensure they can plan the project in terms of cost, manpower and time.
- Maintenance Engineers: understand the functionality, so that they can plan modifications and enhancements.

The SRS document serves as a legal contract between customers & developers.

- **Contents of the SRS document.**
  - Functional Requirements.
  - Non-functional Requirements.
  - Goals of Implementation.

- Functional Requirements



The functional req in the SRS should clearly each function as a transformation of a set of i/p data to o/p data.

- Non-functional Requirements (Characteristics) Include aspects concerned with maintainability, portability, usability, reliability, accuracy, human-computer interface issues and constraints.

- Goals of Implementation
Include revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues. Here the qualitative issues are documented.

• Functional Requirements

First high-level requirements have to be identified which correspond to an instance of use of the system by the users. Here each high-level requirement might consist of several sub-requirement, where, there are series of interactions between the system and one or more of the users.

Interactⁿs betw user and the system in withdraw-cash high level functional Requiremer

Select withdraw cash — o/p user
↓
Display Acc type — sys op
↓
Enter option — user i/p
↓
Prompt for amount to be withdrawn → sys o/p
↓
Enter amount → user i/p
↓
Display balance → sys o/p

- How to identify the Functional Requirements?

can be identified either from an informal problem description document or from a conceptual understanding of the problem. For this the different types of users of the system might be identified and then the requirements from user's perspective.

- How to document the functional requirements?

A function can be specified by identifying the state at wh/ the data is to be i/p to the system, its i/p data domain, the o/p data domain and the type of processing to be carried.

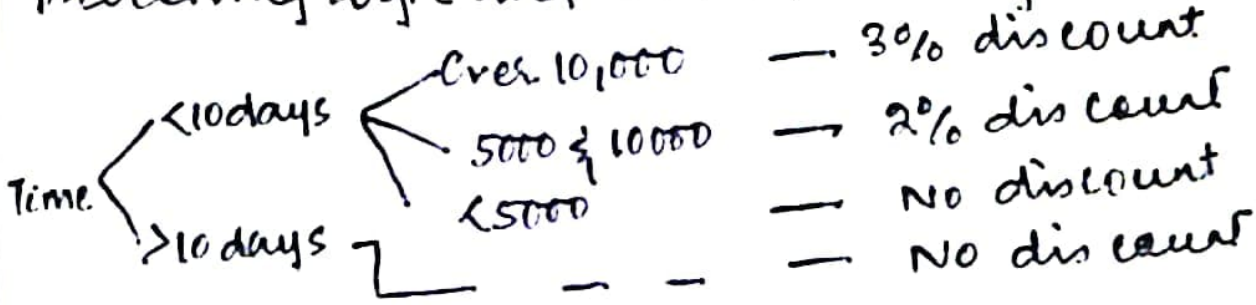
The user interactⁿ sequences may vary frm one invocation to another depending on some conditions. These different interactⁿ sequences capture the different scenarios.

(2) Goals of Implementation.
(3) Functional Requirements
(4) Non-functional Requirements
(5) Behavioural Description
   (a) System states
   (b) Events & Actions.

- Techniques for representing complex Logic.

  - Decision Tree
  - Decision Table

A decision tree gives a graphical view of the processing logic and corresponding actn's taken.



DECISION TREE

DECISION TABLE

CONDITIONS	DECISION RULES					
<10 days	Y	Y	Y	N	N	N
>10000	Y	N	N	Y	N	N
5000-10000	N	Y	N	N	Y	N
<5000	N		Y	N	N	Y
ACTION						
3% discount	X					
2% discount		X				
No discount			X	X	X	X

Ex: Withdraw Cash from ATM
R1: withdraw Cash
Descriptn:
R.1.1: Select- withdraw amt option
i/p: "withdraw amt" option
o/p: User promted to enter the acc type.
!
!
!

* Common dilema is to specify too little or specify too much.

• Traceability

-With traceability. it would be possible to tell wh/ design component corresponds to wh/ requirement, wh/ code component corresponds to which design component, and wh/ test corresponds to wh/ requirement

- Each function should be uniquely and consistently numbered to achieve traceability.
- Traceability ensures that all the requirements have been taken care of at in all phases. It also assess the impact of a requirements change. can be used to identify wh/ part of the design and code would be affected b'coz of a requirements change. Also used to study the impact of a bug on various requirements.

- Characteristics of a Good SRS document.

- Concise : complete, consistent, unambiguous.
- Structured : easy to understand & modify.
- Black-box View: specify the external, behaviour of the system, but not the implementation issues.
- Conceptual Integrity: easily understandable contents.
- Response to undesired events: systems response to exceptional conditions.
- Verifiable: all requirements shld be verifiable.

- Examples of Bad SRS document.

Apart from incompleteness, ambiguity and contradictions, the other problems are:-

- over specification when you try to poke into the implementation issues.
- Forward References shld be avoided.
- Wishful thinking are aspects wh/ are difficult to implement.

- Organization of the SRS document.

1. Introduction
   (a) Background
   (b) overall description
   (c) Environmental characteristics
      (i) Hardware
      (ii) Peripherals
      (iii) People

5. SOFTWARE DESIGN

(1) Introduction
(2) What is a Good software design?
(3) Cohesion and coupling.
(4) Neat Arrangement.
(5) Software Design approaches.
(6) Object Oriented .Vs. Function Oriented design.

INTRODUCTION

The objective of the ~~SRS~~ design phase document is to take the SRS document as input and produce the o/p mentioned below:

- Different modules required to implement the soln.
- Control Relationship among the different modules.
- Interface among different modules.
- Data structures of the individual modules.
- Algorithms required to implement the individual modules.

- The design activity is an iterative procedure broadly divided into two phases.
  - ✓ Preliminary (High-level design)
  - ✓ Detailed Design.

WHAT IS A GOOD SOFTWARE DESIGN?

The goodness of a design depends on the targeted application, and varies from s/w engineers to academicians. But there are certain characteristics agreed in common-

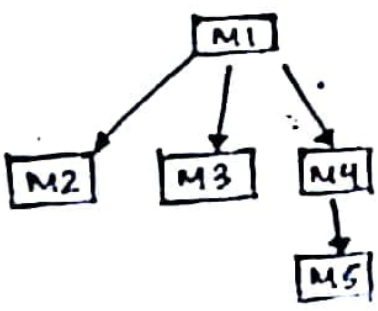
(a) Correctness : a design should correctly implement all the functionalities of the system
(b) Understandability : so that the s/w is easy to understand, change and develop.
(c) Efficient
(d) Maintainability: should accomodate change.

The ~~woodaclassidy~~ of a des understandability of a design should have following features:

- should use consistent and meaningful names for various design components.
- The design should be modular.
- Modules should be arranged neatly in a hierarchy.

• Modularity.
Decomposition of the problem into independent modules, reduces the complexity and makes the problem more understandable.

• clean Decomposition
The modules should display high cohesion and low coupling.



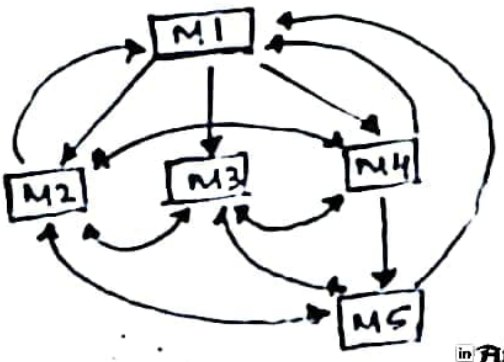


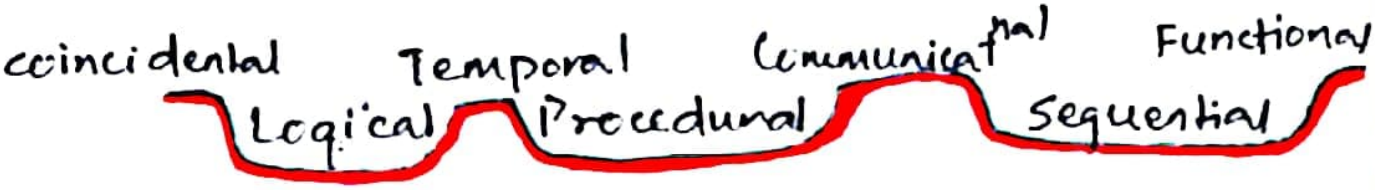
fig: Two design soln's for same problem.

COHESION & COUPLING.

A Good design is characterised by clean decomposition into modules & neat arrangement. The characteristic of a neat module decomposition are high cohesion and low coupling.

- cohesion is a measure of functional strength of a module.
- coupling is the degree of interdependance or interactn between two modules.
- A module having high cohesion & low coupling is said to be functionally independant
- Functional independence leads to a good design beco's:
  - ✓ Error Isolatn
  - ✓ Scope for reuse
  - ✓ Understandability

- classification of Cohesiveness



← low (Degree of Cohesiveness) high →

- coincidental : Tasks are loosely related.
- Logical : All elements perform similar operations eg error handling, data i/p
- Temporal : All the functions must be executed in the same time span eg:- funct's for initialize
- Procedural : set of functns are a part of a proc. - -dure or algorithm (Algo for decoding a message)
- Communicational : All the funct's of a module refer to or update the same data structure eg: set of functions defined on a array/stack.
- Sequential: The elements of a module form a part of the sequence, wh/ o/p from one becomes input to the next.
- Functional: different elements of a module acheive a single function. ex: emp payroll.

• classification of coupling.

The degree of coupling between two modules depends on their interface complexity, wh/ in turn is determined by the number & type of parameters that are interchanged between them.

- Data Coupling : Communication using an elementary data item that is passed as a parameter betn the modules (eg: float).
- stamp: Communicatn using composite data item. (eg: structure in C).

- control: If data from one module is used to direct the order of instruction executed in another.
- Common: If modules share some global data items.
- Content: If code is shared.

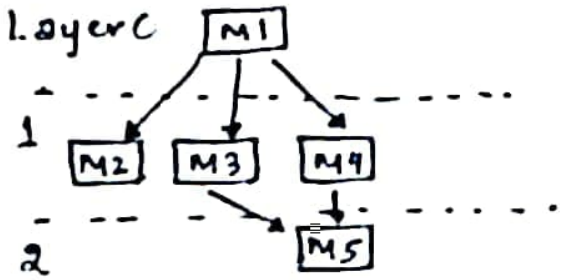
Data stamp Control Common Content

Low ——— coupling ——→ High

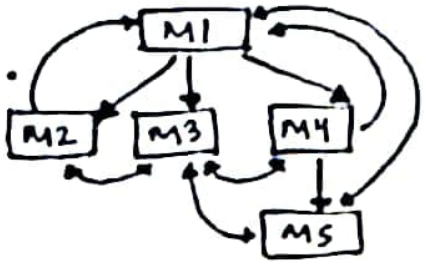
NEAT ARRANGEMENT

The control hierarchy represents the org^{n} of the program components. Its also called the program structure is represented in a tree-like diagram. The characteristics of such a design are:-

- Layering: The modules are arranged in layers. A module that calls another module is said to be superordinate to it. A module controlled by another module is called sub-ordinate. A module A is said to be visible to another module B, if A directly or indirectly calls B.
- Control Abstract^{n}: The modules at higher layer should not be visible to modules at lower layers.



Layered design with good ctrl abstraction



Layered design with poor ctrl abstraction.

- **Depth & Width** : No of levels of ctrl and span of control.
- **Fan-out**: The no of modules that are directly controlled by a given module.
- **Fan-in**: No of modules directly invoking a given module.

Low Fan-out and High fan-in are desirable

SOFTWARE DESIGN APPROACHES

- Function - Oriented Design
- Object - Oriented Design

• Function - oriented Design

(1) A system is used as something that performs a set of functions where each high-level function is successfully refined into detailed functns.

Ex:- create-new-library (function)
→ assign membership-no } sub-functns
→ create-mem-record
→ Print bill . . .

(2) The system state is centralized & shared among different functions.

e.g- Member record is used by create, delete, update.

Ex:- of FOA

→ Structured design → Warnier-Orr

→ Jackson's Struct design etc.

- Object-Oriented Design

System is viewed as a collection of objects where each object manages its own state information

OBJECT ORIENTED Vs FUNCTION ORIENTED

- Unlike FO, in OOD the basic abstraction are not functions like "display", "sort" etc., but real world entities such as emp, picture etc
- In OOD, S/w is not developed by designing functions, but by designing objects
- FOA, techniques, group functns together to constitute a higher-level function, but in OOD, the functions are grouped together on the basis of data they operate on

Questions:-

(1) Classificatn of Cohesion & Coupling.

(2) Diff between CC & FC Design.

FIRE ALARM SYSTEMS

6. FUNCTION ORIENTED SOFTWARE DESIGN

A system is viewed as a BLACK-BOX that performs a set of high-level functions, which are decomposed into more detailed functions & finally mapped in modules. This is also called top-down decomposition.

OVERVIEW OF SA/SD METHODOLOGY

This methodology consists of two distinct activities:

- Structured Analysis (SA)
- Structured Design (SD)

STRUCTURED ANALYSIS

The aim is to transform a textual problem description into a graphic model. Here the major processing tasks (functions) are analyzed, & the data flows among the processing tasks are represented graphically. SA is based on the following underlying principles.

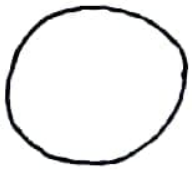
- Top down decomposition
- Divide & conquer principle. Each funct^n is decomposed independtly.
- Graphical representation of the analysis results using DFD's.

A DFD is a graphical model of a system that shows the different processing activities that the system performs & the data interchange

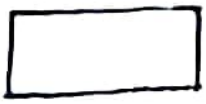
■ DATA-FLOW DIAGRAMS (DFDs)

DFD also known as Bubble-chart- is a graphical representat^n of a system in terms of i/p to the system, processing & o/p generated by the system.

- Symbols used for constructing DFD's.

Function Symbol (Process) ➡ 

Data Flow ➡ 

External Entity (source/sinx) ➡ 

Data store (Logical file) ➡ 

O/p hard copy ➡ 

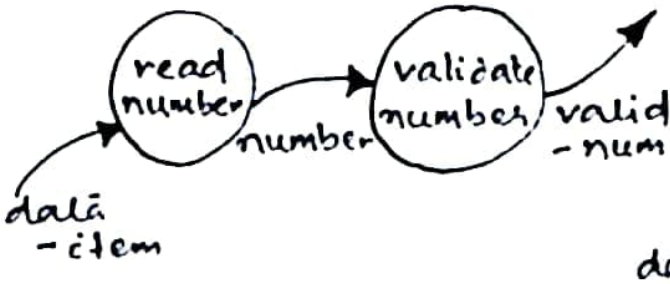
A DFD model
- uses limited type of symbols.
- A simple set of rules.
- Easy to understand & use, b'cos of its hierarchical representation.

- Concepts associated with designing DFD's.

