Page No:
Date:

- Software Engineering - A systematic collection of good program development practices and techniques
  - An engineering approach to develop software
- Software product - Computer programs and associated documentation such as requirements, specifications, design models and user manuals
  - May be developed under/for a specific customer or may be developed for general market.
  - Types
    (i) Generic - Developed to be sold to a range of different customers
    (ii) Custom - Developed for a single customer according to their specification

Without using software engineering principles, it would be difficult to develop large programs. In industry, it is usually needed to develop large programs to accomodate multiple functions. A problem with developing such large commercial programs is that the complexity and the difficult levels of the programs increase exponentially with their size.

* Software Engineering principles use two important techniques to reduce problem complexity : abstraction & decomposition

Abstraction principle implies that problems can be simplified by omitting irrelevant details. In other words, main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose

→ Powerful way of reducing complexity of the problem.

- Decomposition - A complex program is divided into several smaller problems and then smaller problems are divided one by one.

Characteristics of a good software

(i) Operational - Budget, Usability, Efficiency

(ii) Translational - Portability, Interoperability, Reusability

(iii) Maintainance - Modularity, Flexibility, Scalability

Need of software engineering

Because of higher rate of change in user requirements and environment on which the software is working.

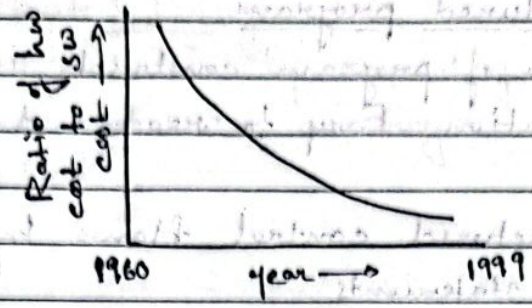
- Large Software - As the size of software become large, engineering has to step to give it a scientific process

- Scalability - If the software process were not based on scientific and engineering concepts, it would be easier to recreate new software than to scale existing one.

- Dynamic Nature - The always growing and adapting nature of software hugely depends upon the environment where user works.

Software Crisis

- Software products:
  - fail to meet user requirements
  - frequently crash
  - expensive
  - difficult to alter, debug and enhance
  - often delivered late
  - use resources non optimally

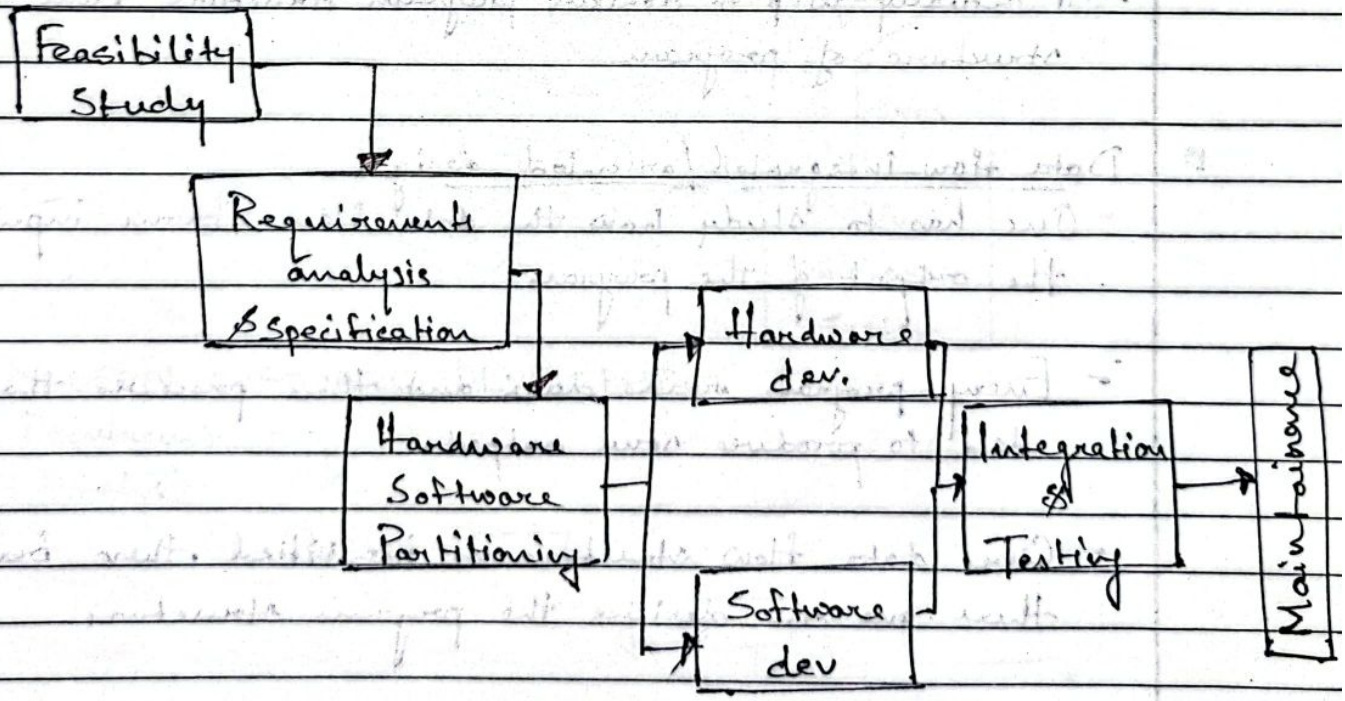


Relative Cost of hardware and softwares

- Factors contributing to software crisis
  - Larger problems
  - Lack of adequate training in s/w
  - Increasing skill shortage
  - Low productivity improvements

- Programs v/s software products

Programs	Software products
- Small in size	- Large
- Ad-hoc development	- Systematic development
- Single developer	- Team of developers
- Lacks proper documentation	- Well documented and user manual prepared



Project Management

Page No:
Date:

Features of a structured program

- It uses three types of program constructs i.e. selection, sequence and iteration. Easy to read and understand.
- They avoid unstructured control flows by restricting the use of GOTO statements
- It consists of well partitioned set of modules
- Uses single entry, single exit program constructs such as if-then-else, do while etc.

STRUCTURED PROGRAMMING PRINCIPLE EMPHASIZES DESIGNING NEAT CONTROL STRUCTURES FOR PROGRAMS

Data structure oriented design

- It is important to pay more attention to design of data structure of program rather than the design of its control structure.
- It actually help to derive program structure from data structure of program.

Data flow integrated/oriented design

- One has to study how the data flows from input to the output of the program
- Every program reads data and then processes that data to produce some output.
- Once data flow structure is identified, then from there one can derive the program structure.

Page No:
Date:

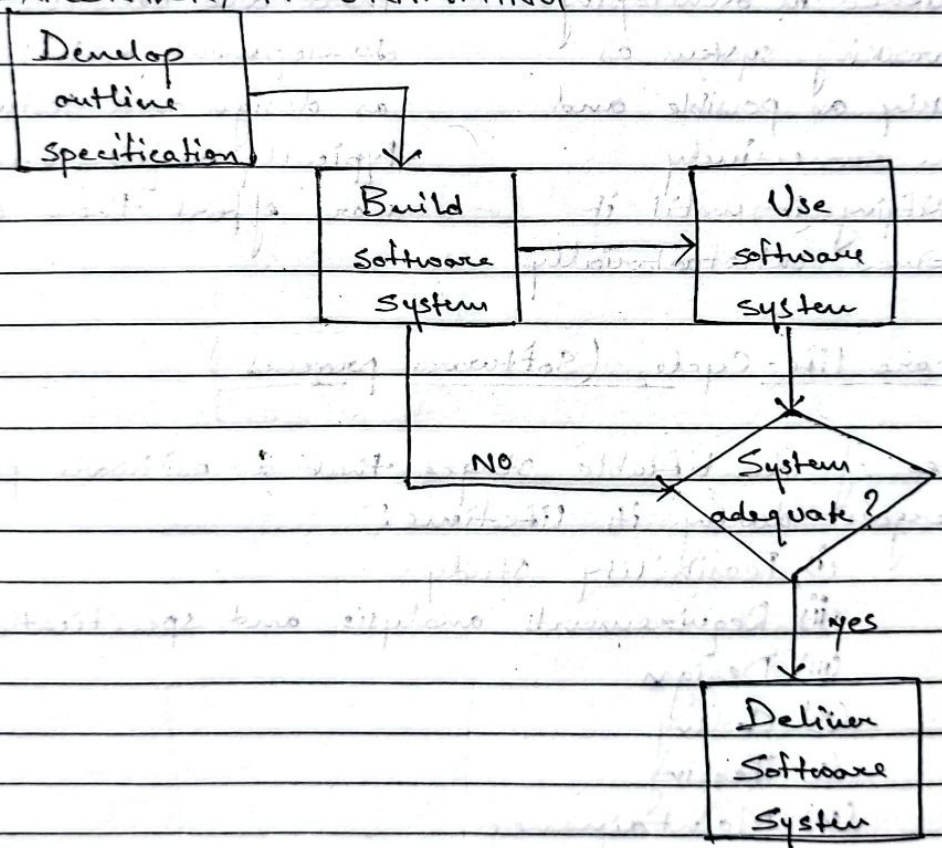
Object Oriented Design

- It has intuitively appealing design approach in which natural objects occuring in the problem are first identified

- Relationships among objects are determined

- Each object essentially acts as a data hiding entity

EXPLORATORY PROGRAMMING



MODERN SOFTWARE DEVELOPMENT PROCESS

Requirement Analysis → Design → Coding → Testing → Maintainance

Exploratory Style	Modern Style
• Emphasis on error correction.	• Emphasis on error prevention
• Coding was considered synonymous with software development.	• Coding is regarded as only a small part of the overall software development activities
• Believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorially	• There are several development activities such as design and testing which typically require much more effort than coding

Software life Cycle (Software process)

Series of identifiable stages that a software product undergoes during its lifetime:

(i) Feasibility Study
(ii) Requirements analysis and specification
(iii) Design
(iv) Coding
(v) Testing
(vi) Maintainance

Life Cycle Model

Description or diagrammatic model of software life cycle.

- Identifies all the activities required for product development
- establishes a precedence ordering among the different activities
- Divide life cycle into phases

Page No:
Date:

Why life cycle model?

- A written description
  - forms a common understanding of activities among software developers
  - helps in identifying inconsistencies, redundancies and omissions in the development process
  - Helps in tailoring a process model for specific projects

- Processes are tailored for special projects
- Development team must must identify suitable life cycle model and must adhere to it which helps in the development of software in systematic and disciplined manner.

- When program developed by single programmer, he has the freedom to decide his exact steps.

- Life cycle model defines entry and exit criteria for every phase. Phase is considered complete only when exit criteria are satisfied.

Page No:
Date:

Agile Methodology

It is a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project. Both development and testing activities are concurrent unlike the waterfall mode.

- Agile software development emphasizes on four core values:
  (i) Individual and team interactions over processes and tools
  (ii) Working software over comprehensive documentation
  (iii) Customer collaboration over contract negotiation
  (iv) Responding to change over following a plan

- Five reasons
  (i) Early time to market need
  (ii) Ever changing requirement - difficult to adopt
  (iii) Basic systems in place (than waterfall)
  (iv) Need of ~~customer~~ continuous interaction with customer
  (v) Waterfall was heavy on documentation

Agile Testing Methodology

- Scrum
- Crystal Methodologies
- DSDM (Dynamic Software Development Method)
- Feature driven development (FDD)
- Learn Software Development
- Extreme programming (XP)

Page No:
Date:

Agile Model	Waterfall Model
• It proposes incremental and iterative approach to s/w design.	• Development of s/w flows sequentially from start point to end point.
• Customer has early and frequent opportunities to look at the product and make decision and changes to the project.	• The customer can only see the product at the end of the project.
• Agile model is considered unstructured compared to the waterfall model.	• They are more secure because they are so plan oriented.
• Development process is iterative, and the process is executed in short weeks iterations.	• The development process is phased, and the phase is much bigger than iteration
• Testers and developers work together.	• Testers work separately from developers

Scrum Methodology

- Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.
- The scrum framework consists of scrum teams and their associated roles, events, artifacts and rules. All events are time boxed events such that every event has a maximum duration.

Sprint

The heart of scrum is sprint, a time box of two weeks or one month during which a potentially releasable product

increment is created. A new sprint starts immediately after the conclusion of the previous sprint.

Sprints consists of the -
(i) Sprint planning
(ii) Daily scrum meeting
(iii) The development work
(iv) The sprint review
(v) The sprint retrospective

- **Roles in Scrum**
  (i) Scrum Master : Master is responsible for setting up the team, sprint meeting and removes obstacles to progress

  (ii) Product Owners : The product owner creates product backlog, prioritizes the backlog and is responsible for the delivery of the functionality at each iteration.