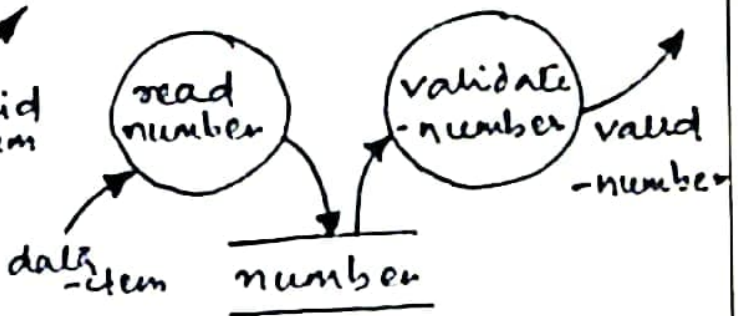
◆ Synchronous & asynchronous operations.

If two bubbles are directly connected by a data flow arrow, they are synchronous.
If two bubbles are indirectly connected i.e connected through a data store in called asynchronous.

(a) SYNCHRONOUS OPERATN



(b) ASYNCHRONOUS OPERATN



◆ Data Dictionary

A Data dictionary lists all the data-items, their purpose and the definition of all composite data items (alongwith components), appearing in a DFD Model.

CA:- Composite data item

Gross Pay = Regular Pay + Overtime Pay

• Importance of a Data dictionary

- Provides a standard terminology for all users. A consistent vocabulary reduces confusion.
- Provides the analyst with a means to determine the definition of different data structures in terms of their components.

For large projects, CASE tools can be used to generate Data Dictionaries.

◆ Data Definition

composite data items can be defined in terms of primitive data items using the following operators

+: a+b represents data a and b.

[,,]: [a,b] represents either a occurs or b occurs.

( ): a+(b) - either 'a' occurs or 'a+b' occurs.

{ }: represents iterative data definition

{name}5 - five name data

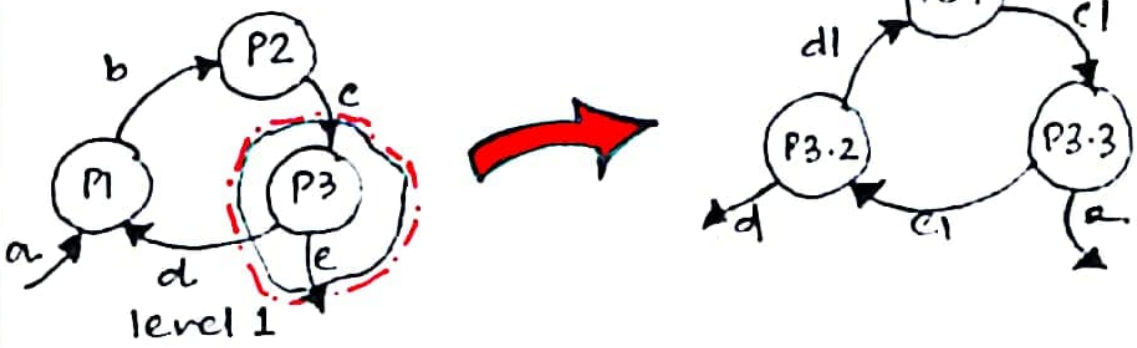
{name}* - zero or more instances.

=: equivalence, a=b+c means 'a' represents 'b' and 'c'.

/* */: comment.

◆ Balancing DFD's..

The data that flow into or out of a bubble must match the data flow at the next level.



In the level 1 of the DFD, data item 'c' flows into the bubble P3 and the data item d and e flow out.

In the next level bubble P3 is decomposed, the decomposition is balanced as data item 'c' flows into level 2 diagram and d and e flow out.

• Developing the DFD of a system.

To develop the DFD, first the most abstract representation of the problem (also called the context diagram) is worked out. Then the higher level DFD's are developed.

◆ Context Diagram (Level 0 DFD)

→ Represents the entire system with a single bubble.

→ External entities with wh/ the system interacts and the data flows between them are represented.

→ The bubble is annoted with the name of the software system.

◆ Level 1 DFD

Here the context bubble is decomposed into 3 to 7 high-level functional reqs, wh/ can be represented as bubbles in level 1. If the system has more than 7 high-level requirements, then some of the related requirements should be combined into 1, and decomposed later. If a sys has <3 to 7 high level requirements, then some of them need to be split, so that there are 5 to 7 bubbles on the diagram.

• Decomposition

Decomposition of a bubble into sub-functns at successive levels is known as factoring or exploding a bubble. Systematic development of a DFD model.

(1) The SRS is examined to determine high-level functions, data i/p, o/p & interactions.

(2) The high level functions are decomposed.

(3) Different sub-functions are identified data i/p, o/p and interactions are also identified and represented diagramatically.

(4) Step (3) is repeated recursively until a sub-function can be represented using a simple algorithm.

• Numbering of bubbles

- The bubble at the context level is numbered '0'.
- When a bubble numbered 'x' is decomposed its children bubble are numbered x.1, x.2, x.3..

• Commonly made errors while constructing a DFD model.

- Drawing more than one bubble in the context diagram.
- External entities should not appear anywhere except the context diagram.
- Only 3 to 7 bubbles per diagram should be allowed.

- DFD should not be left unbalanced.
- control information shouldnot be included in a DFD.
- A data store should not be connected to another data store or external entities.
- No funct^n specified in SRS shld be overlooked.
- Designer shldn't assume any functionality.
- Data and function names shld be intuitive.

- **Shortcomings of the DFD Model.**

- DFD's are sometimes imprecise. A shortlabel maynot capture the entire functionality of a bubble.

- Control aspects are not defined by a DFD. e.g: order of execution of different bubbles.

- The method and level of decomposition is not specified.

- **Extending DFD to Real-time Systems**

Here explicit representation of control and event flow aspects are essential. This can be done using extra processes that handle control flows, or a separate control flow diagram could be introduced. The CSPEC describe the following:

- The effect of an external event or ctrl signal.
- The processes that are invoked as a consequence of an event.

STRUCTURED DESIGN

The main aim is to transform the results of the structured analysis into a structure chart. A structure chart represents-
- The s/w architecture (various modules)
- Module dependency.
- The parameters that are passed.

- **Notations in a structure chart:**
  - Rectangular box : module
  - Module Invocation arrows : dependancy.
  - Data flow arrows.
  - Library modules: Rect with double edges
  - Selection ◇
  - Repitition ↺

In any structure chart-
- There should be one & only one module at the top called the root.
- There shld be atmost 1 ctrl Relationship between two modules.

- **Flow Chart Vs Structure Chart-**
  - It is difficult to identify the i/o modules in flow chart.
  - Data interchange among different modules is not represented in Flow chart.
  - Sequential ordering of tasks with is shown in a flow chart is suppressed in a structure chart.

TRANSFORMATION OF DFD INTO A STRUCTURE CHART

→ Transform analysis
→ Transaction analysis

- If all the data flows into the diagram are processed in similar ways then transform analysis is applicable, else transaction " is applicable.

• Transform Analysis

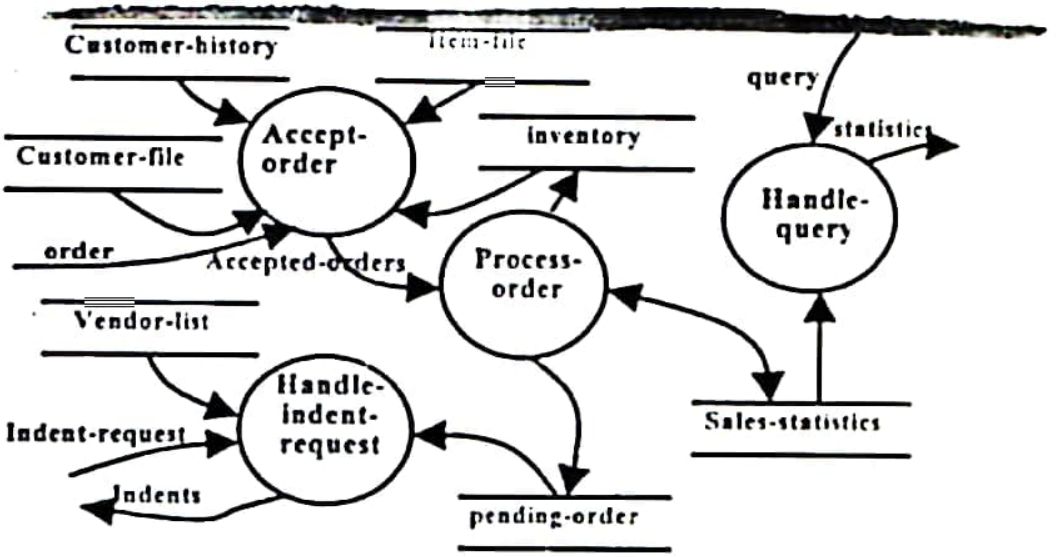
step1: Divide the DFD into three parts.
→ i/p, logical processing, output.

- The processes that transform i/p data from physical to logical form constitute the i/p portion & called as Afferent branch
- The processes that transform o/p data from logical to physical form constitute the o/p portion & called as efferent branch.
- The remaining portion is called central transform

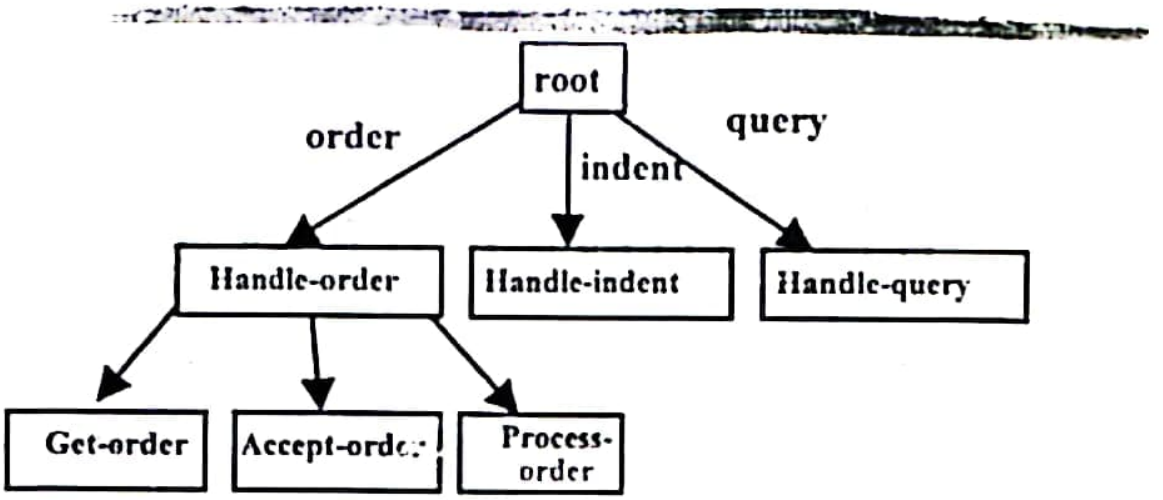
step2: Identify the highest level i/p and o/p transforms. The 1st level of structure chart is representing each i/p and output units as boxes & each central transform as a single box.

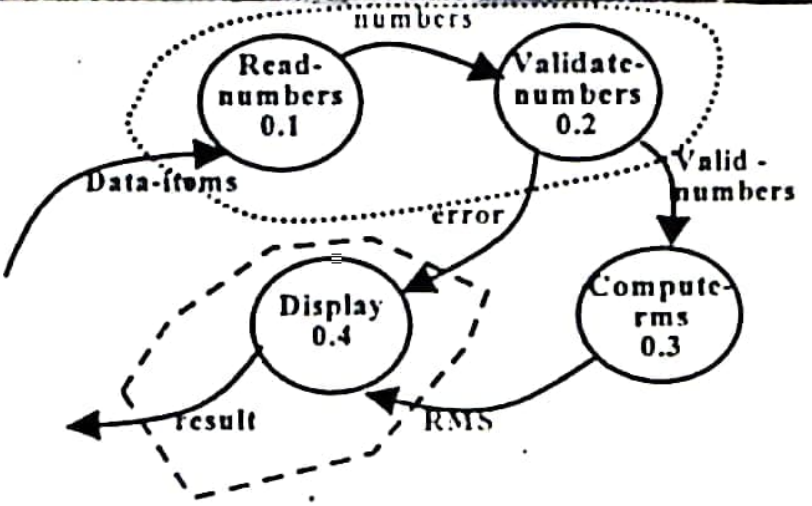
step3: Subfunctions are added. The process of breaking functional components into sub-components is called Factoring. This continues until all the bubbles are represented in the structure chart.

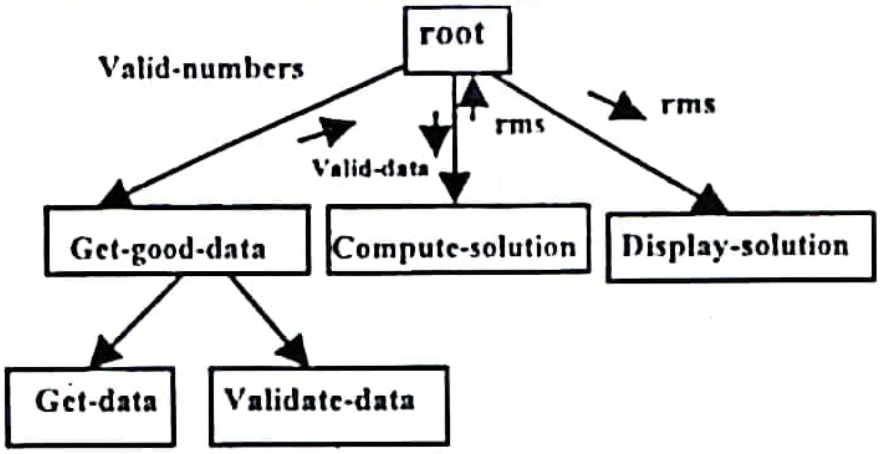
Level 1 DFD for TAS



Structure Chart







• Transaction Analysis

Here these are diff i/p data items, & each diff way in wh/ i/p data is handled is a transactn. The no of bubbles on wh/ the i/p data to the DFD are incident- defines the number of transactions.

DETAILED DESIGN

pseudo code descriptn for processing & different data structures are designed. Represented in the form of MSPECS. MSPECS are :-

- written using structured English.
- For non-leaf mod's describe the different condtns under wh/ responsibilities are given.
- For leaf modules algorithms are used.
- The DFD & SRS and referred.

DESIGN REVIEW

Team consists of members from design, implementation, testing and maintenance. They chk

→ Traceability : chk whtr each bubble of the DFD can be traced to some mod in the chart & vice-versa.

→ Correctness : whether algorithm and the data structures are correct.

→ Maintainability : in future.

→ Implementation : whether the design can be easily and effeciently implemented.

10. CODING & TESTING

CODING

- The i/p to the coding phase is the design document. Here the mod identified in the design document are coded according to the module specificat^{n}s.
- Objective of coding phase
  - Transform design into code.
  - Unit test the code.

Coding standards

* well-defined & standard style of coding adopted by prod s/w devpers is called coding standard. Some adv are:-

→ It gives a uniform appearance to the codes written by different engineers
→ Enhances code understanding.
→ encourages good programming practices.

Standards & Guidelines

There are certain general standards & guidelines wh/ are adopted but good s/w dev org^{n}s develop their own style depending upon their needs.

REPRESENTATIVE CODING STANDARDS

- Rules for limiting the use of globals.
- contents of the headers for different modules.

- Naming conventions for global variables, local and constant identifiers.
- Error return conventions and exception handling mechanisms.

REPRESENTATIVE CODING GUIDELINES

- Do not use too clever and difficult to understand coding style
- Avoid obscure side-effects.
- Do not use an identifies for multiple purposes, because they make understandability and future enhancements difficult.
- code shld be well documented, there shld be one comment line on the avg for every three source lines.
- The length of any function shld not exceed 10 source lines.
- Do not use GOTO statements.

CODE REVIEW

code review of a module is carried out after the mod is successfully compiled and all the syntax errors eliminated. There are two types:

→ code walkthrough.

→ code inspection.

• Code Walk-Throughs

- Is an informal code analysis technique, undertaken after coding of a module is complete.
- A few members of the development team select some test cases and simulate execution of the code by hand-using, to discover the algorithmic and logical errors.

Some Guidelines:- (not Rules)

- The team performing code walkthrough should not be either too big or small (3-7)
- Discussion should focus on discovery of errors and not how to fix.
- To foster co-operation, managers should not attain the walkthrough meeting.

• Code Inspection

aims at discovery of commonly made errors caused due to oversight and improper programming, whether proper coding standards were followed or not..

Common errors chkd during inspections -

- Use of uninitialized variables.
- Jumps into loops, non-terminating loopes.
- Improper storage allocation & deallocation.
- Use of incorrect logical operators or incorrect precedence among operators.

- Clean-room Testing (IBM)

- walkthroughs, inspect^n & formal verification.
- Programmers are not allowed to test their code.
- Produces reliable and maintainable code and documentation.
- Time-consuming processes.

- Software Documentation

Purpose of documents :-

- enhance understandability and maintainability
- help users effectively use the system.
- help in overcoming the manpower turnover probs.
- helps managers in tracking the progress of the project.

Types of Documents :-

Internal Documentat^n — App module headers and comments in the source code, meaningful names, module and function headers, use of user defined data types, appropriate code commenting and meaningful variable names.

External Documentation — user manual, SRS, design document, test plans.

There shld be consistency in documentation and the documents should be up-to-date, & shld be produced in an orderly manner

TESTING

➡ Aim of testing process is to identify defects existing in a software product.

- **What is Testing?**

is a set of activities wh/ subject the program to a set of test i/ps and observe if the program behaves as expected.

- **Terms associated with testing.**

Failure: manifestation of an error.

Test case: is a triplet [I, S, O], where I is the data i/p to the system. S is the state of the system at wh/ data is i/p & 'O' is the expected o/p the system.

Test suite: is a set of all test cases with which a given software product is to be tested.

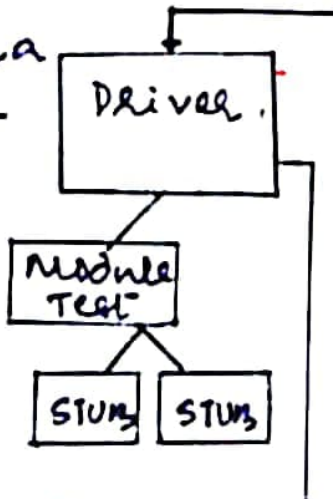
- **Verification Vs Validation**

Verification	Validation
➡ Refers to the set of activities that ensure the s/w correctly implements a specific function.	➡ Set of activities that ensure that the s/w that has been built is traceable to customer requirements.
➡ Are we building the product right.	➡ Are we building the right product.
➡ Concerned with phase containment of errors	➡ Final product shld be error-free

• **Design of Test Cases:**
is essential to prepare optimal no of test-suites. For this two main systematic approaches are applied - white box and black box testing.

■ **Testing in the large / small - Strategies.**
A S/W product goes through three levels of testing
- Unit testing (small)
- Integration testing ↕
- System testing (large)

⇨ **UNIT TESTING**
undertaken when a module has been coded and successfully reviewed.
Since modules linked with a particular module are not-ready by the time it is tested, an environment consisting of stubs & drivers are used.



- Drivers is a program that accepts test case data, passes on to the module and prints results.
- Stubs replace modules that are sub-ordinate to the module to be tested. For this it may use a simple table look-up mechanism.

INTEGRATION TESTING

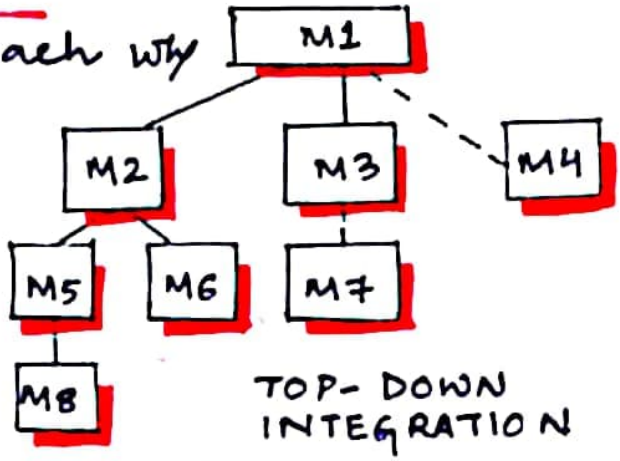
is a systematic technique for constructing the S/W architecture, while at the same time conducting tests to uncover errors associated with interfacing, passing parameters, invoking modules. This is done in a planned manner using an integration plan. Following app can be used to dev the test plan:

- Big Bang
- Top-down
- Bottom-up
- Mixed

↳ **Big-Bang approach:** Here all the modules making up a system are integrated in a single step and tested. But the errors found here are difficult to localize, hence expensive.

↳ **Top-Down Integration:**

is an incremental approach wh modules are integrated moving downward through the ctrl hierarchy. Incorporation can be done using a depth-first or breadth-first manner.



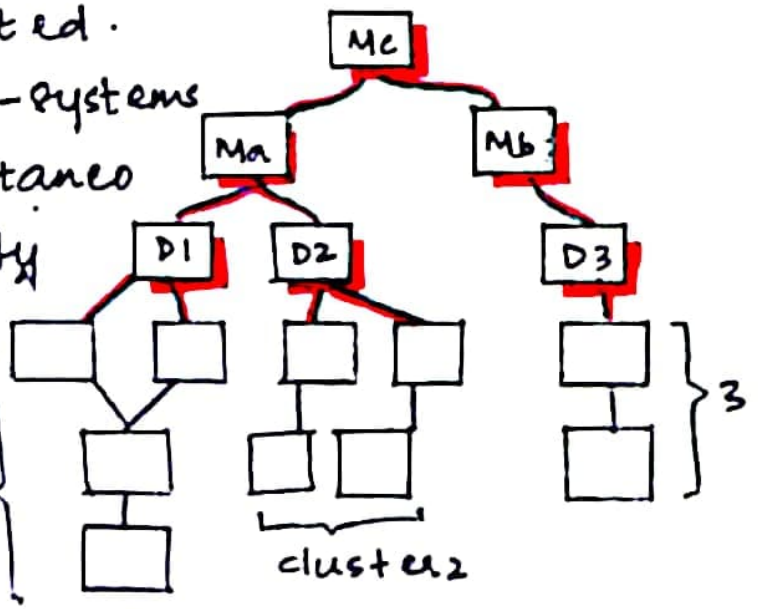
TOP-DOWN INTEGRATION

A major disadvantage in the top-down approach is in the absence of lower level routines, it is difficult to test the high-level routines.

Bottom-up Integration

Here each sub-system comprising of several modules is tested separately and then the full system is tested. Here the need for stubs is eliminated.

Several disjoint sub-systems can be tested simultaneously. But complexity arises when there are large number of small sub-systems.



Mixed Integration Testing

Here a combination of top-down and bottom-up testing approaches is used.

PHASED vs INCREMENTAL INT TESTING

→ A group of related modules are added each time to the partial system under test.	→ Only one new module is added each time
→ less no of steps	→ More no of steps.
→ Difficult to detect or localize failures	→ Easy to locate failures & debug.

SYSTEM TESTING

System tests are designed to validate a fully dev system to assure that it meets its reqs when incorporated with h/w, people & info. There are three main kinds of sys testing:-

- Alpha testing: carried out by the test team within the developing organization.
- Beta testing: performed by a selected group of friendly customers.
- Acceptance testing: performed by the customer to decide whether to accept/reject the delivery of sys.

System tests can be classified as functional and performance tests (black-box): Here we will consider the performance tests.

Performance Tests:

is carried to check the run-time performance of a system as specified in the SRS document.

- Stress Testing: (Endurance) executes a system in a manner that demands resources in abnormal quantity, freq or volume. I/p data vol., data rate, processing are tested beyond the designed capacity. Stress testing is important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours.
- Sensitivity testing.

- **volume Testing**: to chk whether the data structures have been designed successfully for extra-ordinary situations.
- **Configuratn testing**: Is used to analyze sys behaviour in various h/w & s/w configurations.
- **Compatibility testing**: chks whether the interface functions perform as required.
- **Regression Testing**: running an old test-suite after each change to the system or after each bug fix to ensure that no new bug has been introduced as a result of change or bug fix.
- **Recovery testing**: tests the response of the system to the presence of faults, loss of power, devices, services, data etc.

- **maintenance testing**: verify that the artifacts exist and they perform properly.
- **Documentation testing**: ensures that the req manuals (user, maintenance, tech) exist & or consistent.
- **Usability testing**: user interface reqs r tested.
- **ERROR SEEDING**

seeding the code with some known errors.
Let N → Total no of defects n → def found by testing
S → " " of seeded defects s → " " " ".

Then, $n/N = s/S \Rightarrow N = S \times \frac{n}{s} \Rightarrow N - n = \frac{n(S-s)}{s}$

N − n ⇒ Defects still remaining after testing can be found out.

⇒ The effectiveness of testing strategy.

BLACK-BOX TESTING

Here test cases are designed from an examination of the i/p/o/p values only and no knowledge of design or code is required. Common approaches to test case design: -

- Equivalence class partitioning.

Divides the set of i/p values into such classes, where the behaviour of the program is similar to every i/p belonging to the same equivalence class. Some guidelines:

* If the i/p data val's to a sys can be specified by a range of values, then one valid and two invalid equivalence classes shld be defined.

* If the i/p data assumes values from a set of discrete members of some domain, then one ec for valid and another for invalid shld be made.

- Boundary value Analysis

Greater no of errors occur at the boundary of the i/p domain. So BVA selects test cases at the boundaries of different equivalence classes.

Guidelines: -

* If an i/p condition specifies a range bounded by values A and B, test cases shld be designed with values A & B, just above and below A & B.

Other methods: -

(1) Graph-Based Testing Methods.

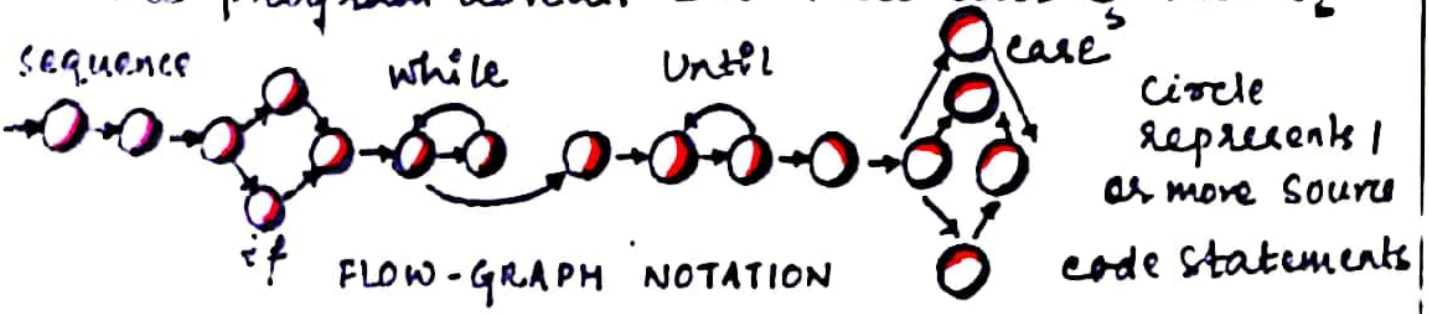
(2) Orthogonal Array Testing. . .

WHITE-BOX TESTING

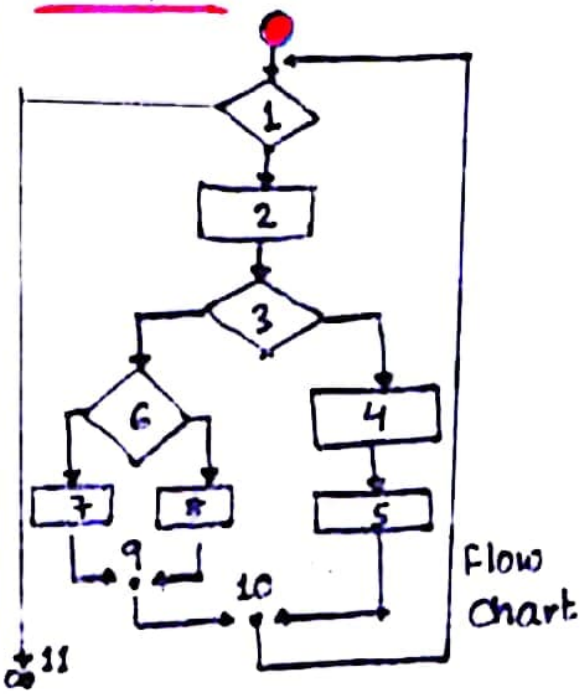
also called structural testing / glass-box testing uses the ctrl. structure as internal structure to design test cases.

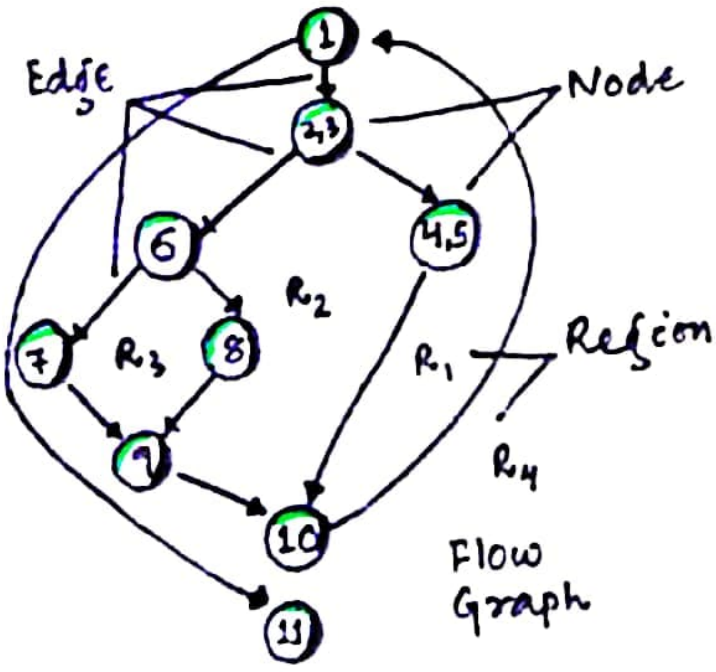
- **Basic-Path Testing**

derives a logical complexity measure of a procedural design. Test cases derived here, execute every statement in the program atleast one time during testing.