  (iii) Scrum Team : Team manages its own work and organizes the work to complete the sprint or cycle.

- **Artifacts in scrum**
  (i) Product backlog
  (ii) Sprint backlog
  (iii) Sprint/release burn down chart
  (iv) Increment

Page No:
Date:

eXtreme Programming (XP)

eXtreme Programming is based on the five values

- Communication
- Simplicity
- Feedback
- Courage
- Respect

Extreme Programming is a systematic approach with a set of values, rules and practices for rapidly developing high quality software that provides the highest value for customers.

It is very useful when there is constantly changing demands or requirements from the customers or when they are not sure about the functionality of the system.

It advocates frequent releases of the product in short development cycles which inherently improves productivity of the system and also introduces a checkpoint where any customer requirements can be easily implemented

The XP develops software keeping customer in the target.

- **Basic principles of extreme programming**

- Business requirements are gathered in terms of stories. User stories are simple and informal statements of the customer about the functionalities needed. All these stories are stored in a place called Parking Lot.

- Now based on user stories, project team proposes metaphors. Metaphors are a common vision of how the system would work (design)

- The development team may decide to build a spike (like prototype) for some features.

- In this type of methodology, releases are based on the shorter cycles called iterations with span of 14 days time period. Each iteration includes phases like coding, unit testing and system testing where at each phase some minor or major functionality will be built in the application.

- Activities

(i) Coding – XP argues that code is a crucial part of any development system process since without code it is not possible to have a working system.

(ii) Testing – XP places high importance on testing and considers it be the primary means for developing a fault free software

(iii) Listening – The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not need to have an in-depth knowledge of the domain of the system under development.

(iv) Designing – Without a proper design, the system implementation becomes very complex and the dependencies within the system becomes more numerous and becomes difficult to comprehend the solution.

(v) Feedback – It recognises the importance of user feedback in understanding the exact customer requirements. The time that elapses b/w the development of a version and collection of feedback on it is critical to learning and making changes

(vi) Simplicity – Attention should be focussed on specific features that are immediately needed and making them work rather than devoting time and energy on speculations about future requirements.

Classical Waterfall Model

Classical Waterfall Model divides lifecycle into phases –
- Feasibility Study
- Requirement analysis and specification
- Design
- Coding and unit testing
- Integration and system testing
- Maintainance

* Phases b/w feasibility study and testing requires the maximum efforts and is known as the development phase.
* Maintainance phase requires max. efforts among all life cycle phases
* Among development phases, testing phase requires or consumes max. efforts.

- Most organisations usually define
  - Standards on the outputs (deliverables) produced at the output of every phase
  - Entry and exit criteria for every phase

- Feasibility Study – The main idea of feasibility study is to determine the product is financially viable and technically feasible. First gather customers requirements by analysing input and output data. Understand the necessary processes and system constraints. Then explore possible solutions evaluating each based on required resources, development cost and time.

- Requirement analysis and specification: The aim of this phase is to understand and document the customers exact requirements through two activities
  - Requirement gathering and analysis
  - Requirement specification.

- Goals of requirement analysis
  - Collect data from the customers
  - Analyse the data to clarify needs
  - Identify and resolve inconsistencies or incomplete requirements

- Requirements gathering methods
  - One-on-one interviews : With users, management and stakeholders
  - Group interview : Highlights agreements and differing issues
  - Questionnaires / Surveys - Used for graphically dispersed stakeholders
  - User Observations - Observe user interaction at different times
  - Document analysis - Review current systems and processes.

Initial data may contain contradictions & ambiguities which are resolved through discussions. The finalized requirements are documented in Software Requirement Specifications (SRS). Analysts handle this phase.

- Design

It transforms the requirement specification into a form suitable for implementation. It involves deriving the software architecture from the SRS document using two approaches:
Traditional and Object Oriented

Traditional / Procedural design approaches - Based on data flow modelling. It consists of two parts:
- Structured Analysis
  - Identify functions and data flow
  - Decompose functions to sub functions
  - Use data flow diagrams (DFDs)
- Structured design
  - Decompose the ~~modules~~ system into modules and represent model interactions
  - Design algorithms, data structures and objects, refining relationships between or among objects.

Object Oriented Design - Various problems that occur in the problem domain and the solution domain are first identified and the different relationships that exist among them are identified. The object structure is further refined to have a detailed approach.

Advantages
- Lower development effort and time
- Better maintainability

- Implementation Translates software design to source code

During implementation phase,
- each module of design is coded
- each module is unit tested
  - tested independently as stand alone unit and debugged
  - each module is documented

Purpose - Test if individual modules work correctly

- Integration and system testing
  - Integrate modules incrementally, testing the system after each step
  - System testing ensures that the system meets requirements from the SRS

- Maintainance

Requires more effort than development (typically 40:60 ratio)

- Corrective - Fix errors missed during development
- Perfective - Improve and enhance system functionality
- Adaptive - Port software to new requirements

Iterative Waterfall Model

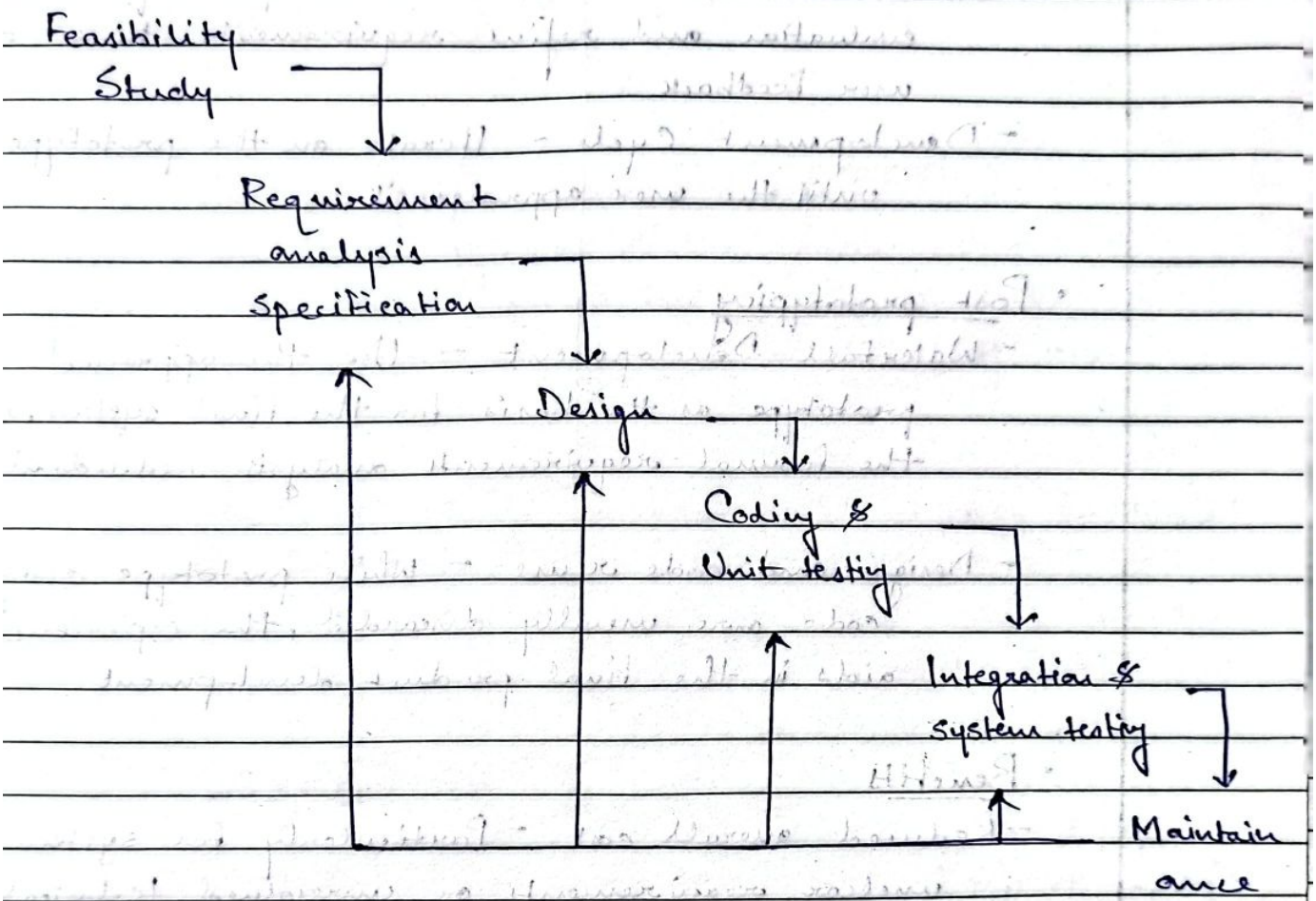
The classical waterfall model is idealistic, assuming no defects are introduced during any phase. In practice, defects often arise and detected later in the life cycle, necessitating rework of previous phases.

(i) Defect Detection - Defects often go unnoticed until later phase

(ii) Feedback paths - To address defects, feedback paths are necessary to visit and revise earlier phases.

(iii) Error detection - Errors should ideally be detected in the same phase they are introduced to simplify corrections.

(iv) Phase containment of errors - This principle involves detecting and addressing errors as close to their introduction point as possible to minimize rework



Prototyping Model

Before actual development, it is essential to build a working prototype of the system. A prototype is a simplified, often non functional version of the system with limited capabilities

- Reasons for prototyping
  - Demonstrate input data formats, messages, reports or interactive dialogs to the customers
  - Address technical issues and major design decisions such as hardware response times or algorithm efficiency
  - The prototype helps refine requirements and improve design.

- Prototyping process
  - Start with approximate requirements - Create a quick design using shortcuts and dummy functions
  - Customer feedback - Submit the prototype for evaluation and refine requirements based on user feedback
  - Development Cycle - Iterate on the prototype until the user approves it.

- Post prototyping
  - Waterfall Development - Use the approved prototype as the basis for the final system, rendering the formal requirements analysis redundant
  - Design and code reuse - While prototype design and code are usually discarded, the experience aids in the final product development

- Benefits
  - Reduced overall cost - Particularly for systems with unclear requirements or unresolved technical issues, prototyping can prevent costly redesigns and change requests later in the development process.

Evolutionary Model

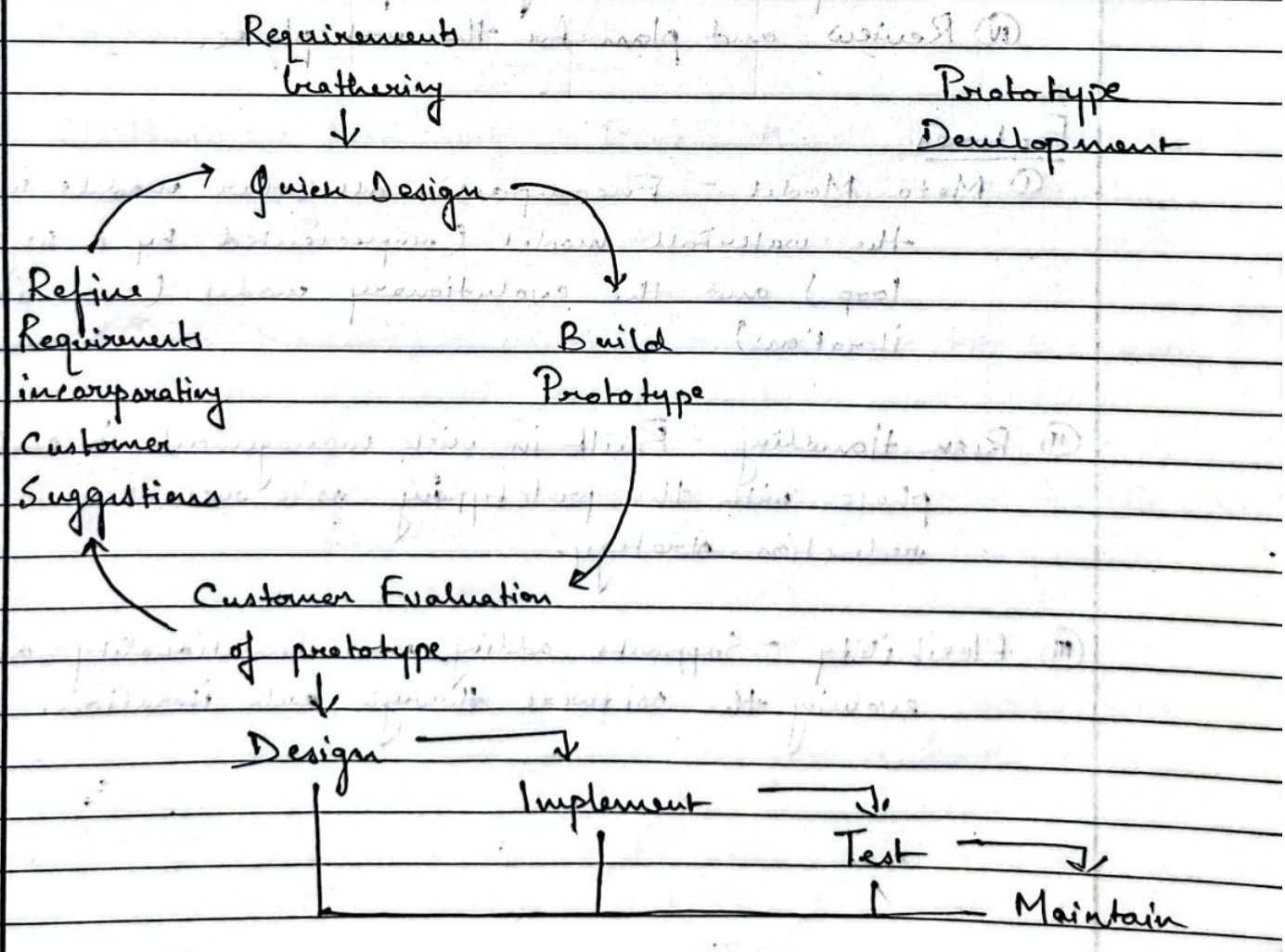
The evolutionary model allows users to experiment with a partially developed system early on, helping refine user requirements before the full system is built.