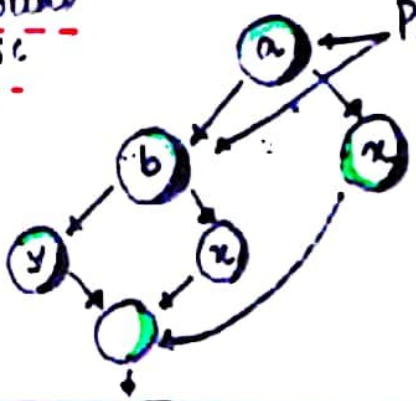


Example:





Compound logic



IF a OR b
then procedure x
else procedure Y
ENDIF

• Independent Program Paths

An independent path is any path through the program that introduces at least one new set of processing statements or a new condtn. It must move along at least one edge that has not been traversed before the path is defined.

Ex:- path 1: 1-11 , 2: 1-2-3-4-5-10-1-11

" 3: 1-2-3-6-8-9-10-11, 4: 1-2-3-6-7-9-10-11.

Paths 1,2,3,4 → Basis set. (But it is not unique)

■ Cyclomatic Complexity.

is a s/w metric that provides a quantitative measure of the logical complexity of a program or/ it provides the no of independent paths in the basis set & provides a upper bound to the no of tests that must be conducted.

Cyclomatic complexity can be calculated in one of the three ways:

1. The no of regions in the flow graph.
2. CC = E - N + 2

   E → number of flow graph edges

   N → N of flow graph edges.

3. CC = P + 1

   P → number of predicate nodes.

➡ Deriving Test-Cases

→ Using the design/code, draw a flow graph.
→ Determine the cyclomatic complexity.
→ Det a basis set of linearly independent paths.
→ Prepare test cases that'll force execution of each path in the basis set.

CONTROL-STRUCTURE TESTING

Condition Testing

exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression.

$E_1 <$ Relat^n Operator $> E_2$, $E_1, E_2 \rightarrow$ arithmetic exp's.

$<, \leq, =, \neq, >, \geq$.

A compound condition contains two or more simple conditions, Boolean operators and parenthesis.

Here error includes, incorrect, missing, extra Boolean operators, Boolean variable errors, parenthesis errors, relat^n operator and arithmatic exp errors.

Data-Flow Testing

selects test-paths according to the location of defn's & uses of variables in the program. Assume:-

- Each statement is assigned a unique statement no.
- Each funct^n does not modify its parameters.

For a statement, with S NO → S

DEF(S) = {X | statement S contains a defn of X}

USE(S) = {X | statement " " a use of X}

* Defn use Chain.

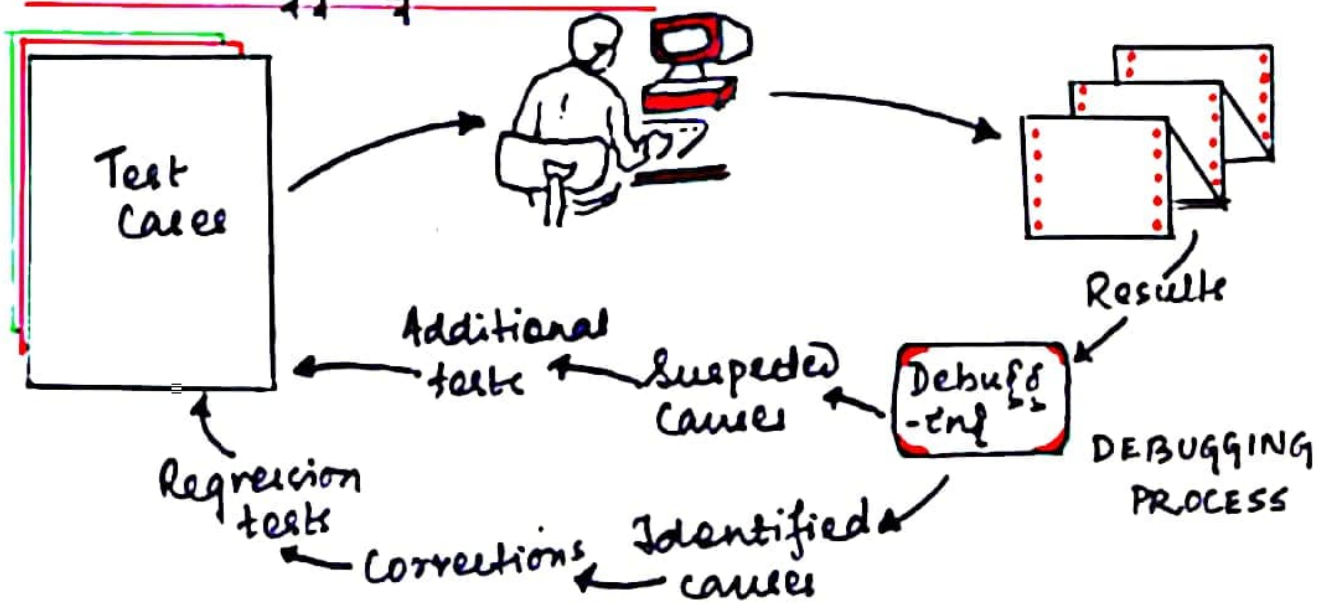
Mutation Testing

Here the program under test is changed to create a faulty version called mutant. Then the mutant program is run through a set of test cases which produce new failures. If no new failures appear, then the test does not exercise the path containing the mutated code, which means the program is not fully tested.

THE ART OF DEBUGGING

Debugging as a consequence of successful testing removes the error uncovered by the test-cases.

- The Debugging Process



- Characteristics of bugs

- Symptom and the cause may be geographically remote.
- Symptom may disappear temporarily.
- There may not be any error at all.
- caused by human error, timing problems.
- Real-time problems.
- may be due to variety of reasons.

- Debugging approaches / Strategies

➡ Brute-force

Here the prints of the intermediate values are taken, memory dumps, run-time traces are considered in a hope to find out the error. Its a costly & least efficient method.

➡ Backtracking

Beginning at the site where a symptom is uncovered, the source code is traced backward until the site of cause is found.

➡ Cause elimination
This is done by binary partitioning, where data related to the error are organised to isolate potential causes, using hypothesis.

➡ Automated Debugging
Here debugging approaches are automated with tools like debugging compilers, tracers, test case generators and cross-reference mapping tools.

• Correcting the error
Questions to be asked b'fore debugging.
? is the cause of the bug reproduced in another part of the program.
? what next bug might be introduced by the fix.
? what cud we have done to prevent this bug.

* After every thing, regression testing is done.

■ PROGRAM ANALYSIS TOOLS
I/p → source code, executable code.
O/p → report- regarding prog size, complexity etc.

• Static Analysis Tool
Assess characteristics without executing it, code walk-throughs, inspections, tools, compilers. Produces a Kiviat chart that shows analyzed values for cc, no of source lines, percent of comment-lines.

• Dynamic Analysis Tools
requires actual program execution to describe the structural coverage of the test suite. The results are reported in form of a histogram or pie-chart
ex:- data on extent of statement, branch and path coverage achieved.

SOFTWARE QUALITY MANAGEMENT

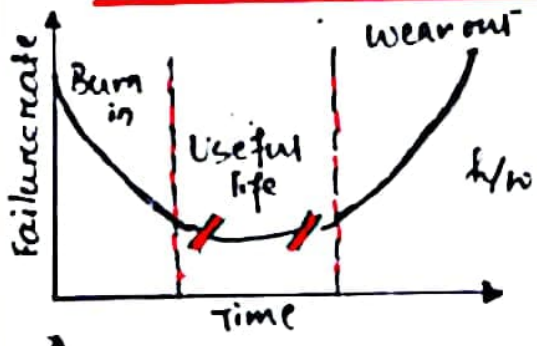
SQA: Software Quality Assurance (SQA) is one, major aim is to help an organization develop high quality s/w products in a repeatable manner.

- **SOFTWARE RELIABILITY:** probability of failure-free operat^n of a computer program in a specified environment for a specified period of time.

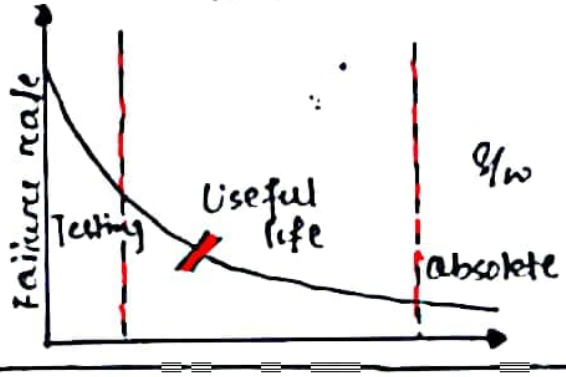
Ex: Prog X has a Reliability of 0.96 over eight elapsed processing hours. i,e If 'x' were to be executed 100 times and require a total of 8 hours of execut^n time, then it is likely to operate correctly 96 times.

- The quantity by wh/ the overall reliability of a program improves, due to the correct^n of a single error, depends on how frequently the corresponding instruct^n is executed.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

- **HARDWARE Vs SOFTWARE RELIABILITY**



Most h/w failures are due to component wear and tear
eg: logic gate may get stuck at 1 or 0, or a resistor might get short-circuited.



Most s/w failures are attributed to design techniques, inefficient use of tools and standards.

■ Realiability Metrics

are required to quantify the reliability of a s/w product, and specify in the SRS.

Some reliability metrics that are widely used:

1. ROCOF (Rate of occ of Failure): Freq of occurrence of failures over a specified period of time.
2. MTTF (Mean Time to Failure): Average time (run-time) between two successive failures, observed over a large number of failures.

$$MTTF = \sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}$$

$n \rightarrow$ no of failures
$t_1, t_2 \ldots t_n \rightarrow$ time of occurrences of failure.

3. MTTR (Mean time to repair): Average time taken to track the errors and fix them.
4. MTBF (Mean time bet failures): MTBF = MTTF + MTTR
5. POFOD (Probability of failure on demand): likelyhood of a system failing when a service request is made.

   POFOD of 0.001 means?
6. Availability: Probability that a program is operating as per requirements at a given point of time.

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

This metric is an indirect measure of the maintain-ability of a s/w.*

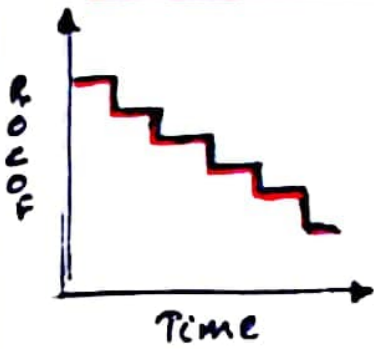
But reliability can be estimated more accurately considering the consequences of failure. A possible classificath is:

- Transient: For certain i/p values.
- Permanent: For all i/p values
- Recoverable errors
- Unrecoverable
- Cosmetic: minor irrat"s, do not lead to incorrect result.

■ Reliability Growth Modelling
is a mathematical model which determines -
- how s/w reliability improves as errors are fixed.
- predict when a particular level of reliability is attained.
- when to stop testing.

Some simple models are:-

➡ Jelinski and Moranda model



Here it is assumed that reliability increases each time an error is detected and repaired. But it is unrealistic to assume that all errors contribute equally to reliability growth.

➡ Littlewood and Verrall's model

Here negative reliability growth is reflected, i,e when a repair is carried out, it may introduce additional errors. Also as errors r repaired, the avg improvement in reliability per repair decreases (diminishing Returns)

■ STATISTICAL TESTING

obj is to determine reliability than discover errors.

steps:

(1) Determine the operational profile: i,e divide the i/p data frm diff users into a no. of i/p classes & then assign prob [P_i] to each class [C_i], wh/ signifies the prob of an i/p value from that class.

(2) Generate a set of test data corresponding to profile.

(3) Apply the test case to the s/w and Record time between each failure.

(4) Then reliability is computed.

* It requires some significant test cases & seeded errors.
* This concentrates on most used parts of the system, but the method is difficult to implement.

Software Quality

in conformance to explicitly stated functional and performance characteristics, explicitly documented development standards, & implicit characteristics that are expected of all professionally developed software.

There are several quality factors :-

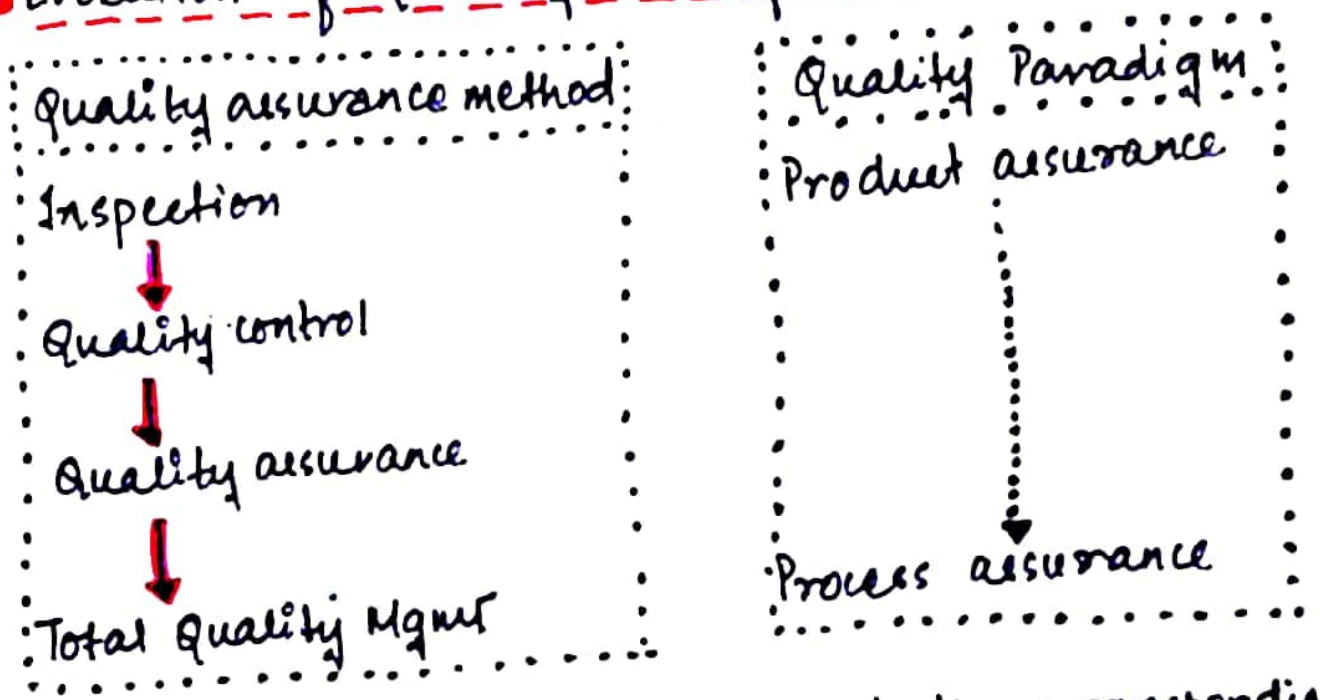
- Portability
- Usability
- Reusability
- Correctness
- Maintainability
- Reliability
- Efficiency
- Integrity → eq: $I = \Sigma[1-(T \times (1-S))]$
  T - Threat
  S - Security
- Flexibility
- Testability
- Interoperability

Software Quality Management System

ensures that the products org's develop have the desired quality. A quality system consists of-

- Managerial structure & individual responsibilities.
- Quality system activities.

Evolution of Quality Management Systems..



Evolution of Quality System and the corresponding shift in the quality paradigm.

ISO 9000

(International Standards Organizat^n) ISO is a consortium of 63 countries established to formulate and foster standardization.

What is ISO 9000 Certification?

serves as a reference for contract between independent parties. It specifies a set of guidelines for repeatable and high quality prod development.

ISO 9000 is a series of three standards:

- ISO 9001: S/w org's, design, development, product and servicing of goods.
- ISO 9002: steel and car manufacturing. not app to software products.
- ISO 9003: orgn's involved in installation & testing of prod.

ISO 9000 for Software industry

ISO 9000 is a generic standard applicable to a large gamut of industries, hence its difficult to interpret them in the context of software development organizat^{n}s.

The reason being:

- Software is intangible.
- Only raw material is data.

∴ A separate document called ISO 9000 part-3 in 1991 was released to interpret for software industry.

Why get ISO 9000 Certification?

- confidence of customer's increase especially in international market.
- well-documented process contributes to repeatable and higher quality s/w.
- Dev process becomes more focused, efficient & cost-effective
- Recommends remedial action.
- Basic framework for optimal process & TQM

➡ How to get ISO 9000 Certification?

A company to get certification needs to apply to a ISO 9000 registrar for registratn. The process :-

- Applicatn to the registrar.
- Pre-assessment of the orgaizatn.
- Documents are reviewed and suggestn's made.
- Registrar chks whether the suggestions made earlier have been complied with.
- Registrar awards ISO 9000 certificate after successful completion of all the phases.
- Continued surveillance is done periodically.

* certificate can be used for process not for product.

➡ Summary of ISO 9001 requirements.

- management Responsibility
  - effective quality policy.
  - Responsibility of persons associated with quality, should be documented.
  - 1 mgmt person, independent of dev process, must be responsible for the quality system.
  - Periodical Review.
- Quality system
  - must be maintained and documented.
- Contract Reviews
- Design control
  - i/p must be adequate, o/p of required quality.
  - design verification, change control.
- Document control
  - configuration management tools.
- Purchasing (Quality).

- Purchased supplied prod should be properly checked.
- Product identificatn [conf mgmt]
- Process control
- Inspectn and testing.
- Equipments must be properly calibrated.
- Test status
- Keeping faulty software out of released product.
- Corrective action
- Handling • Quality records • Audits • Training

➡ Salient features of ISO 9001 req's.

- All documents shld be properly managed, authorized, and controlled
- Proper planning and progress review.
- Documents shld be checked and reviewed.
- Product testing against specificatn.
- Addressing organisational aspects.

➡ Shortcomings of ISO 9000 certification

- Does not give any guidelines for defining an appropriate process.
- Certification process is not foolproof and no internal accredition agency exist. So variatns in the norms of awarding may exist.
- Companies get over-confident over the process and neglect the requirement that special expertise and experience may be required in certain domains. Creativity and individual skills are neglected.
- ISO9000 does not automatically lead to continous process improvement, or lead to TQM.

SEI CAPABILITY MATURITY MODEL

proposed by SEI, Carnegie Mellon University, USA.
CMM is a reference model for inducting the s/w process maturity into different levels It is used in two ways:- (1) capability evaluatn
(2) Software Process Assessment.

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

CMM level	Focus	Key Process Areas (KPA's)
• Initial	competent people	
• Repeatable	Project Management	- S/w Project planning - S/w Config mgmt
• Defined	Defn of processes	- Process defn - Training program - Peer reviews.
• Managed	Prod & process Quality	- Quant process metrics - S/w Quality mgmt.
• Optimi-zing	Continous process improvement	- Defect prevention. - Process change mgmt. - Tech change mgmt.

- Each stage is designed for gradual improvement in the maturity of the org.
- Most Frequent Complain, need more guidance on "How to improve the processes"?

➡

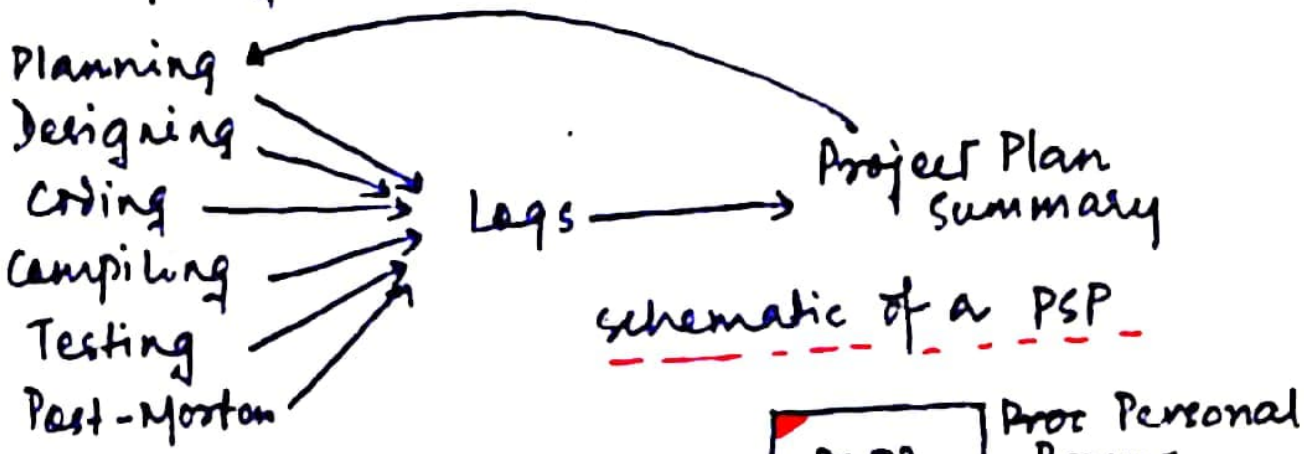
SEI CMM	ISO 9000
(1) For S/w industry	(1) For all
(2) Beyond TQM, Qual Assurance	(2) SEI - level 3
(3) For internal use	(3) Can be quoted in official documents.
(4) Provides Key process areas for improvement	(4) No such thing is provided.

PERSONAL SOFTWARE PROCESS

- Process for individual use. Helps engineers to measure and improve the way they work. Some features are:-

- Time Measurement: Engg shld measure the time he spends for designing, writing code, testing etc using a step-clock. The clk maybe stopped during breaks, while attending telephone calls.

- PSP Planning: Individuals must estimate the max, min & avg LOC for a prod, and dev time.



schematic of a PSP

Proc Personal Process Evolution

checklists are prepared to review design & code.

Personal Qual mgmt Des

Time and schedule planning

Personal mgmt, size measures coding standards

* SEI-CMM - not advisable for small projects
* Six Sigma.

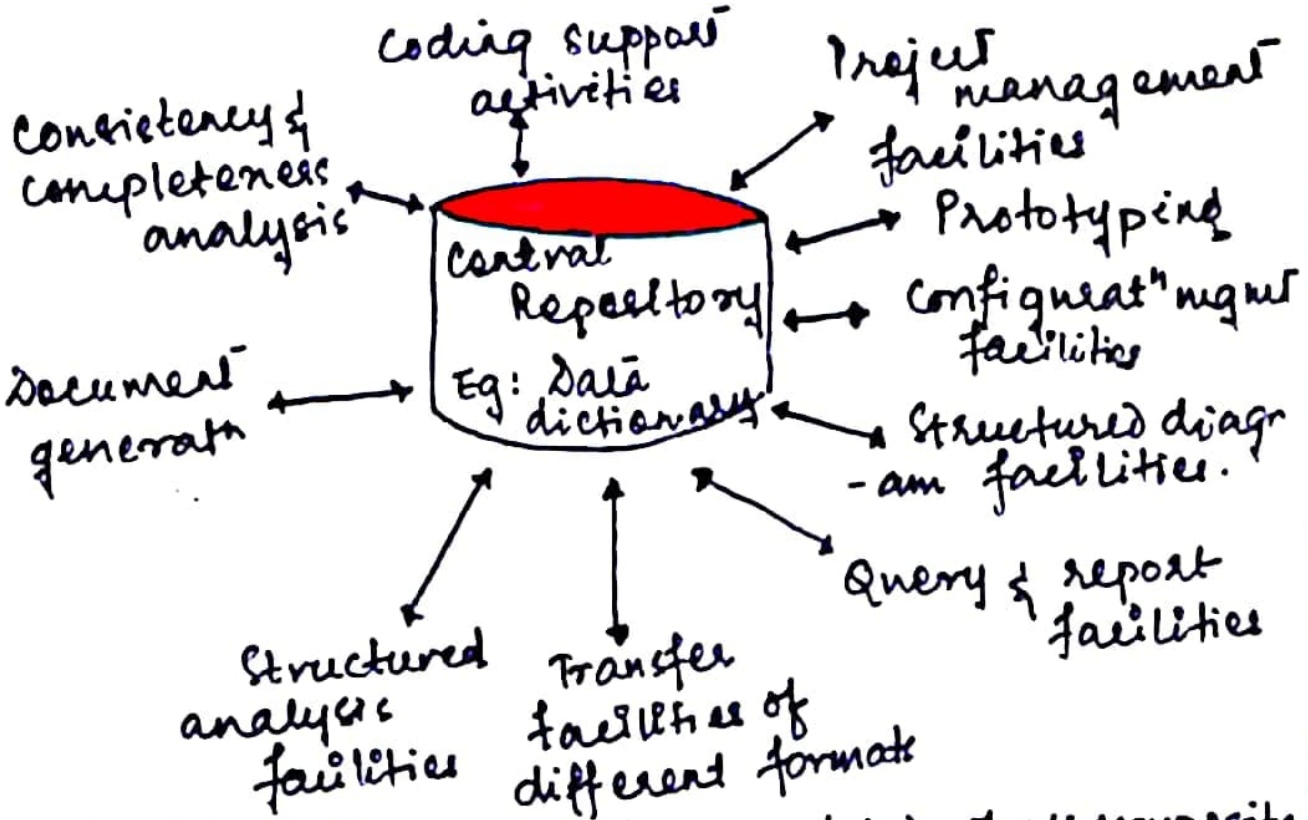
COMPUTER AIDED SOFTWARE ENGINEERING

CASE TOOLS : any tool used to automate some activity or phase related task of software development.

Objective : - To increased productivity
- To produce better quality s/w at lower cost.

CASE ENVIRONMENT

is required for integrating different CASE Tools. otherwise the data generated by one tool would have to input to the other tools. This may include format conversions (different vendors use diff format). case tools are characterized by stages of SDLC on wh/ they focus. But these tools share common Informat" hence it is required that they integrate through some central repository.



The central repository consists the defn's of all composite and elementary data items. Thus a CASE environment facilitates the automation of step by step methodologies for software development.

• Programming Environment - integrated collection of tools to support only the coding phase. The tools integrated are text editor, a compiler and a debugger.
Ex:- Turbo C, Visual Basic etc.

Benefits of CASE

- Effort is reduced by 30-40%.
- Improvements in Quality.
- Produce high quality and consistent documents.
- Reduce hard/dull work in a sw engineers work.
- Cost savings.
- Structured and orderly approach.

CASE SUPPORT IN SOFTWARE LIFE CYCLE

Prototyping support

Here CASE TOOL requirements are:-
Define user interaction, system ctrl flow, store and retrieve data, Incorporate some processing logic.
A good prototyping tool should support:
- should support the user to create a GUI using a graphics editor.
- should integrate with the data dictionary.
- " " external user-defined modules.
- user shld be allowed to control the running of the prototype
- i/p and o/p data management, mock-up scrn.

Structured analysis & design.

- Support effortless making of analysis and design programs.
- Easy navigation through different levels.
- support completeness and consistency checking.
- The system should disallow any inconsistent operation, but it is difficult to implement this.

➡ CODE GENERATION
- should support - generatn of module skeletons or templates.
- Brief description of module, author name and the date of creatn in some selectable format.
- shld generate records, structures & class defn's.
- generate database tables.
- shld generate for UI, window-based applicatns.

➡ TEST CASE GENERATOR
- support both design & requirement testing.
- Generate test set reports in ASCII format wh/ can be directly imported into the test plan.

■ OTHER CHARACTERISTICS
➡ Hardware and Environmental Requirements
- The CASE Tool should fit into existing H/w capabilities work satisfactorily for max possible number of users, working simultaneously
- shld supp multi-windowing environment for the users to see >1 diagram at a time.
- Also facilitates navigatn & switching from 1 part to another

➡ Documentation Support
The deliverable documents shld be organized graphically and should be able to incorporate text & diagrams.

➡ Project Management
Support collecting, storing and analyzing info on the H/w's project progress such as estimated task durat'n, scheduled and actual start date, completion etc.

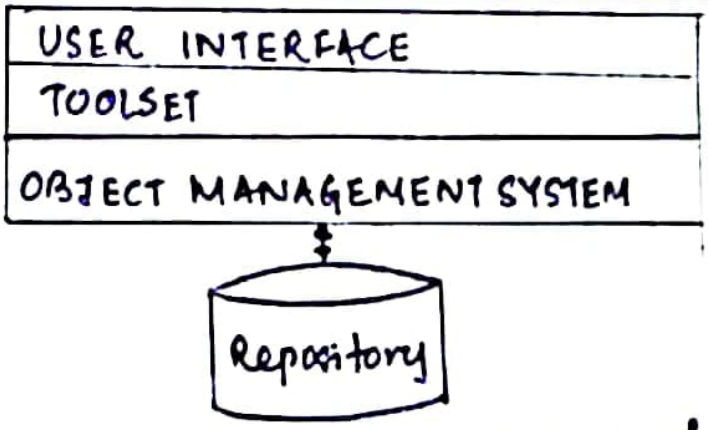
➡ External Interface
Exchange of info for reusability of design. The data dictionary should provide a programming interface to access information..

➡ Reverse Engineering support
Generatn of structure charts, data dictionaries from the existing source codes.
➡ Data Dictionary Interface
update and view access to the data items & their relatns. Shld have generatn of print and analysis reports.
➡ Tutorial and help
regarding, the techniques and facilities of the case tool

2nd GENERATION CASE TOOL

→ Case admin to tailor the tool to a particular methodology
→ Intelligent diagramming support.
→ Integratn with implementation Environment.
→ Data dictionary standards
→ Customization support.

ARCHITECTURE OF A CASE ENVIRONMENT.



- User Interface: provides a consistent framework for accessing different tools.
- object management system & repository:
Different CASE Tools represent the software product as a set of entities such as specification, design, test data, project plans etc. The OMS maps these logical entities to the repository.