- Key features
  - The system is divided into modules that are incrementally implemented and delivered.

- Core functionalities are developed and tested thoroughly reducing errors in the final product.
- Each new version adds new functionalities or enhances existing ones, allowing the system to gradually evolve into the final product.

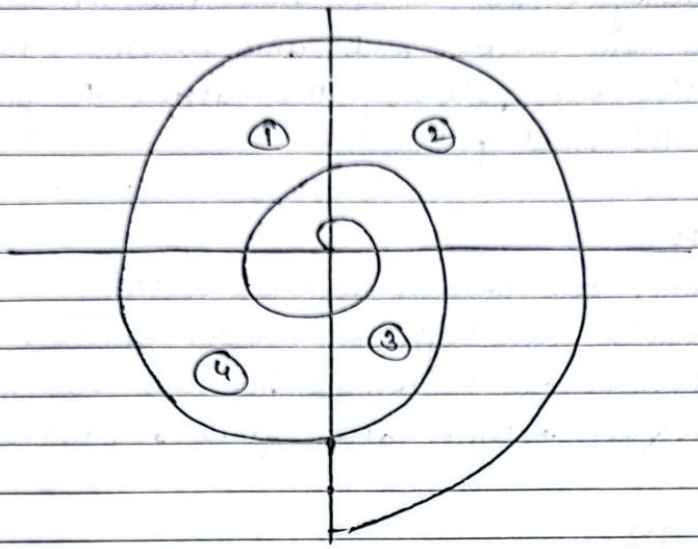
- **Benefits**
  - Users can provide feedback earlier
  - Training can begin on earlier versions
  - Markets can be created for new functionalities
  - Frequent releases allow quick resolution of issues.

- **Challenges**
  - Difficult to divide complex problems into functional units for incremental delivery.
  - Best suited for large projects where models can be developed incrementally



Page No:
Date:

Spiral Model



(I) Determine objectives and identify alternative solutions
(II) Identify and resolve risks
(III) Develop next level of the product
(IV) Review and plan for the next phase.

Features

(I) Meta Model - Encompasses all other models including the waterfall model (represented by a single loop) and the evolutionary model (succeeding iterations)

(II) Risk Handling - Built in risk management in each phase with the prototyping as a risk reduction strategy.

(III) Flexibility - Supports adding new functionality and evolving the software through each iteration.

Page No:
Date:

- Advantages
  - Enhanced risk management
  - Suitable for large, mission-critical projects
  - Allows early software delivery
  - Strong documentation and approval control

- Disadvantages
  - Can be costly
  - Requires specialized expertise on risk analysis
  - Success depends heavily on risk assessment
  - Not ideal for smaller projects.

Rapid Application Development (RAD)

It is an incremental model where components or functions are developed in parallel like mini-projects. These developments are time boxed, delivered and then assembled into a working prototype, allowing early customer feedback.

- Phases of the RAD Model

(i) Business Modelling - Design the business model based on information from business activities to capture a complete view of the process.

(ii) Data Modelling - Identify and analyse the necessary data based on the business model

(iii) Process Modelling - Plan the processing and functionality or modification of data ~~based on business model~~ to achieve business functionality

(iv) Application Modelling - Develop the application with coding and implement using automation tools

(v) Testing and turnover - Perform testing activities on the developed applicatio

- Advantages
  - Fast development and delivery
  - Minimal testing needed
  - Progress is easy and easy to track
  - Cost effective and resource efficient

- Disadvantages
  - Required highly skilled resources
  - Constant client feedback is necessary
  - Automated code generation can be expensive
  - Hard to manage and not ideal for large-long term projects.

Agile Model

It combines iterative and incremental processes with a focus on adaptability and customer satisfaction through rapid delivery of working software. Agile breaks the product into small incremental builds, delivered in iterations typically lasting 1-3 weeks. Each iteration involves teams working on ~~teams~~ planning, requirements, design, coding, testing and showcasing the product to stakeholders

- Advantages
  - Realistic approach to development
  - Promotes teamwork and cross training
  - Requires minimal ~~training~~ resources and delivers early working solutions.

- Disadvantages
  - Not ideal for complex dependencies
  - Requires overall plan, agile leadership & PM practices
  - Strict delivery management required to meet deadlines.

Page No:
Date:

Requirement Engineering

- Requirement Gathering – Also popularly known as the requirement elicitation. The primary objective of the requirements gathering task is to collect the requirements from the stakeholders

Methods:
(i) Studying existing documentations
(ii) Interviews
(iii) Task, scenario

- Requirement Analysis – The main purpose of the requirement analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

- During requirement analysis, the analyst needs to identify and resolve three main types of problems in the requirements:
  - Anomaly – Anomaly is an ambiguity in the requirement. When a requirement is anomalous, several interpretations of that requirement are posible. Any anomaly in any of the requirements can lead to the development of an incorrect system.
  - Inconsistency – Two requirements are said to be inconsistent, if one of the requirements contradicts the other.
  - Incompleteness – An incomplete set of requirements is the one in which some requirements have been overlooked

Page No:
Date:

Software Requirement Specification (SRS)

It is a formal document capturing all user requirements and serves multiple audiences like developers, testers and project managers.

- Key characteristics of a good SRS
  - (i) Concise, unambiguous and complete
  - (ii) Implementation - independent
  - (iii) Traceable - Requirements linked to design elements and vice versa
  - (iv) Modifiable and well structured
  - (v) Verifiable - All requirements must be tested
  - (vi) Identification of response to undesired events - Describe ~~events~~ system behaviour under conditions

- Important Components of an SRS Document
  - (1) Functional requirements
    Clearly describes each function that the system will perform along with input/output data.
  - (2) Non functional requirements
    - External interfaces
    - Performance, security, maintainability, portability and usability
    - Design and implementation constraints
  - (3) Goals of implementation
    General future proofing ideas like easy support for new device or reusability considerations
  - (4) Design and implementation constraints
    Specific limitations like regulatory policies, technology stack, hardware interfaces etc.

- Functional requirements for organisation of SRS document

This section can classify the functionalities either based on the specific functionalities invoked by different users, or the functionalities that are available in different modes etc, what may be appropriate.

1. User class 1
   (a) Functional requirement 1.1
   (b) Functional requirement 1.2
2. User class 2
   (a) Functional requirement 2.1
   (b) Functional requirement 2.2

(Ex) Withdraw cash from ATM

R1 : Withdraw Cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R.1.1 : Select withdraw amount option
Input : "Withdraw amount" option selected
Output: User prompted to enter account type.

R.1.2 : Select account type
Input : User selects options from any one of the following
- savings / checking / deposit
Output : Prompt to enter amount.

R.1.3 : Enter required amount
Input : Amount to be withdrawn in integer values greater than 100 and less than 10000 in multiples of 100
Output : The requested cash and printed transaction statement.

• External Interface Requirements

(i) User Interface - Standards for GUI, navigation, error handling etc.

(ii) Hardware Interface - Systems interaction with the hardware components

(iii) Software Interfaces - Interaction with databases, OS and other applications

(iv) Communication Interfaces - Protocols, network server interactions etc.

• Other Non functional requirements

(i) Performance - Ex: Transactions per second

(ii) Safety - Avoid loss/damage due to software errors

(iii) Security - Identity authentication, data privacy etc.

Techniques for analysing complex logic

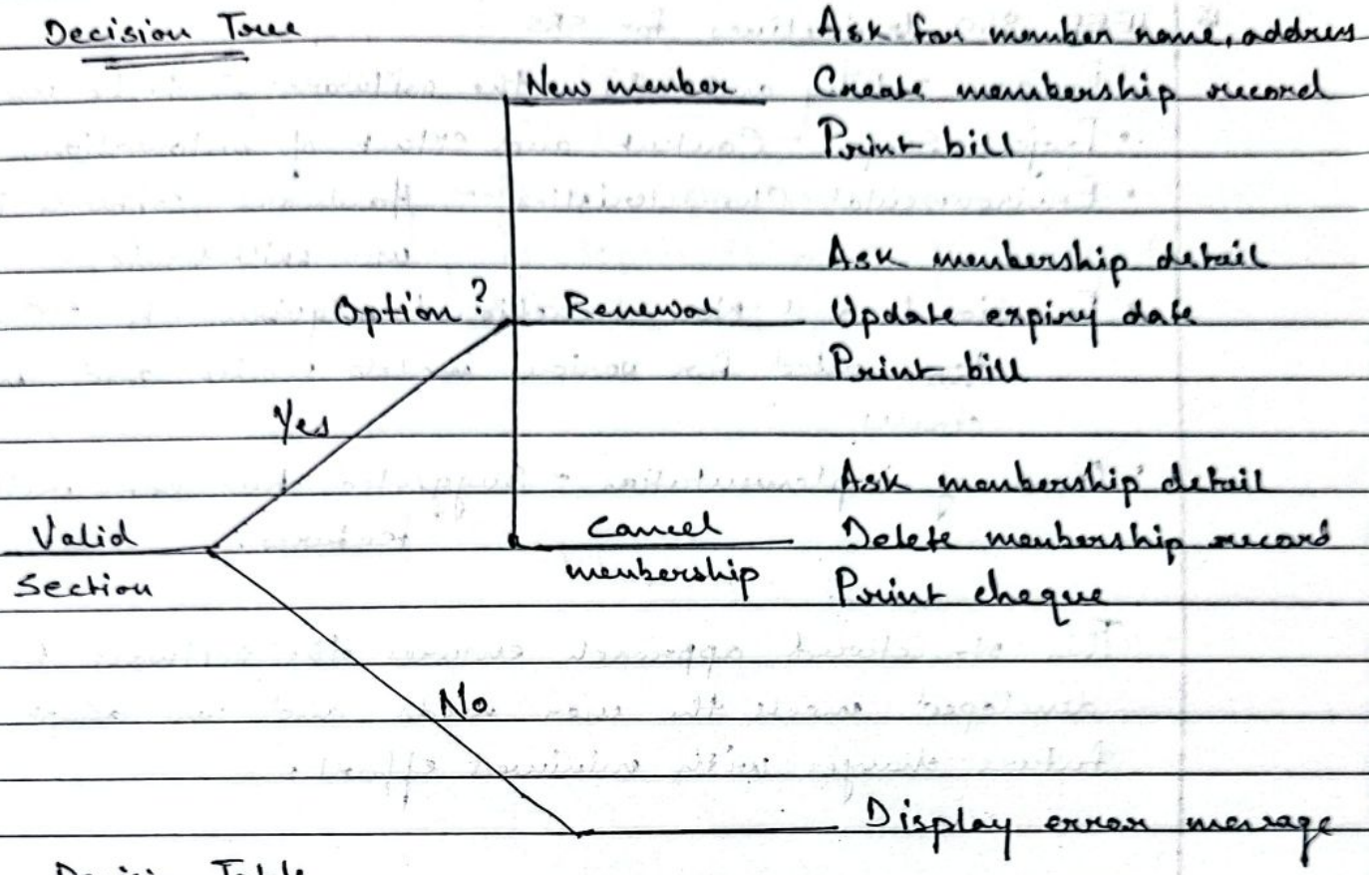
(i) Decision Tree

A visual flowchart for decision making logic and corresponding actions. Decision tables specify which variables are to be tested, and based on this what actions are to be taken depending upon the outcome of the decision making logic and the order in which decision making is performed.