— x —

SOFTWARE MAINTENANCE

Software maintenance denotes any changes made to a software product after it has been delivered to customer.

CHARACTERISTICS OF SOFTWARE MAINTENANCE

- Products need to run on newer platforms, newer env's and enhanced features.
- When support environment changes.

• Types of Software Maintenance

→ Corrective : Rectify the bugs observed while the system is in use.

→ Adaptive : when cust need the prod to run on new OS, platforms, or interface with new h/w or s/w.

→ Perfective : change functionality, add new features as per customer demand and enhance performance.

• Characteristics of Software Evolution (Generalized by Lehman and Belady

→ 1st Law : A s/w prod must change continually or become progressively less useful.

→ 2nd law : The structure of a program tends to degrade as more & more maintenance is carried out on it.

→ 3rd Law : Over a proj's lifetime, its rate of development is approximately constant.

• Problems associated with Maintenance

- Not planned, neglected, ad hoc, no system.
- Necessary to understand others work.
- Majority are legacy sys (hard to maintain b'cos of poor documentation, unstructured code (spaghetti code), non-availability of reqd personnel)
- Process is expensive and takes more time to implement.

SOFTWARE REVERSE ENGINEERING

process of recovering the design and the req's specificatn of the product from analysis of its code. purpose: facilitate the maintenance work by impro. -ving the understandability of a system & prod- -uce necessary documents for a LEGACY SYSTEM.

• Legacy s/w : lack proper documentation & are highly unstructured. Even well-designed s/w become legacy s/w as their structure degrades b'coz of series of maintenance efforts.

PROCESS:

Reformat program → Assign meaningful names → Simplify condtns → Remove GOTOs → Simplify processing

Cosmetic changes carried out for improving the code's readability, structure & understandability

Requirement spec's.

Design → Module specificatn → code

PROCESS MODEL FOR REVERSE ENGINEERING.

SOFTWARE MAINTENANCE PROCESS MODELS

Maintenance projects depend upon various factors such as:-

(1) The extent of modificatn to the product required.
(2) The resources available to the maintenance team.
(3) Condtns of existing product (structure, documents).
(4) Expected project risks.

• For minor changes, the code can be directly modified and the change reflected in all documents.

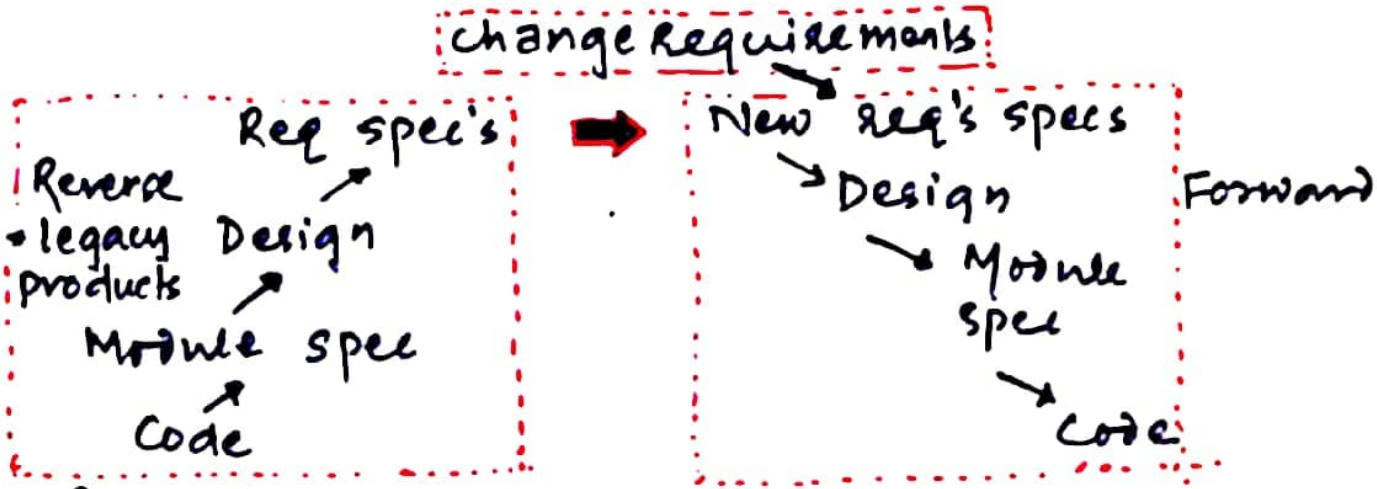
So depending upon the Req's there are 2 models:-

(1) For minor Re-works.

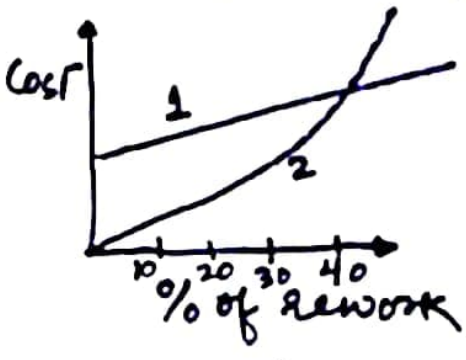
Gather change req's → Analyze change req's → Devise code change strategies → Apply change to old code → Update documents / Integrate n test

(2) Amount of rework is significant

REENGINEERING = REVERSE + FORWARD



* Re-engineering is more costlier.



- Process1 is preferred when the amt of Rework is ≤15%.
- Re-engg preferable for prod's with high failure rate.
- Also preferable for legacy products having poor design and structure.

ESTIMATION OF MAINTENANCE COSTS

By Boehm

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

ACT - Annual change Traffic

Maintenance cost = ACT x development cost

* But these are approx, don't take into account factors such as exp, complexity, familiarity with prod.

SOFTWARE RE-USE

Component based software engineering / reuse is a process that emphasizes the design & and construction of comp-based systems using reusable sw components.

- What can be reused?
  - Requirements specification
  - Design
  - Code
  - Test cases
  - Knowledge (abstract)
- Why almost no re-use so far?

- Difficult to anticipate the exact components that can be reused.
- Difficult to understand & adapt to new apps.

- Basic Issues in any re-use program.

➡ Component Creation: The reusable components must be first identified using some techs.
➡ component indexing and storing: classification of the reusable components so that they can be easily searched & storing them in Relational d/b.
➡ component searching: with some matching tech
➡ Component understanding: The programmers need a sufficient understanding of the component to decide whether to use it or not.
➡ Component adaptation: but this may turn out to be a source of bug.
➡ Repository maintenance: New components as and when created shld be entered into the repository, fault comps be tracked & obsolete one's removed.

■ REUSE APPROACH

➡ Domain Analysis

: To identify the reusable components for a problem.
Reuse domain - A technically related set of app areas. A reuse domain is a shared understand-ing of some community, characterized by concepts, tech's and terminologies that show some coherence.
- Domain analysis identifies the objects, opera-tion's and the relationships among them.
For eg: Airline Reservation System.
- The actual construction of the reusable components for a domain is called domain engineering.

➡ Evolution of a re-use domain

stage 1. There is no clear and consistent set of notat^n. All sw is written from scratch. No reuse.
stage 2. Experience from similar prj's is used in a new development effort. Means only knowledge reuse.
stage 3. Standard sol's to standard problems are availa-ble. There is both knowledge & component reuse.
stage 4. The domain has been fully explored. The dev can be automated. Prog's are written using a domain specific language, also known as applicat^n generator.

➡ Component- Classification

Component's need to be properly classified for indexing and storage. Here is Prieto-Diaz's classificat^n scheme:-
Objects are classified based on:
- Actions they embody
- Objects they manipulate
- Data structures used
- Systems they are part of.

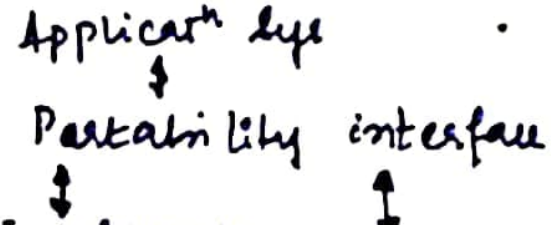
➡ Searching
→ Using a web-interface, search for an item using keywords, and then frm these results do a browsing.
→ The links help to locate additional products and compare their detailed attributes.
→ Several iterations may be repeated.

➡ Repository maintenance
→ enter new items, retiring obsolete items.
→ modifying the search attributes.
→ modifying the search links.

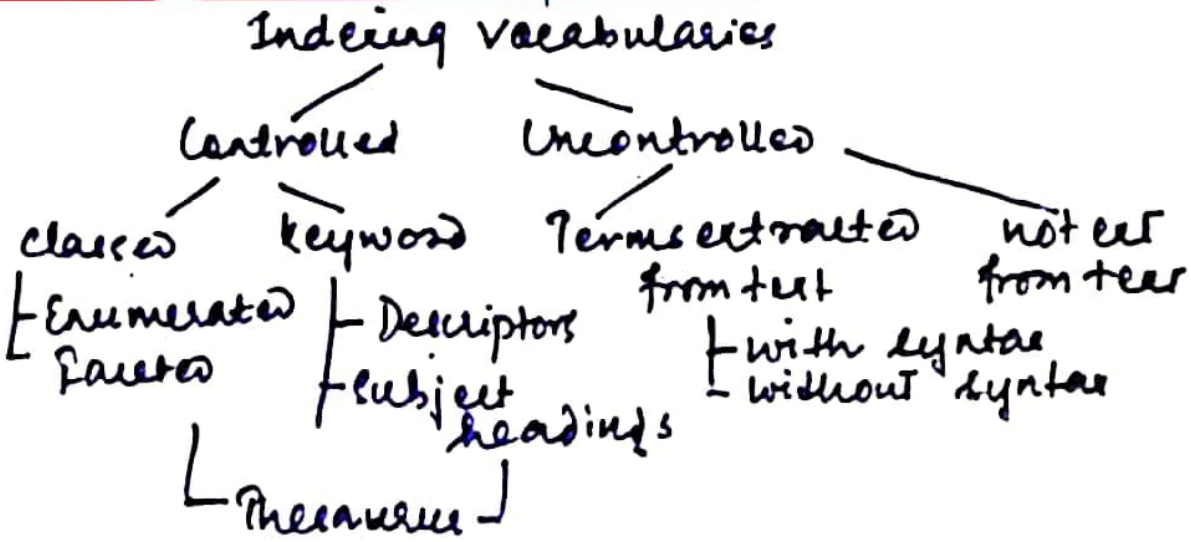
➡ Re-use without modification
Once standard solutions emerge, no modifications are required - one can directly plug in the part.
Applicatn generators translate specificatns into application programs. These programs generate fewer errors, easier to maintain, Reduced development effort etc. But these are less successful with real-time systems.

■ REUCE AT ORGANIZATION LEVEL
◉ Assessing a product's potential for reuse
A questionnaire is used to elicitate response from programmer's working in similar app domains, then depending on the answers the component is either taken for re-use, or it is modified and refined, or ignored.
◉ Refining products for greater re-usability.
→ Name generalizatn
→ Operation generalization
→ Exception Generalization . . .

Handling Portability Problems



A taxonomy of indexing method.



The Reuse Environment-

→ Component db capable of storing clio components & classification information necessary to retrieve.
→ Library mgmt sys to provide access to the db.
→ Component Retrieval system.
→ Tools to supp the integratn of reuse comps into new design or implementation.

other components identified.

→ Qualified components
→ Adapted components
→ Assembled components
→ Updated components.

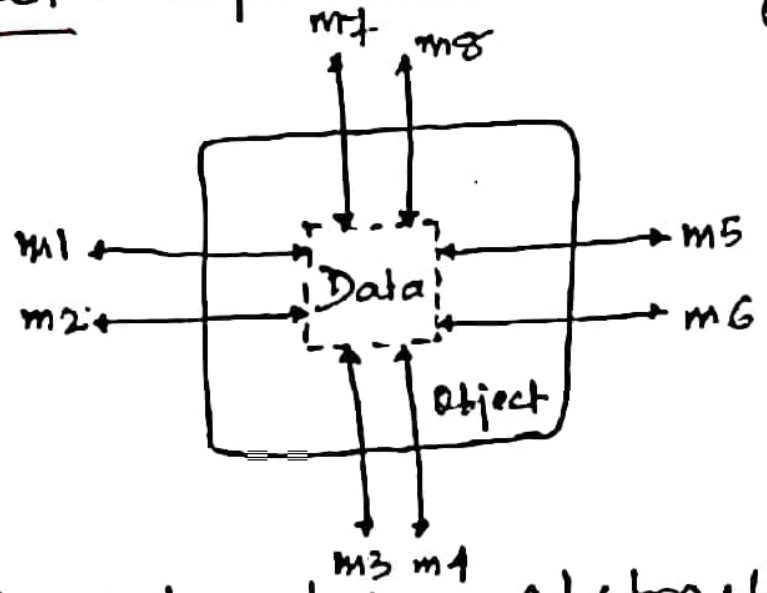
Also read:
Pressman Chapter-30, CBSE

1

15. OBJECT-ORIENTED DESIGN USING-(

➡ Basic mechanisms

(1) Object : represents a tangible real



mi are
the obj

→ Implements data abstraction

eg: a stack might store its internal d
in form of an array or as a linked
objects do not know. they only know t
possible such as push, pop, top-of-stac

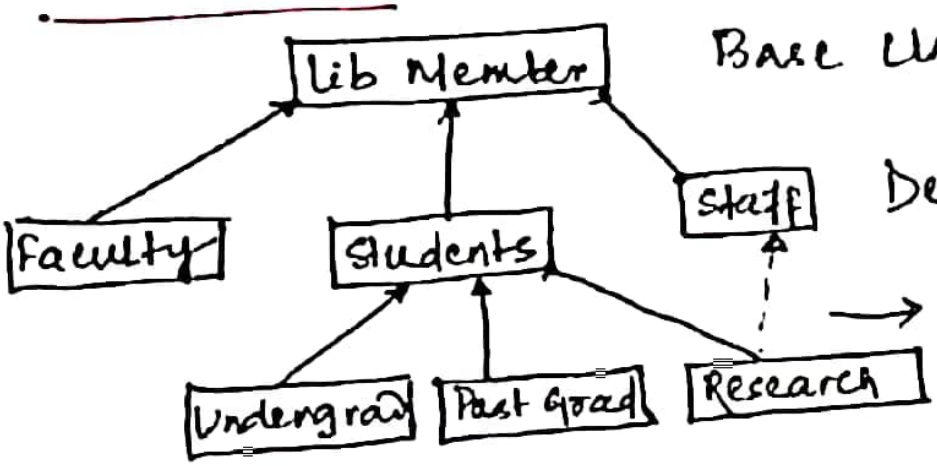
eg: An employee Object

Employee	→ Object name
name: string address: string dob: date empno: int salary: int	→ Attributes

(3) Methods & Messages

Operations supported are methods and are invoked by sending messages to it

(4) Inheritance



Base class is a

Derived class

→ Multiple In

⇨ Key concepts

- Abstraction
- Encapsulation
- Polymorphism — eg: create circle.

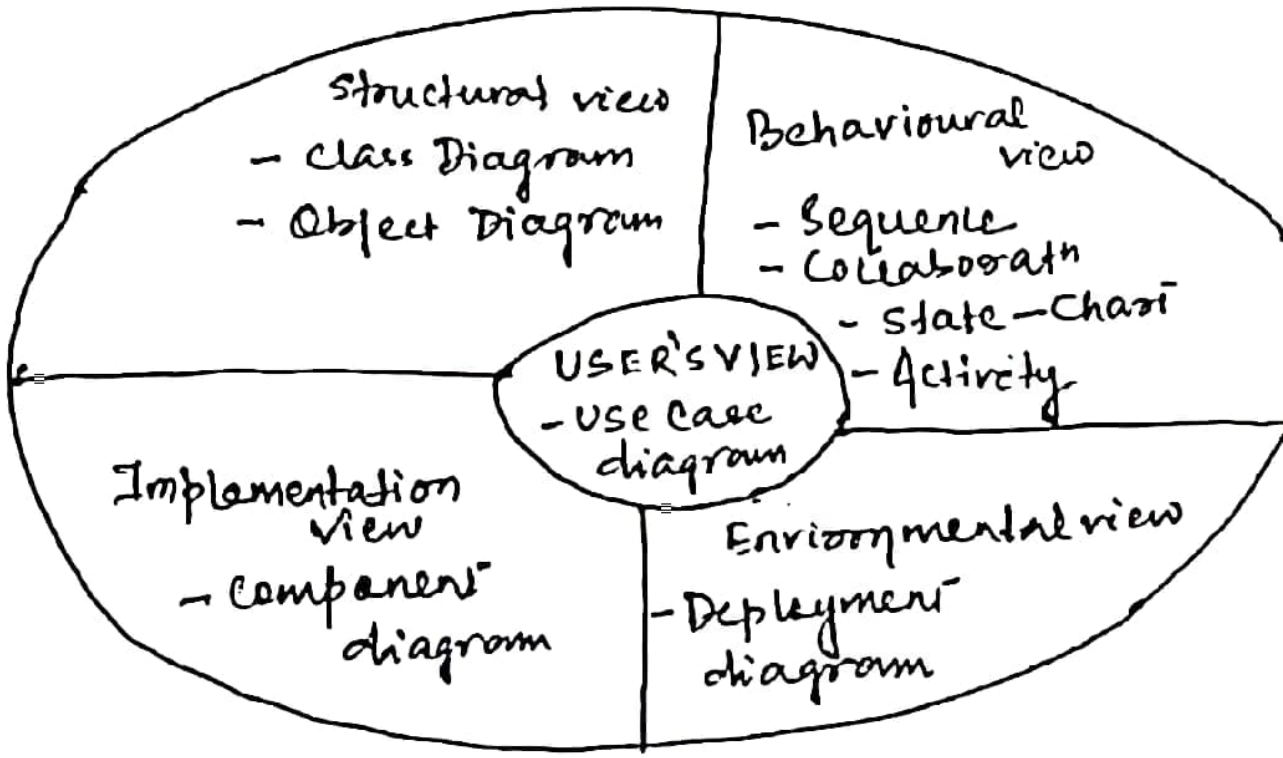
default
centre, radius,
centre, radius, f

depending upon the
provided appropriate m
invoked.

- Genericity (ability to parameterize c

eg: class of type stack, later instantiat

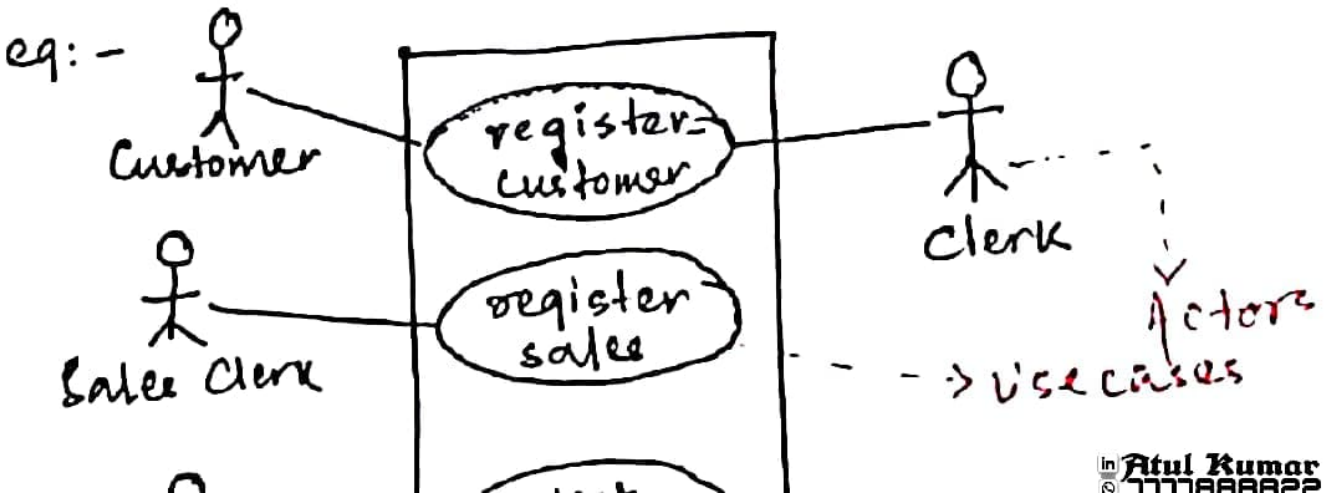
UNIFIED MODELLING LANGUAGE (UML).



⇨ USE-CASE DIAGRAM

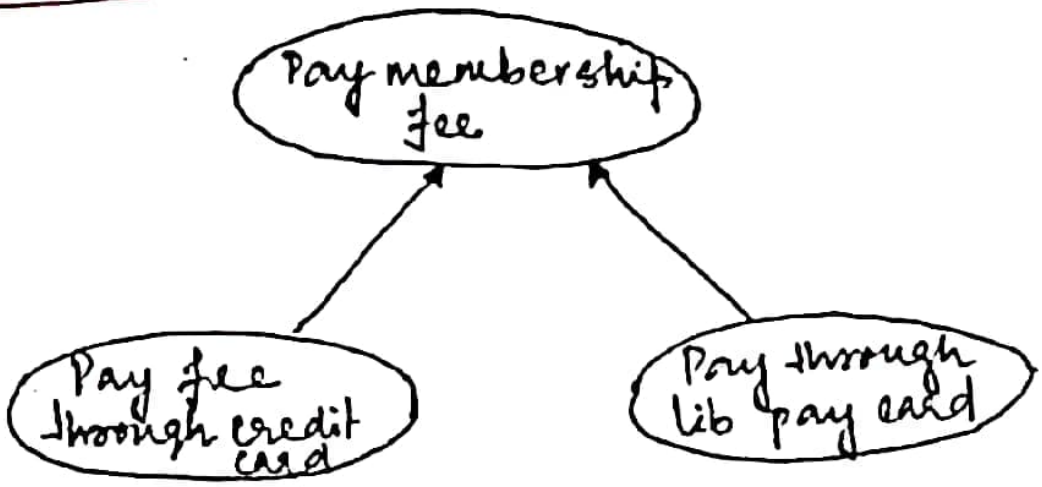
consists of a set of "use cases", why different ways in why a system can b

eg:-



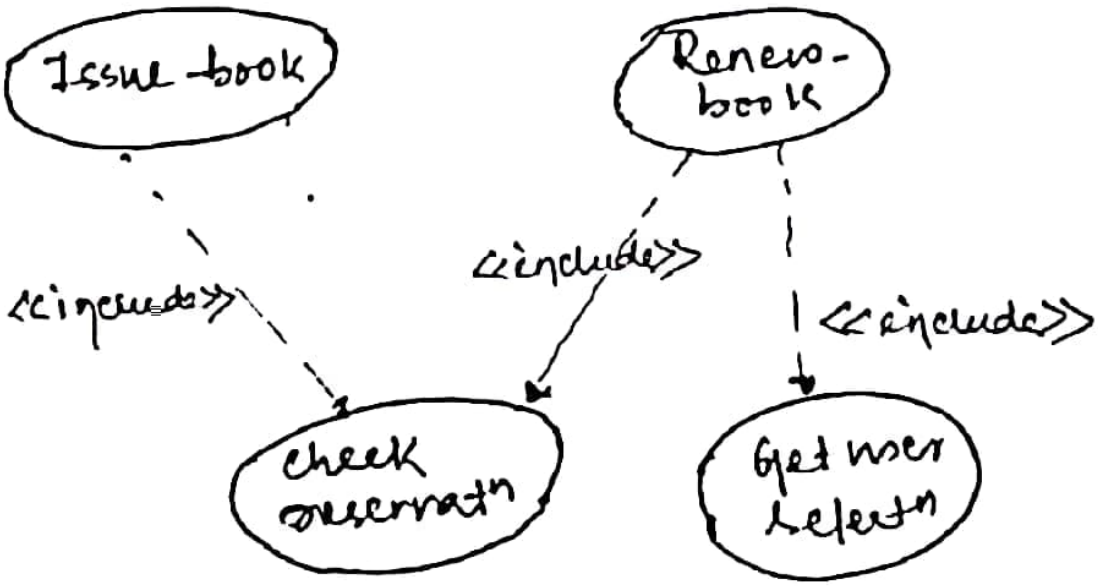
⇒ Factoring of commonality among use-c

(1) Generalizatn



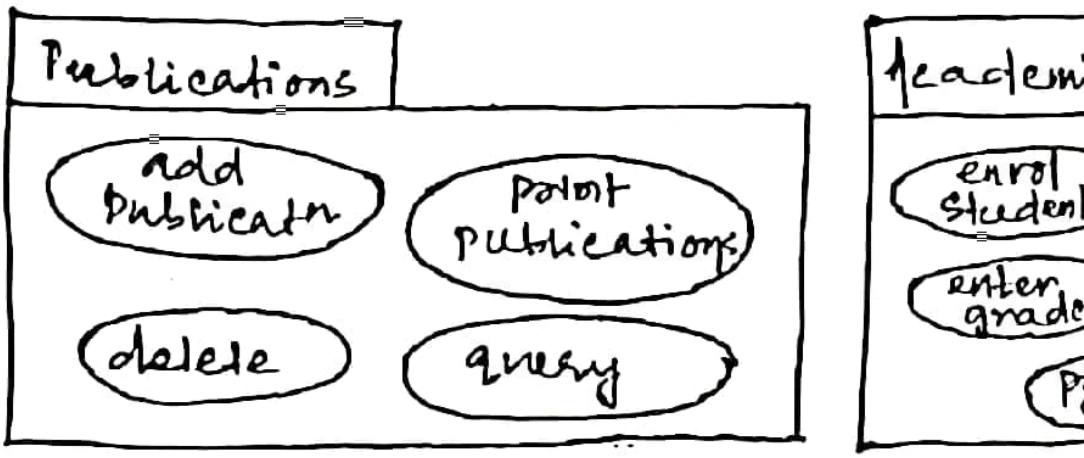
(2) Includes

involves one use-case including the behav use-case.



(3) Extends

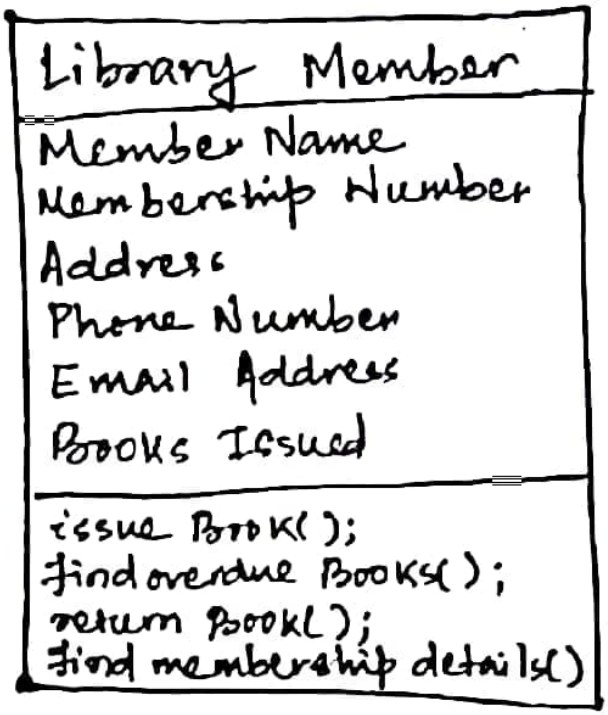
Use-case Packaging.



CLASS DIAGRAMS

describes the static structure of a sy
of a number of class diagrams and thei

eg:



Relationship between Objects / Classes

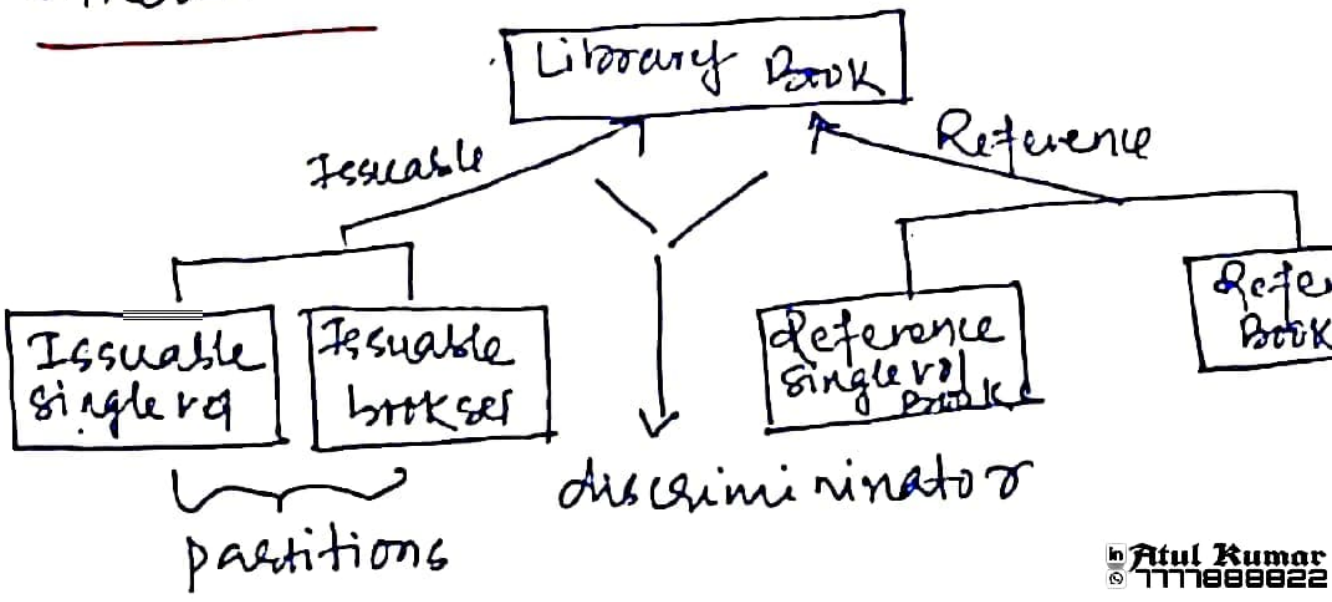
Aggregation.