Library Management System (LMS)

Page No:
Date:

Decision Tree



Decision Table

Conditions				
Valid Selection	NO	YES	YES	YES
New member	-	YES	NO	NO
Renewal	-	NO	YES	NO
Cancellation	-	NO	NO	YES
Actions				
Display error message	X			
Ask members name etc		X		
Build customer record		X		
Generate bill		X	X	
Ask membership detail			X	X
Update expiry date			X	
Print cheque				X
Delete record				X

Page No:
Date:

IEEE 830 Guidelines for SRS

- Purpose - Why and where the software is to be used.
- Project Scope: - Content and extent of automation.
- Environmental Characteristics - Hardware, software interaction user skill levels
- Functional and Non functional requirements - Clearly documented for various ~~models~~ modes and user classes
- Goals of implementation - Suggested but not critical features.

This structured approach ensures the software being developed meets the user needs and can adapt to future changes with minimal effort.

Page No:
Date:

Software Project Management

The main goal of software project management is to enable a group of developers to work efficiently / effectively towards the completion of a successful project.

Complexities

① Invisibility - Progress is hard to gauge until the software is operational.

② Changability - Software is easily modified, leading to the frequent changes due to business needs or user demands

③ Complexity - Even small software projects involve numerous interconnected parts (functions, state transitions, dependencies)

④ Uniqueness - Each software project have different challenges making it difficult to apply one-size-fits-all solutions.

⑤ Exactness of solutions - Software requires exact matches in parameters making reuse across projects difficult.

⑥ Team oriented and intellect intensive work - Software development is collaborative and requires high levels of intellectual effort, adding to its complexity

Responsibilities of a software project manager:

① Project Planning
- Estimating size, cost, effort and duration
- Scheduling resources and manpower
- Staffing and organising the project team.

- Risk management (identifying, analysing & mitigating risks)
- Quality assurance and config. management

(2) Project management and control
- Ensuring that project stays on track and is completed as planned.
- Managing risks and changes throughout project lifecycle.

Skills required for project managers

(i) Knowledge of project management techniques
(ii) Strong decision making abilities
(iii) Previous experience managing similar projects
(iv) Communication, team building and customer interaction skills
(v) Cost estimation, risk management and configuration management expertise

Project Planning Techniques

(i) Size estimation - Estimating the size of the software is critical as it directly affects cost, duration and effort calculations.

(ii) Scheduling - Once size, cost and effort are estimated, the schedule for manpower and other resource is created.

(iii) Risk Management - Identifying and planning for potential risks to avoid project delay.

(iv) Miscellaneous plans - Quality assurance, validation, verification and configuration management planning

Page No:
Date:

Sliding window planning

In large projects, its challenging to make accurate plans from the start, so sliding window planning involves planning the project in stages, allowing for flexibility and adjustment as the project progresses.

Software Project Management Plan (SPMP) Document

① Introduction - Objectives, functions, performance issue, management & technical constraints

② Project Estimates - Historical data used
- Estimation techniques used
- Effort, resource, cost and project duration estimates

③ Schedule - Work breakdown structure
- Task Network Representation
- Gantt Chart Representation
- PERT Chart Representation

④ Project Resources - People
- Hardware and Software
- Special resources

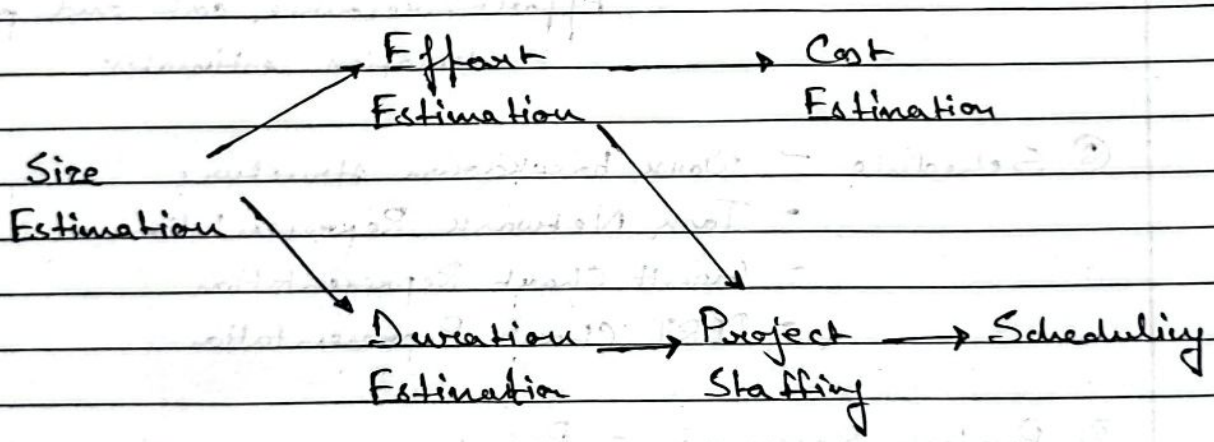
⑤ Staff Organisation - Team Structure
- Management Reporting

⑥ Risk Management Plan - Risk Analysis
- Risk Identification
- Risk Estimation
- Risk Abatement Procedures

⑦ Project tracking and control plan - Metrics to be tracked
- Tracking plan
- Control plan

⑧ Miscellaneous plans - Process tailoring
- Quality assurance plan
- Config. management plan
- Validation & Verification
- System Testing Plan
- Delivery, Installation and the maintainance plan

■ Precedence Ordering among planning activities



■ Metrics - project size estimation

① LOC (Lines of code) - Measures the size of a software project by counting the source instruction in the developed program. It excludes comment and header lines.

Shortcomings -

- Focus on coding alone - LOC assumes that coding effort determines the entire software development effort, which neglects important aspects like designing and testing.

- Subjectivity in Coding Style - LOC counts can vary due to different coding styles leading to inconsistency.
- LOC metric penalises modern programming techniques like code reusability and higher level languages, as fewer lines of code are required to achieve the same functionality.
- LOC measures lexical complexities but fails to address logical and structural complexities

(2) Function Point (FP) It was proposed by Albrecht in 1983 as an alternative to LOC. It measures software size based on functionality, making it easier to compute from problem specification.

FP Computation

Step 1: Compute the Unadjusted Function Point (UFP) based on the five characteristics: Inputs, Outputs, Enquiries, Files and Interfaces

UFP = (Inputs * 4) + (Outputs * 5) + (Inquiries * 4) + (Files * 10) + (Interfaces * 10)

Step 2: Refine UFP based on complexity (simple, average, complex) of each parameter.

Step 3: Adjust UFP for project specific complexities using Technical Complexity Factor (TCF)
Compute TCF as TCF: 0.65 + 0.01 * DI (where DI is the sum of 14 influence factors)

Final FP = UFP * TCP

Page No:
Date:

Shortcomings of FP

- FP does not account for algorithmic complexity. only the number of functions leading to possible misjudgment of effort required for complex ~~structo~~ features.
- The feature point metric was developed to address this by incorporating algorithm complexity

Empirical Cost Estimation Techniques

① Expert Judgement

- Estimations is based on the experience of the expert who divides the projects into units and provides an estimation for each.
- It is prone to human errors and individual bias and may overlook important factors

② Delphi Estimation

- Involves a team of experts providing independent estimates which are refined over multiple iterations without direct discussions to avoid influence.
- A coordinator facilitates by summarizing the results and rationale after which experts reestimate.
- Delphi avoids direct confrontation and encourages unbiased collective jugement.

COCOMO (Constructive Cost Estimation Model)

COCOMO is a widely used software cost estimation model. It helps to estimate the effort and time required to develop software based on project size, expressed in Kilo Lines of code (KLOC).

• COCOMO Project Categories

① **Organic** – Small teams working on well understood applications (Ex. data processing programs)

$$\text{Effort} = 2.4 \times (\text{KLOC})^{1.05} \text{ PM}$$
$$\text{Time } (T_{dev}) = 2.5 \times (\text{Effort})^{0.38} \text{ month}$$

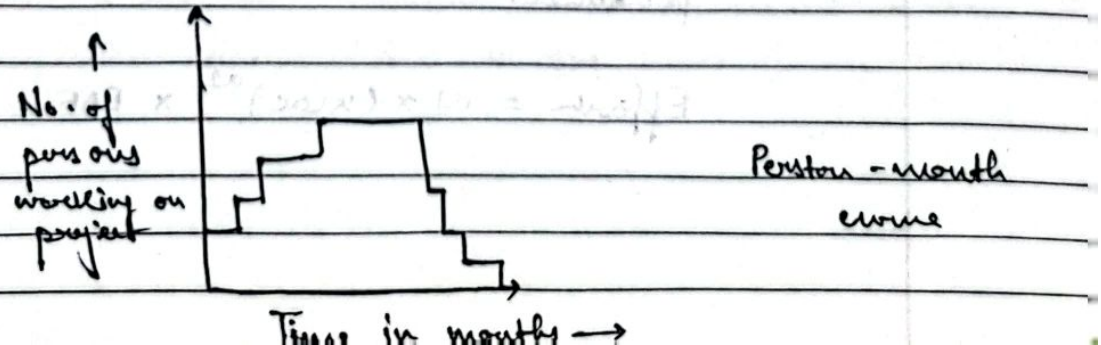
② **Semi detached** – Teams of mixed experience working on moderately familiar systems (Ex. Compiler, linker)

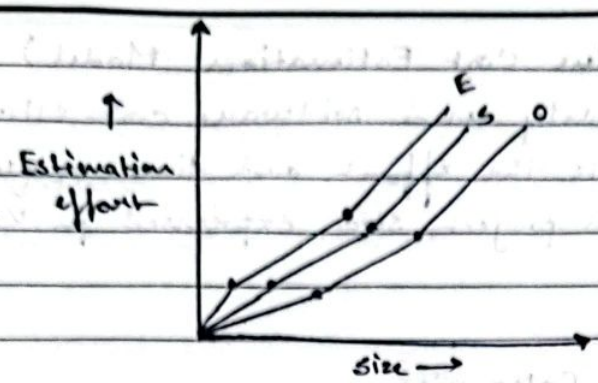
$$\text{Effort} = 3.0 \times (\text{KLOC})^{1.12} \text{ PM}$$
$$\text{Time } (T_{dev}) = 2.5 \times (\text{Effort})^{0.35} \text{ month}$$

③ **Embedded** – Complex, real time systems with tight hardware software integration (Ex. OS)

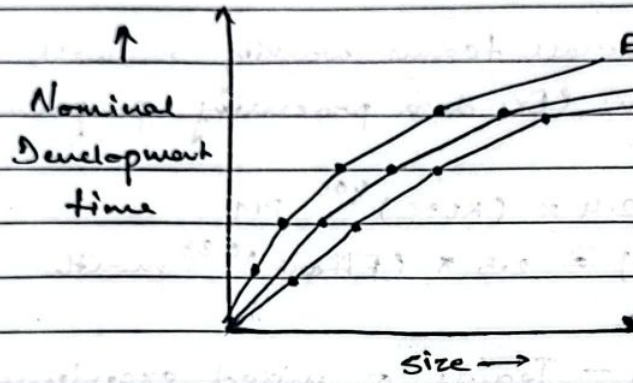
$$\text{Effort} = 3.6 \times (\text{KLOC})^{1.20} \text{ PM}$$
$$\text{Time } (T_{dev}) = 2.5 \times (\text{Effort})^{0.32} \text{ month}$$

• Person Month (PM) – One person month is the effort an individual can typically put in a month. The person month estimate implicitely takes into account the productivity





Effort v/s product size



Development time versus size

- Three stages of COCOMO

(i) Basic COCOMO – The single variable estimation model using project size to approximate effort and time.
- Predicts efforts (in person months) and time (in months) based on product size

$$\text{Effort} = a_1 \times (\text{KLOC})^{a2}$$
$$\text{Time} = b1 \times (\text{Effort})^{b2}$$

(ii) Intermediate COCOMO – Refines basic COCOMO by accounting for 15 cost drivers, such as reliability requirements and experience of personnel.

$$\text{Effort} = a1 \times (\text{KLOC})^{a2} \times \text{EAF}$$

(iii) Complete COCOMO – Extends the intermediate models by dividing the project into sub systems
- Each sub system is estimated separately (Ex - Organic, semi detached, embedded)
- Final cost is the sum of all sub systems cost.

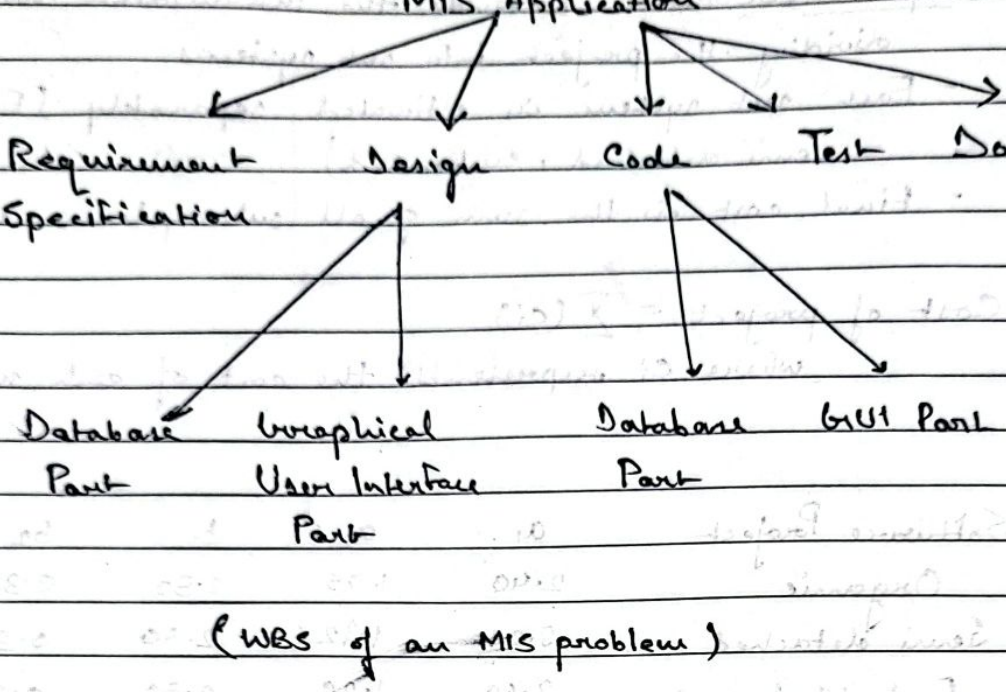
Cost of project = $\sum (c_i)$
where c_i represents the cost of each sub system.

Software Project	a_1	a_2	b_1	b_2
Organic	2.40	1.05	2.50	0.38
Semi detached	3.00	1.12	2.50	0.35
Embedded	3.60	1.20	2.50	0.32

Scheduling – The scheduling problem in essence, consists of deciding which tasks would be taken ~~by~~ up when and by whom.

A software manager must
(i) Identify major activities
(ii) Break activities into tasks
(iii) Determine task dependencies
(iv) Estimate time durations
(v) Represent tasks in activity network
(vi) Set tasks start/end dates
(vii) Identify the critical path (longest sequence of tasks)
(viii) Allocate resources to tasks

A work breakdown structure (WBS) decomposes activities into smaller tasks which are then scheduled. The task decomposition continues until the task duration is manageable (<2 weeks), hidden complexities are revealed or the scene of components is identified.



(WBS of an MIS problem)

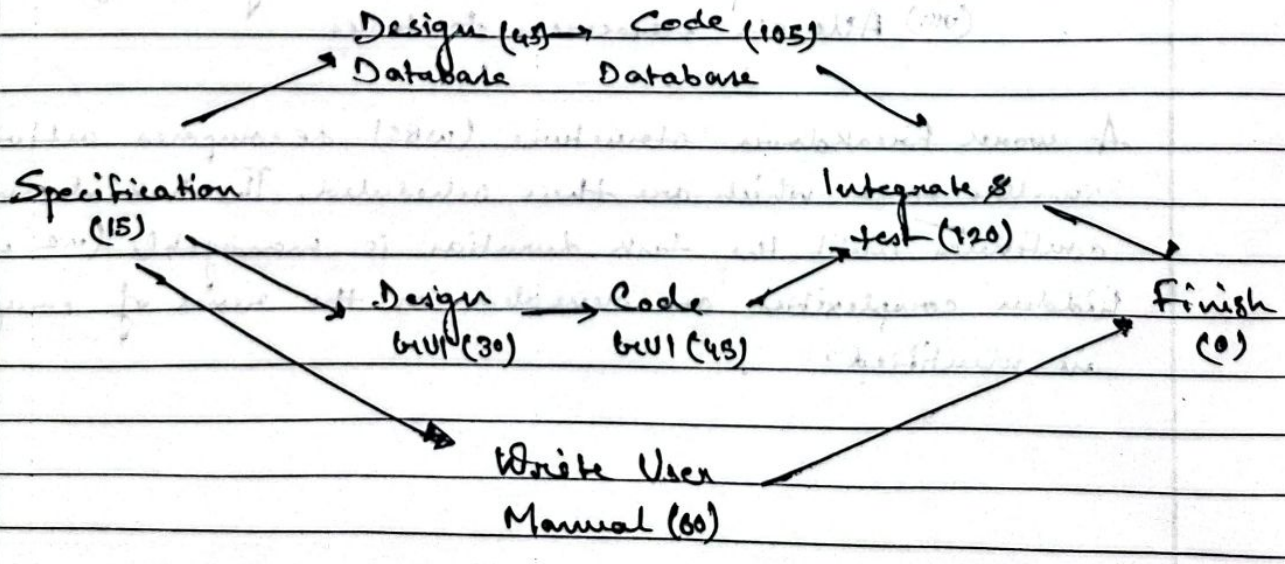
- **Activity Networks (AN)**

An activity network shows the different activities making up a project, their estimated durations and their interdependencies.

Two types of activity networks exist -

① Activity on Node (AoN) - Tasks on nodes
Dependencies on edges

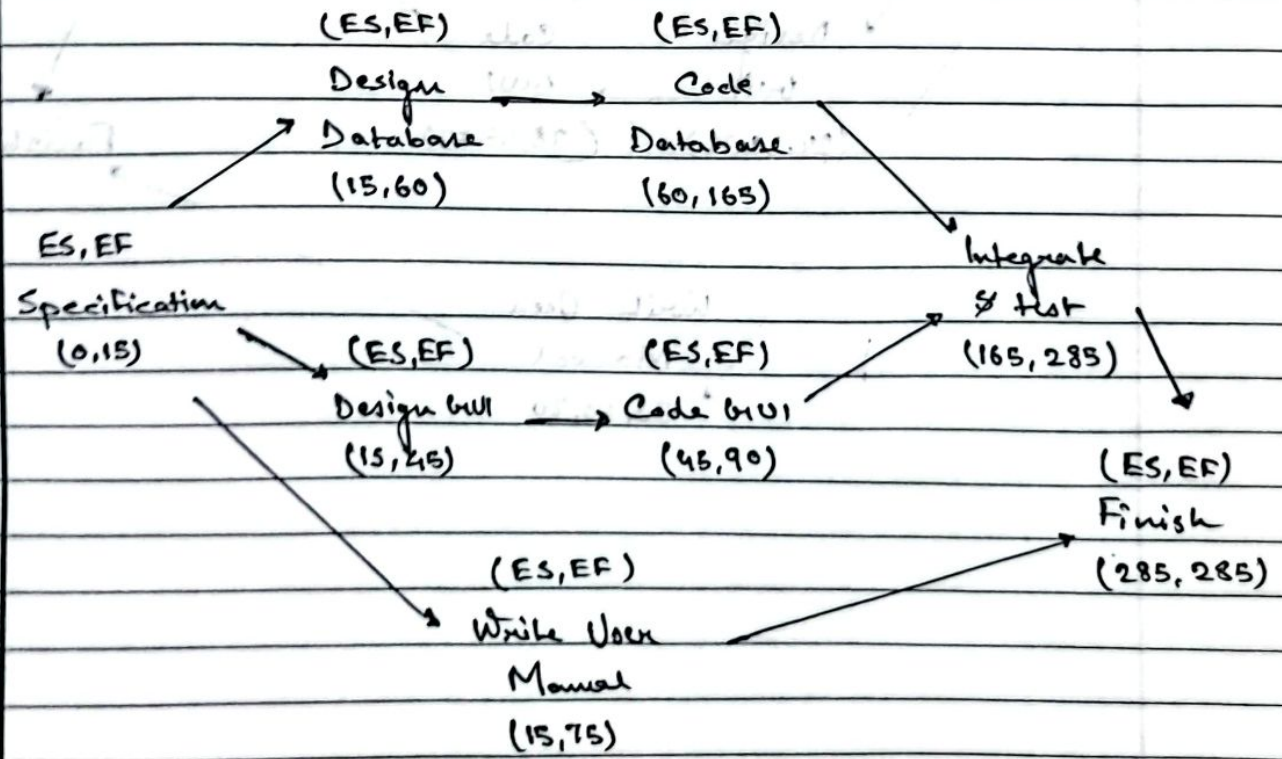
② Activity on Edge (AoE) - Task on edges
Nodes are project milestones



Page No:
Date:

Critical Path Method (CPM) It is an algorithmic approach to determine the critical paths and slack time for tasks not on the critical path involves calculating the following quantities:

- **Minimum time (MT)** - Min time to complete a project determined by the longest path in activity network.
- **Earliest Start Time (ES)** - The earliest the task can start, based on the completion of preceeding tasks
- **Latest start time (LS)** = MT - (max (path from this task to the end))
- **Earliest finish time (EF)** = ES + Duration of task
- **Latest finish time (LF)** = MT - Max. path duration from this task to the end
- **Slack time (ST)** = LS - ES or LF - EF



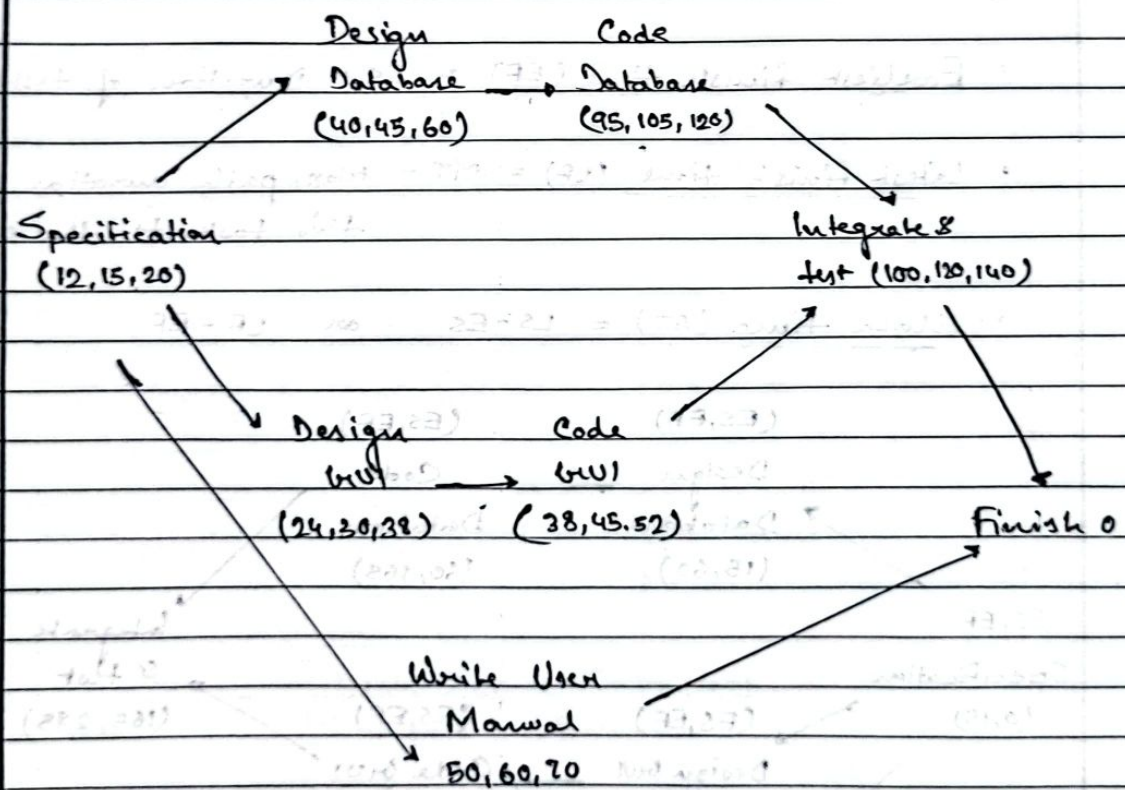
Page No:
Date:

PERT Chart - Project Evaluation and Review Technique
- consists of network of boxes and arrows

Boxes → Activities Arrow → Task dependencies

PERT chart represent statistical variations in the project estimates assuming normal distribution