If instance of one object contains m
some other objects then aggregation
exists between the composite and the
object.

(3) Composition

Stricter form of aggregation, where pa
existence-dependent on whole.

(4) Inheritance



(5) Object Diagrams

also known as the instance diagram, s
shot of the objects in a system at a

- INTERACTION DIAGRAMS.

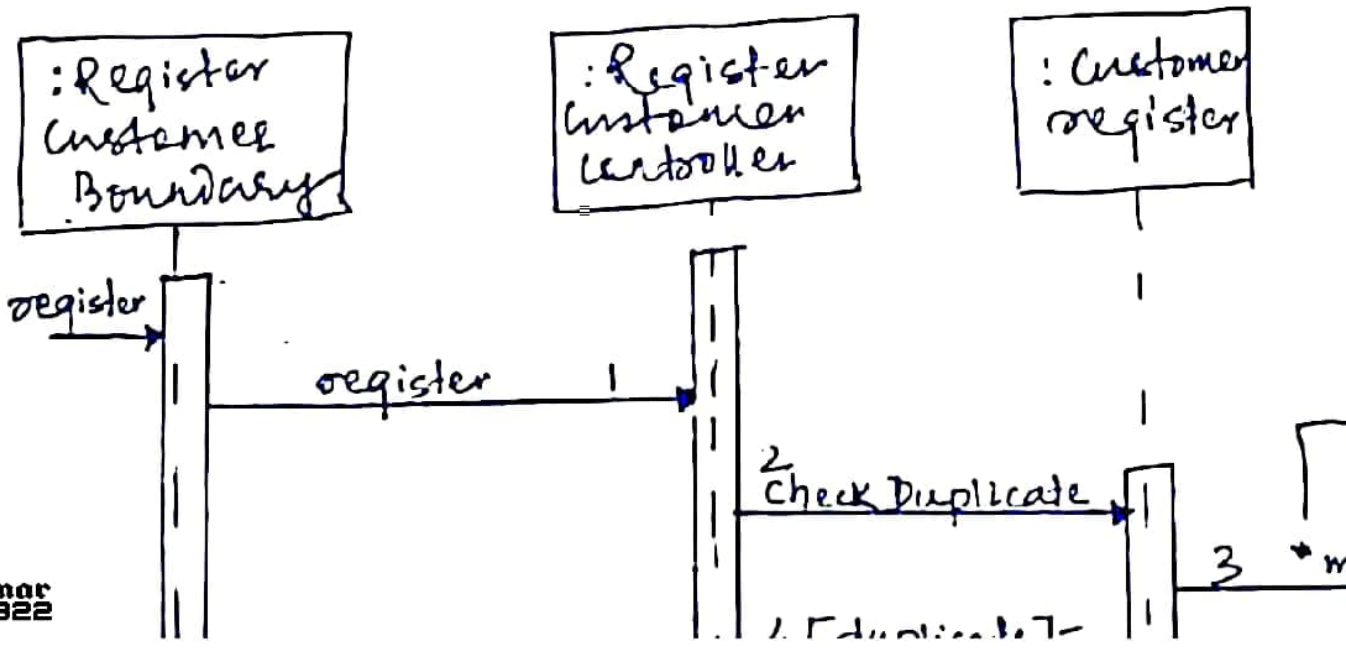
describe how groups of objects collabor
some behaviour (usually a single use-ca

- Sequence Diagrams
- Collaboration Diagrams

➡ SEQUENCE DIAGRAM.

shows interaction among objects as a
chart.

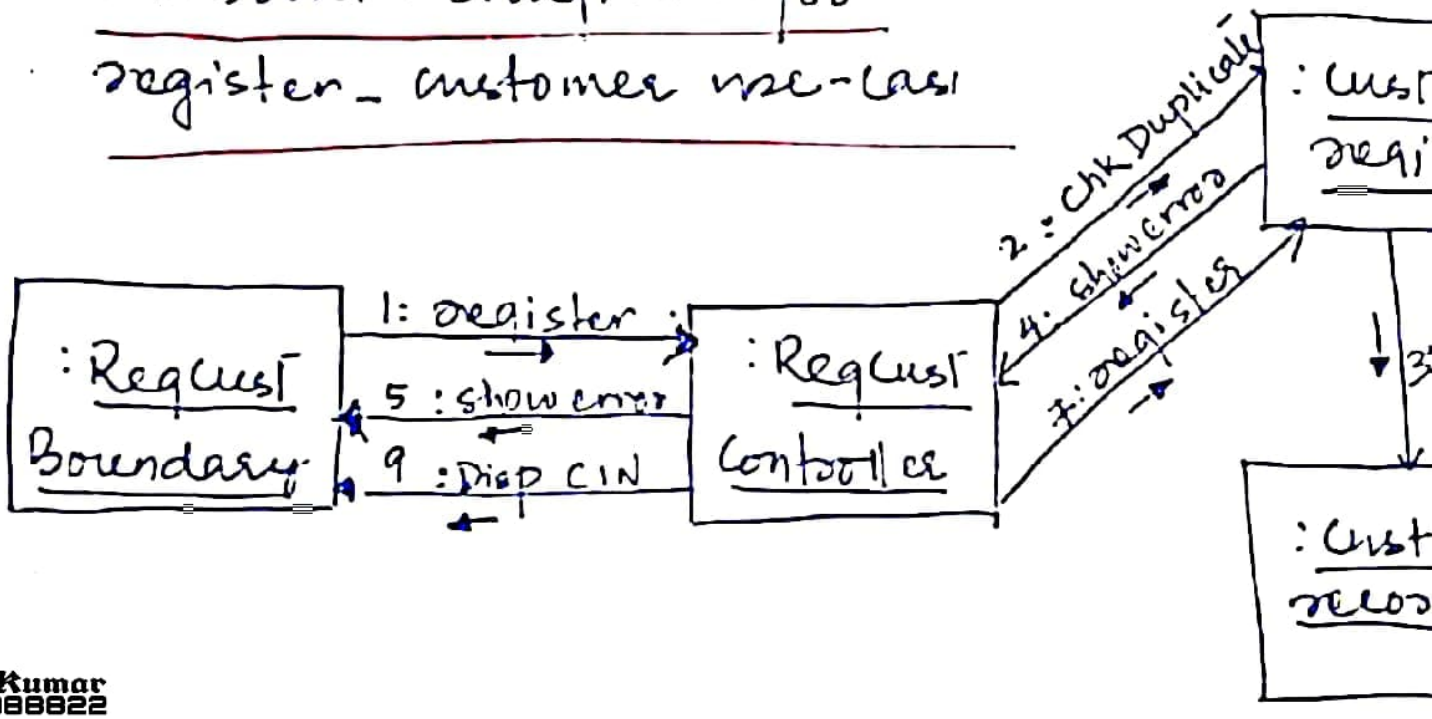
eg: Sequence diagram for Register-



⇨ Collaborat^{n} Diagram.

shows both the structural as well as
aspects. Here objects are collabors and lin
them is established as labelled arrows.
prefixed with sequence numbers to show

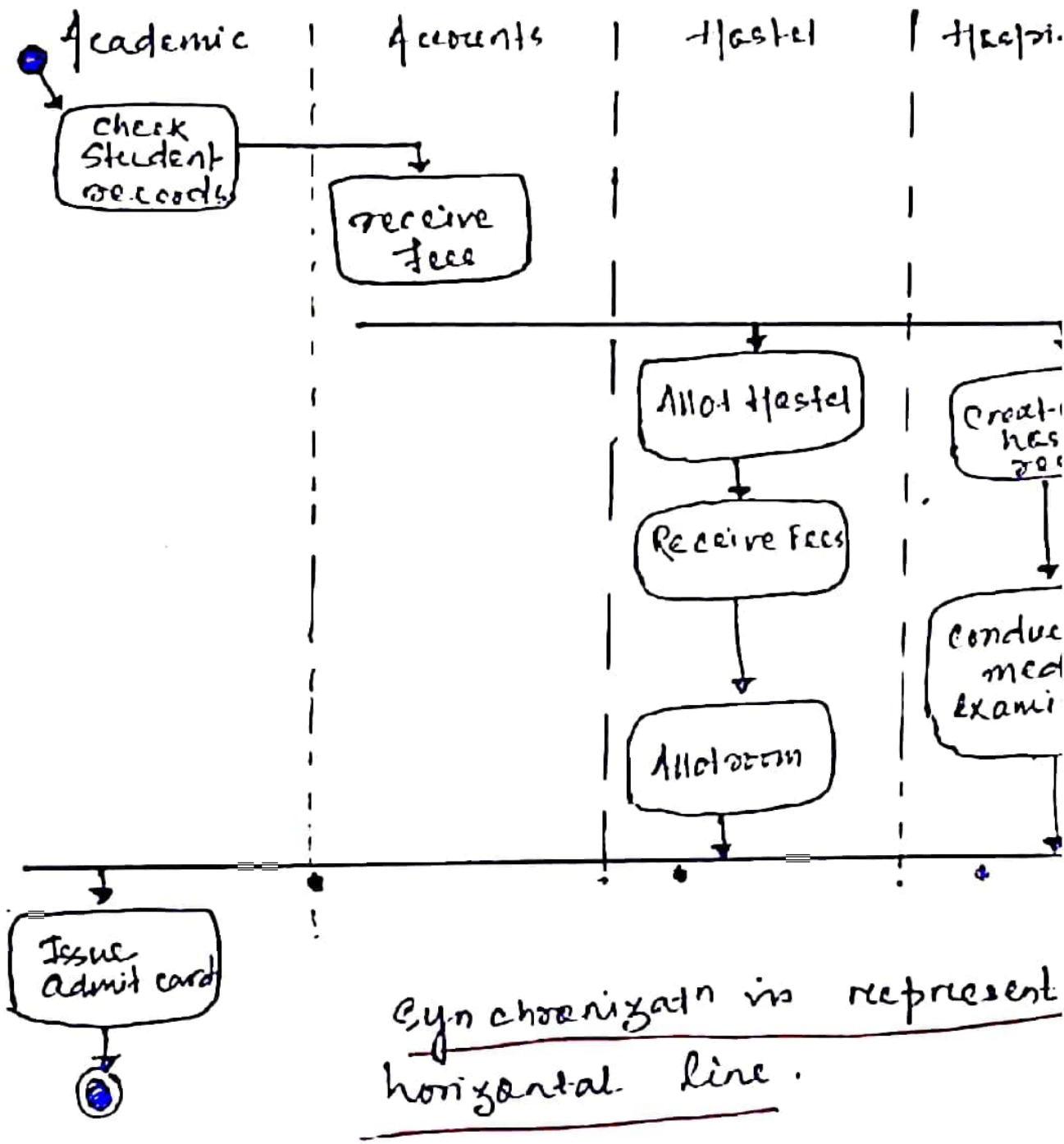
collaborat^{n} diagram for
register_customer use-case



■ ACTIVITY DIAGRAMS

also called swim-lane diagrams repre
activities, why represent parallel & syn
aspects, but may not correspond to meth

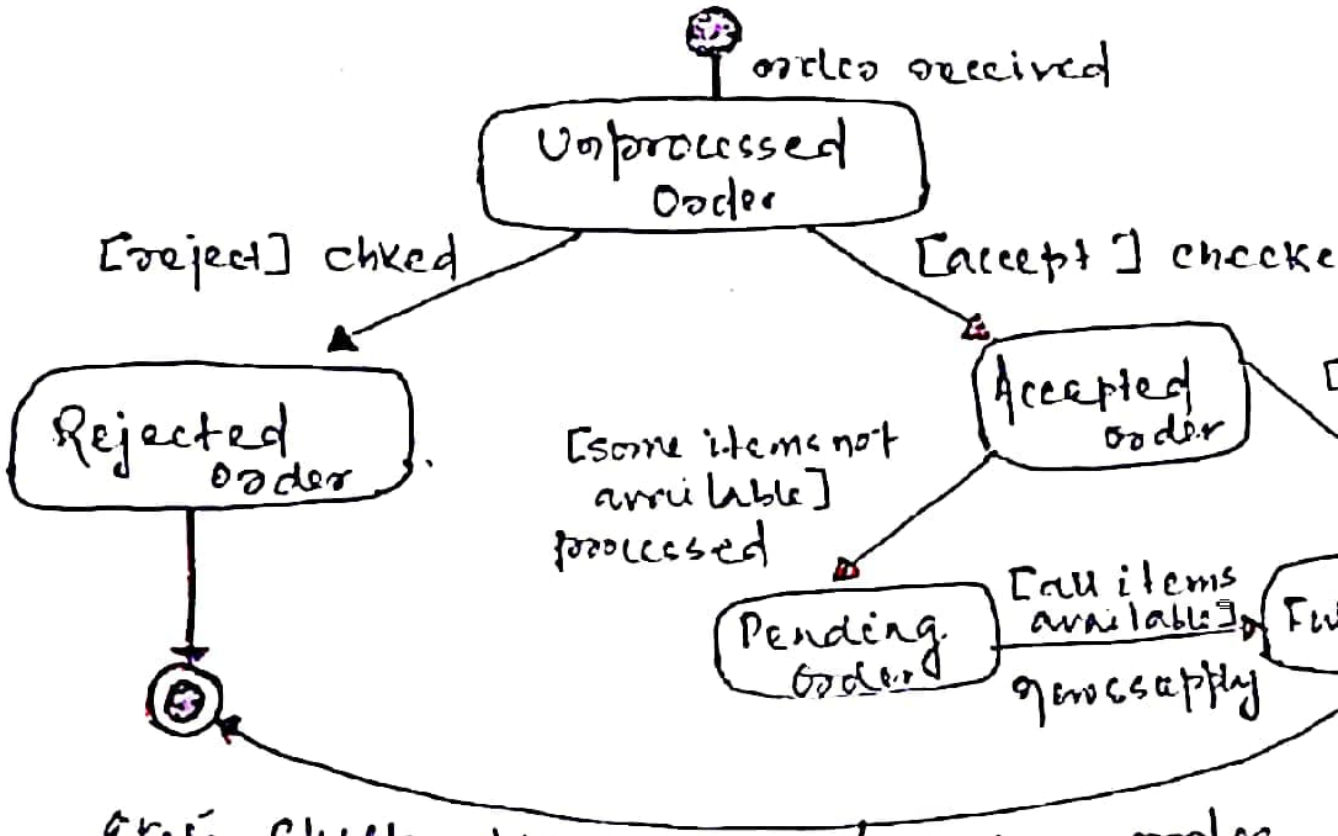
• An activity is a state with an interna
one or more outgoing transitions why a



Synchronization is represent horizontal line.

STATE-CHART DIAGRAM

used to model how the state of an during its life-time, based on the specific event.



State-Chart diagram for our order

OO-Patterns

Objects are identified by examining nou
description and transitive verbs as methods

Domain-Modelling

Objects and relationship. 3types

(1) Boundary Objects: Interact with d
do not perform any processing. only
and formatting.

(2) Entity Objects: hold info such as
and files and are responsible for brain