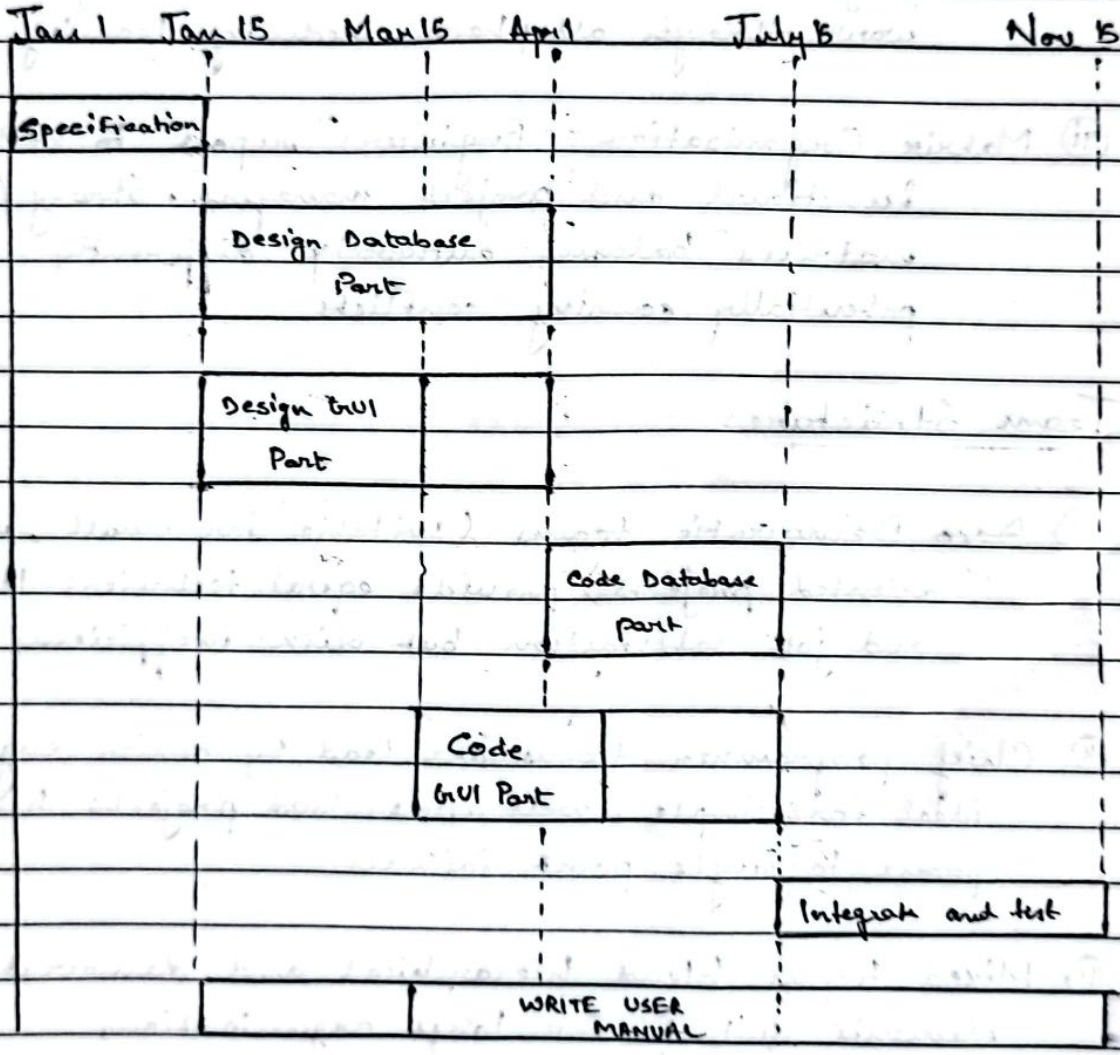
- Instead of making single estimate for each task,
  - (I) Pessimistic, worst case (W)
  - (II) Most likely estimates (M)
  - (III) Optimistic estimates (O) are made



Page No:
Date:

Gantt Chart

- Mainly used to allocate resources to activities
- The resources allocated to activities include staff, hardware and software, a special type of bar chart where each bar represents an activity.
- Bars are drawn along a time line. Each bar is an activity.
- Length of each bar is proportional to the duration of time planned for corresponding activity.



Page No:
Date:

Organisational Structures

(i) Functional Organisation - Engineers work in functional groups (Ex: Coding, testing) with projects borrowing engineers. Advantages include specialization ease of staffing and good documentation.

(ii) Project Organisation - Engineers remain with the project for its entire duration, allowing them to work through all phases, reducing learning time.

(iii) Matrix Organisation - Engineers report to both functional and project managers. Strong/weak matrices balance authority differently potentially causing conflicts

Team Structures

(i) ~~Dem~~ Democratic teams (suitable for small, research oriented projects) provide equal technical leadership and job satisfaction but risk inefficiency

(ii) Chief programmer teams are lead by senior engineer, ideal for simple, well understood projects but prone to single point failures

(iii) Mixed teams blend hierarchical and democratic elements suitable for large organisations

Page No:
Date:

Risk Management

① Project risks – Risks concern varies form of
- Budgetary
- Schedule
- Personnel
- Resource and
- Customer related problems

⑪ Technical risks – Risk concern
- Potential design
- Implementation
- Interfacing
- Testing and
- Maintainance problems

⑪ Business risks
- Building an excellent product that no body wants
- Losing budgetary or personnel commitments.

Risk Assessment

$$p = r * s$$

where p = priority
r = probability
s = severity of damage

Likelihood \ IMPACT	High	Low
High	High impact and likely to occur	Low impact and likely to occur
Low	High impact and unlikely to occur	Low impact and unlikely to occur

Page No:
Date:

Risk Containment

(i) Avoid the risk – discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to reduce the risk of manpower turnover etc.

(ii) Transfer the risk – getting the risky component developed by a third party, buying insurance cover etc.

(iii) Risk reduction – It involves planning ways to contain the damage due to a risk

Risk leverage

$$= \frac{\text{Risk exposure before reduction} - \text{Risk exposure after reduction}}{\text{Cost of reduction}}$$

Page No:
Date:

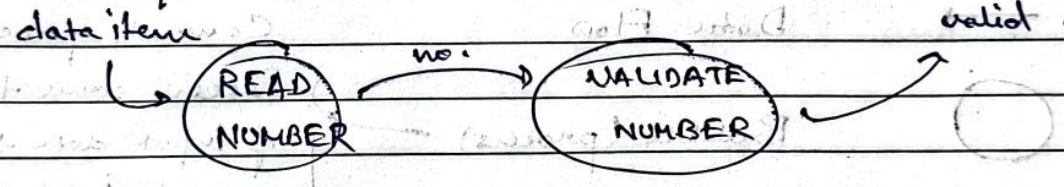
Structured Analysis & Design

The major processing tasks (functions) of the system are analysed and the data flows among these processing tasks are represented graphically.

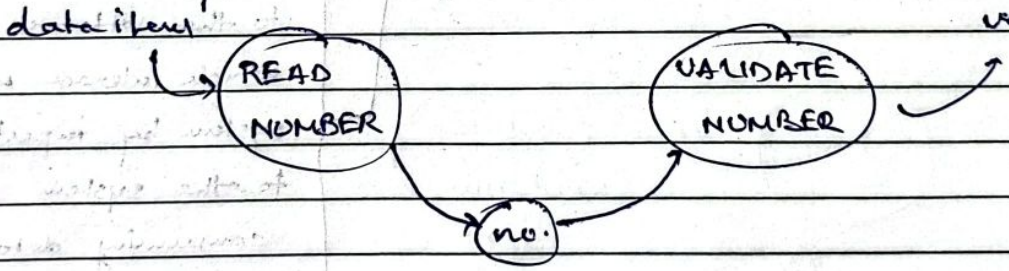
- Based on underlying principles:
  - (i) Top down decomposition approach.
  - (ii) Divide and Conquer principle: Each function is decomposed independently.
  - (iii) Graphical representation of the analysis i.e. creation of the DFD.

- Synchronous & Asynchronous Operation.

  * Synchronous



  * Asynchronous → With data store

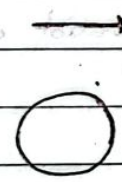


- Data Dictionary
  - Simple repositories to store information about all data items defined in DFD.
  - Every DFD must be accompanied by data dictionary
  - Includes: Name of data item, Aliases (other names), Description/purpose, related data items, Range of values, data flows, DS definition.

+ : Composition - of two data (a+b)
[,,] : Selection - [a,b] represents either a or b
() : Optional - a+(b) represents either a or a+b
{} : Iterative data definition
{name}2 : represents two data names
{name}* : represents zero or no names
=: Equivalence a =: b+c
/* : Comments :

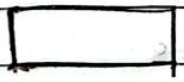
Data Flow Diagram

It shows the flow of data through the system.
- All names should be unique
- It is not a flow chart
- Supress logical decisions
- Defer error conditions and handling until the end of the analysis



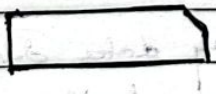
Data Flow — Connect process

Bubble (process) — Process some transformation of input data to yeild output data



External Entity — Physical entities external to the software system which interact with the system by inputting data to the system or by consuming data from the system

Data Store — A repository of data



Output — It is ~~produced~~ used when a hard copy is produced

Page No:
Date:

Context Diagram - Data Flow Diagram

- Most abstract (highest level) data flow representation of a system.
- Purpose of context diagram is to capture the context of the system rather than its functionality.
- It represents the entire system as a single bubble.
  The bubble in context diagram is annotated with the name of the software system being developed (usually a noun).
  This is only bubble in a DFD model where a noun is used for naming a bubble.
- The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble

Read three numbers in the range of -1000 to +1000 and determine the RMS of three numbers and display it.

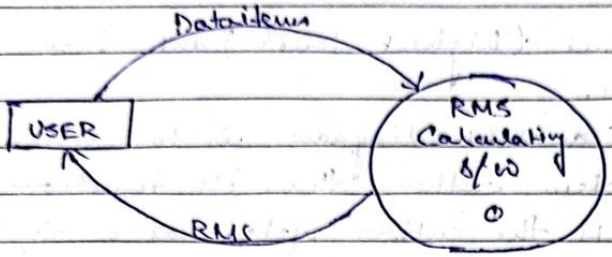
FUNCTIONAL REQUIREMENTS

I. Read three numbers
II. Validate the numbers in the range
III. Calculate RMS
IV. Display the result

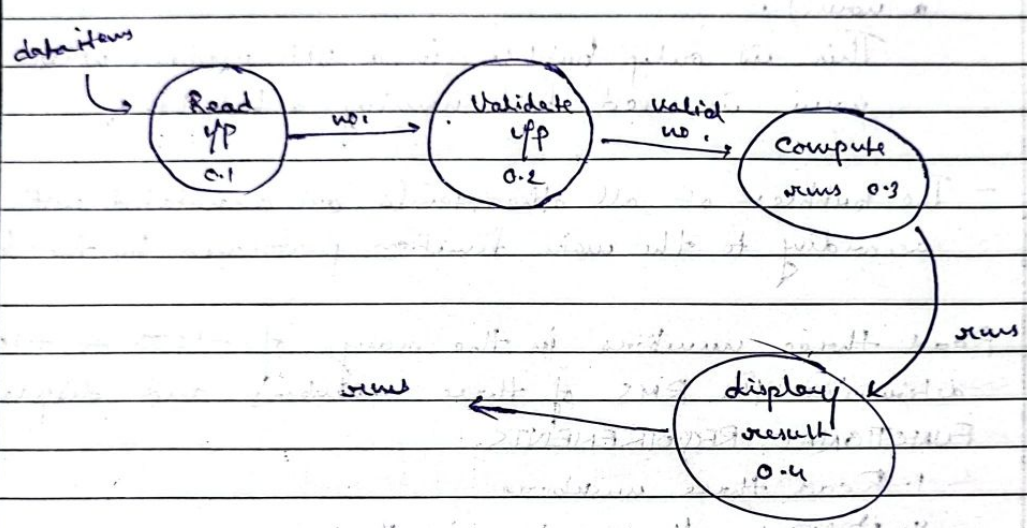
R1 : Accept input from user
R2 : Validate the numbers
R3 : Calculate the RMS
  R3.1 Calculate the square of three inputs
  R3.2 Calculate the mean
  R3.3 Calculate the root
R4 : Display results

Page No:
Date:

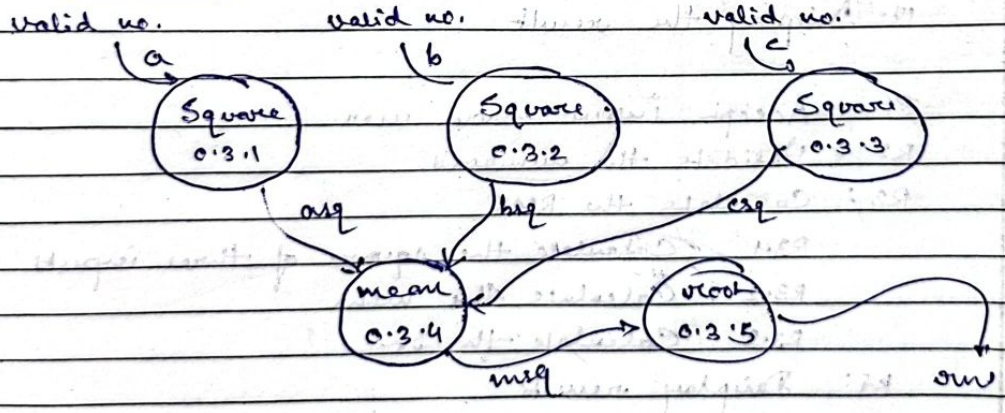
0 level DFD



1 level DFD

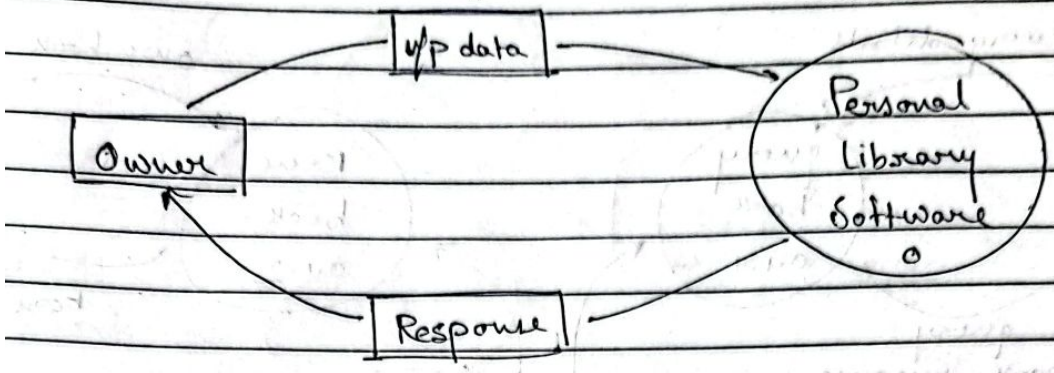


2 level DFD

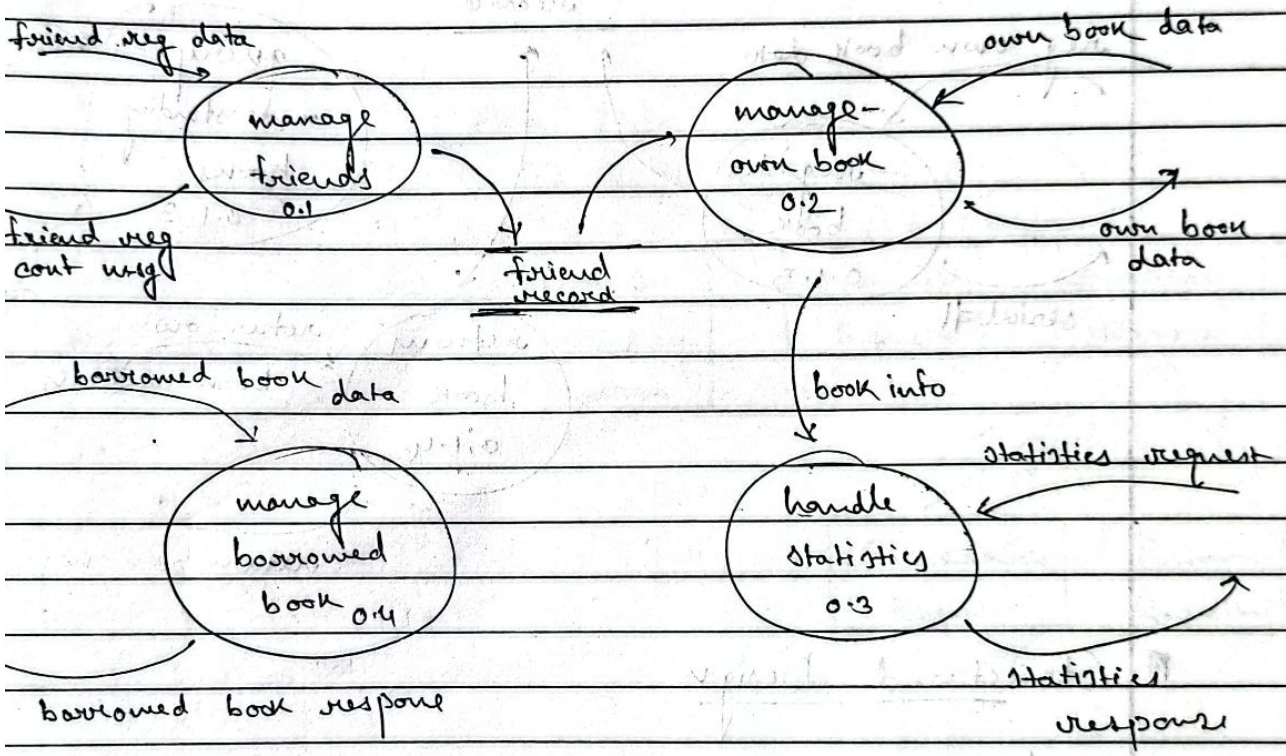


Page No:
Date:

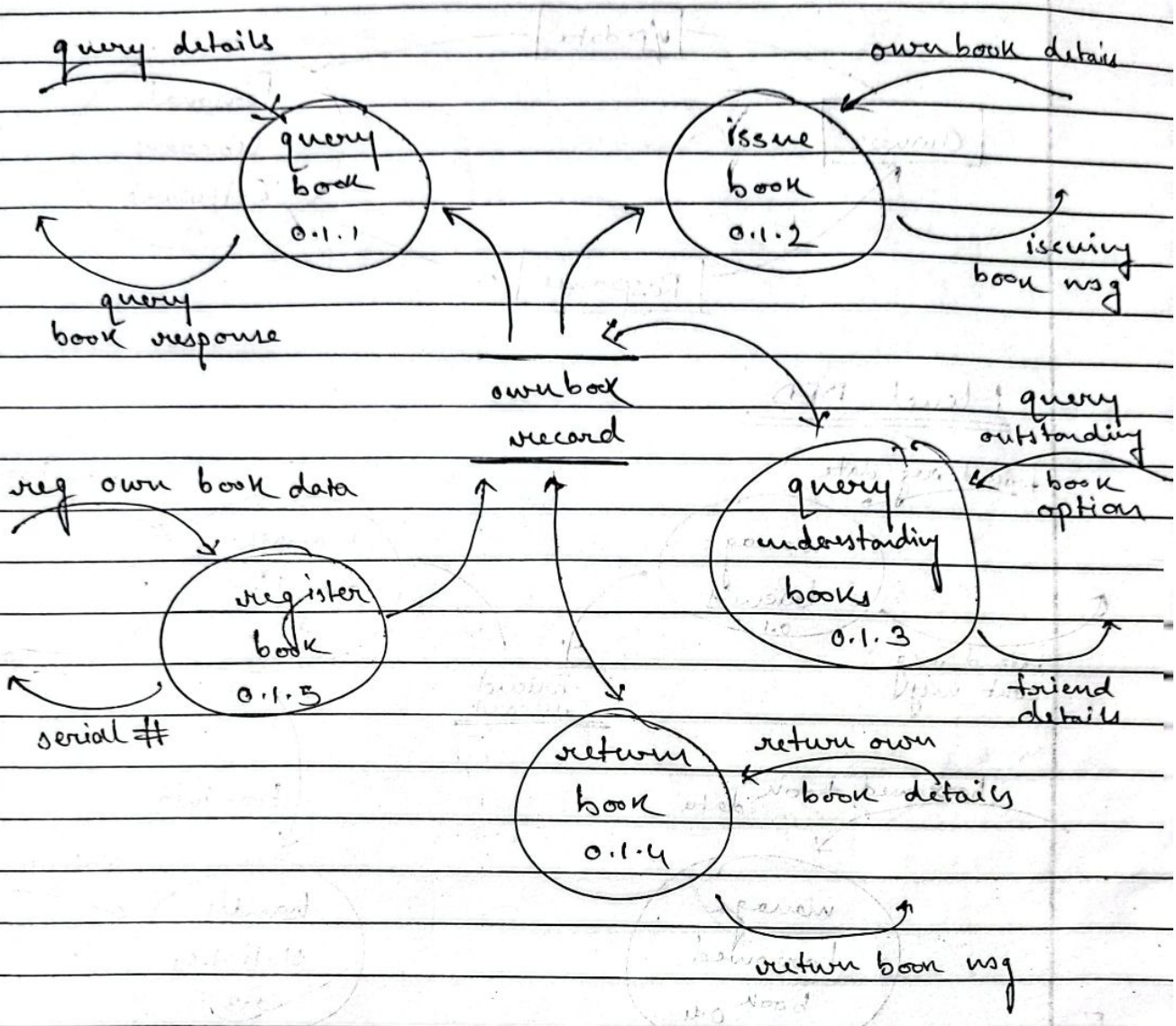
Personal Library Software (0 Level DFD)



1 level DFD



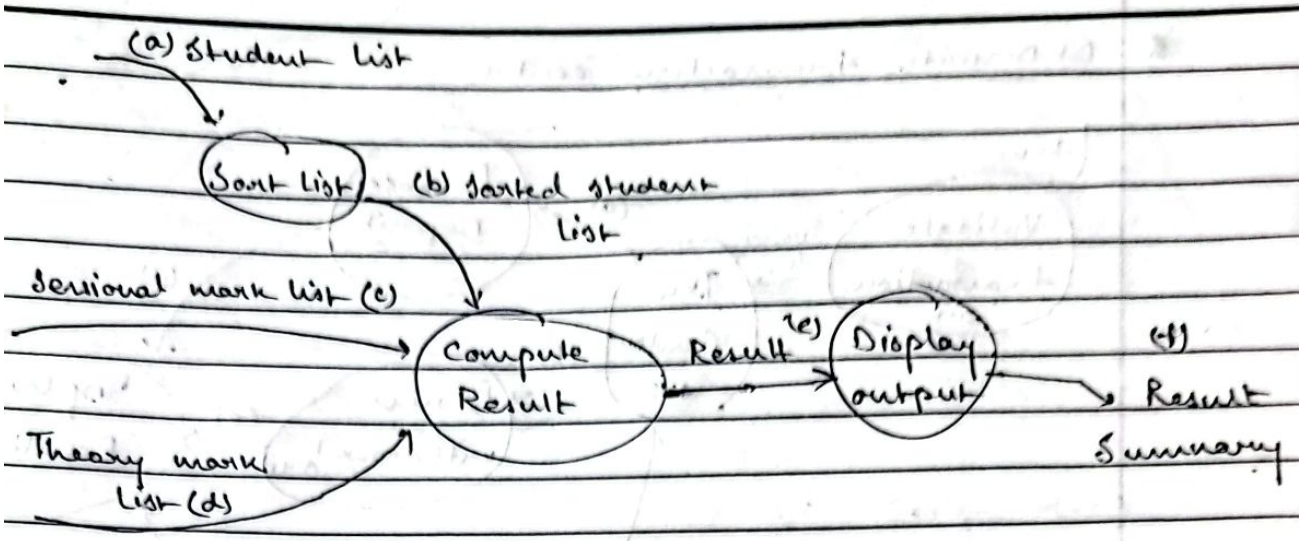
2 level DFD



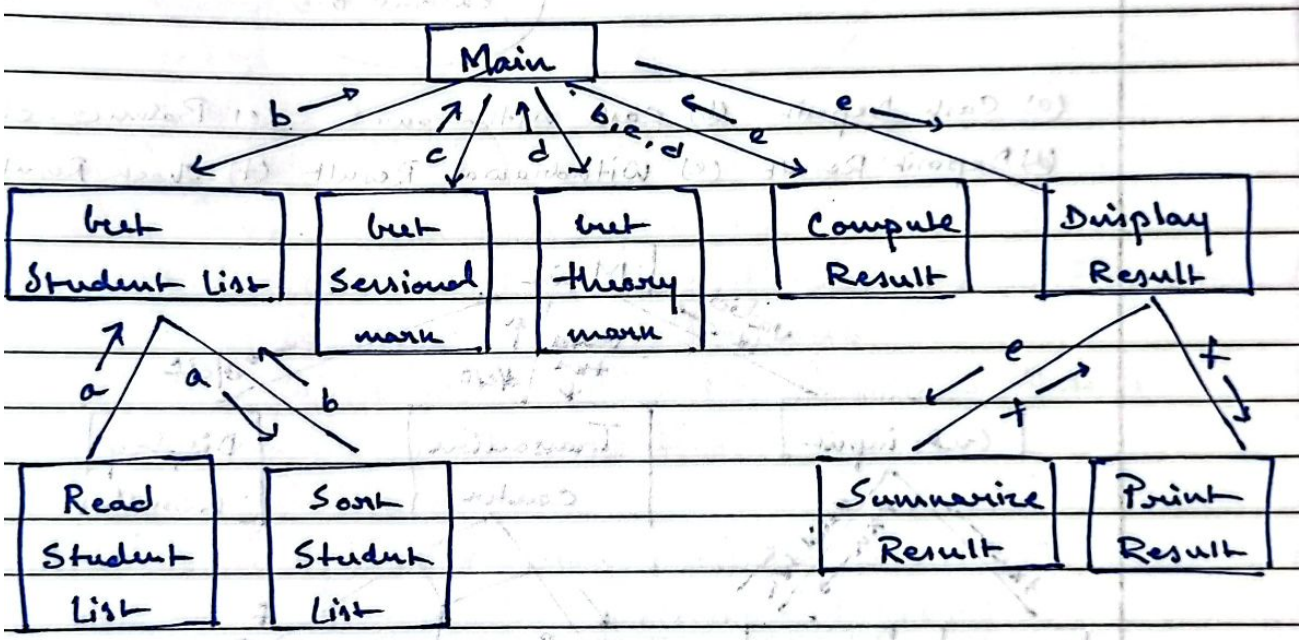
Structured Design

Aim is to transform DFD into a structure chart
- It represents the software architecture, which includes various modules making up the system, the model dependency and the parameters passed among different modules.

Structure charts are used to graphically model the hierarchy of processes within the system. Through the hierarchical format the sequences of process along with the movement of data and control parameters can be mapped for interpretation.

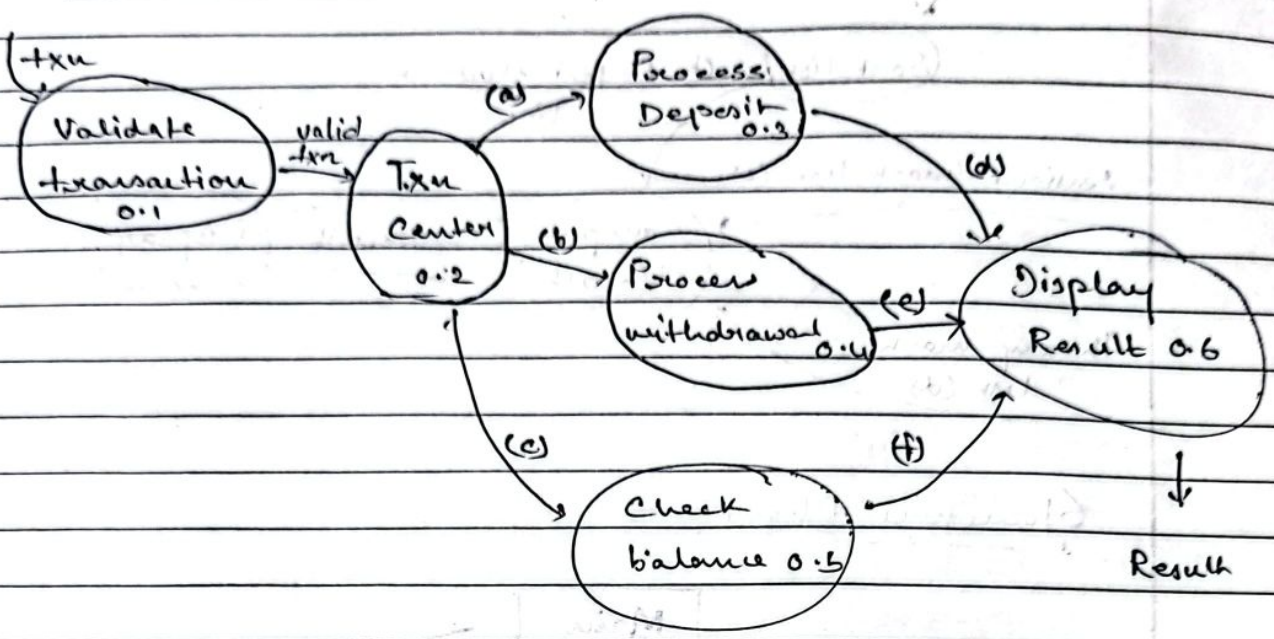


Structure Chart

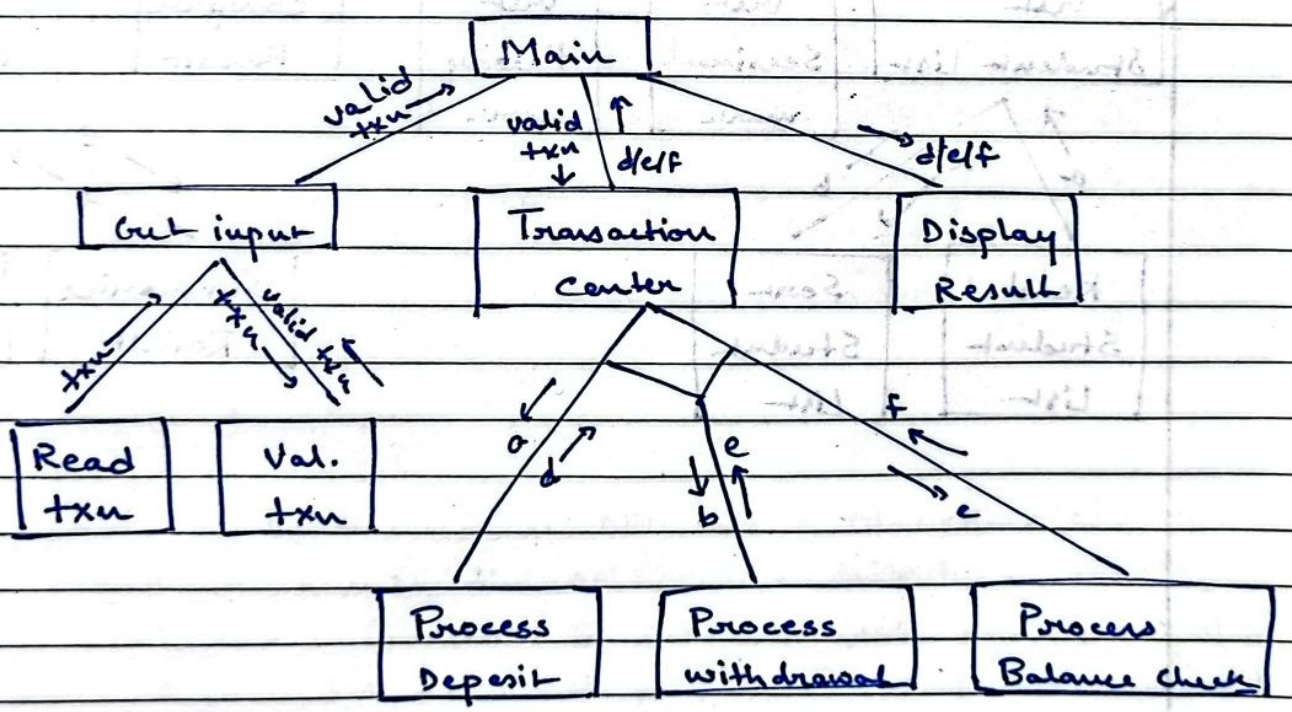


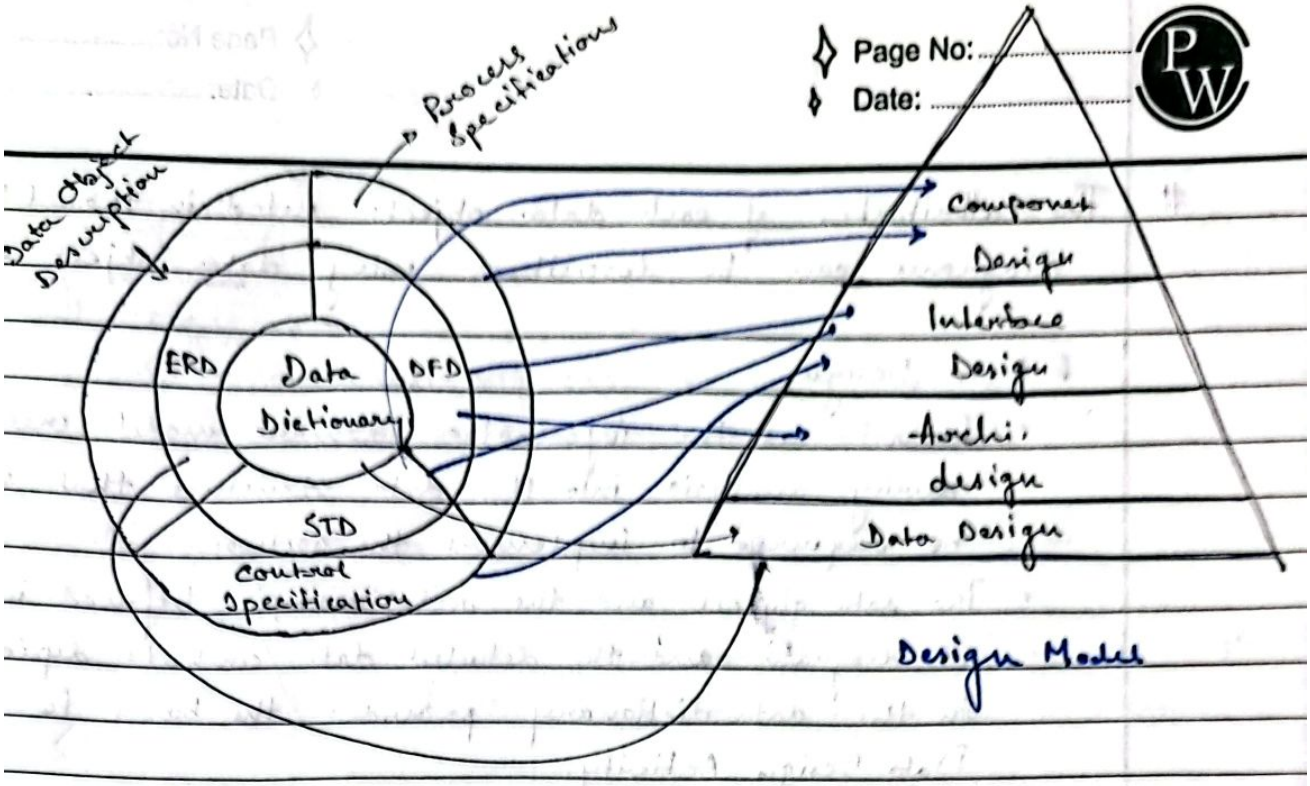
Page No:
Date:

DFD with transaction center



(a) Cash Deposit (b) Cash withdrawal (c) Balance-check
(d) Deposit Result (e) Withdrawal Result (f) Check Result





Analysis Model

Analysis Model / Design Model

DFD serves two purposes :

- Provides an indication of how data are transformed as they move through the system
- To depict the functions (& sub functions) that transform the data flow.

A description of each function presented in the DFD is contained in a processes specification (PSPEC)

STATE TRANSITION DIAGRAM (STD) indicates how system behaves as a consequence of external events. It represents various behaviour (state) of the system and the manner in which transitions are made from state to state.

Additional information about the control aspects of the software is contained in the control specifications (CSPEC)

- CSPEC can be:
  - (i) State transition diagram (sequential spec)
  - (ii) State transition table combinatorial spec
  - (iii) Decision tables activation tables.

Page No:
Date:

The attributes of each data objects noted in the ER diagram can be described using data object desc

Data Design

- Transforms the information domain model created during analysis into the data structure that will be required to implement the software
- The data objects and the relationships defined in the ER diagram and the detailed data contents depicted in the data dictionary provides the basis for the Data Design Activity.
- Part of the design occurs in conjugation with the design of the s/w architecture and more detailed data design occurs as each s/w component is designed.

Architectural Design

- Defines the relationships b/w major structural elements
- It can be derived from the system specification, the analysis model and interactions with the subsystems.

Interface Design

- It describes how the software communicates within itself, with the system that interoperate with it and with humans who use it.

Component level Design

- Transforms structural elements of the software arch. into procedural desc. of s/w components
- Information obtained from PSPEC, CSPEC and STD serves as a basis for component analysis.

Goal of

(i) Des

(ii) Desi

(iii) Desi

Critical

(i)
(ii)
(iii)
(iv)

Design

(1) At

(H) R

#

Page No:
Date:

Goal of design

(i) Design must
- Implement all explicit requirements
- Accomodate all implicit requirements

(ii) Design must be
- Readable
- Understandable guide for those who generate the code, who test and who support the software.

(iii) Design should provide a complete picture of the s/w addressing data, functions and behavioural domain from an implementation perspective.

Critical Success Factors

(i) Creative skill
(ii) Past experience
(iii) A sense of what makes "good" software
(iv) An overall commitment to quality.

Design Concept

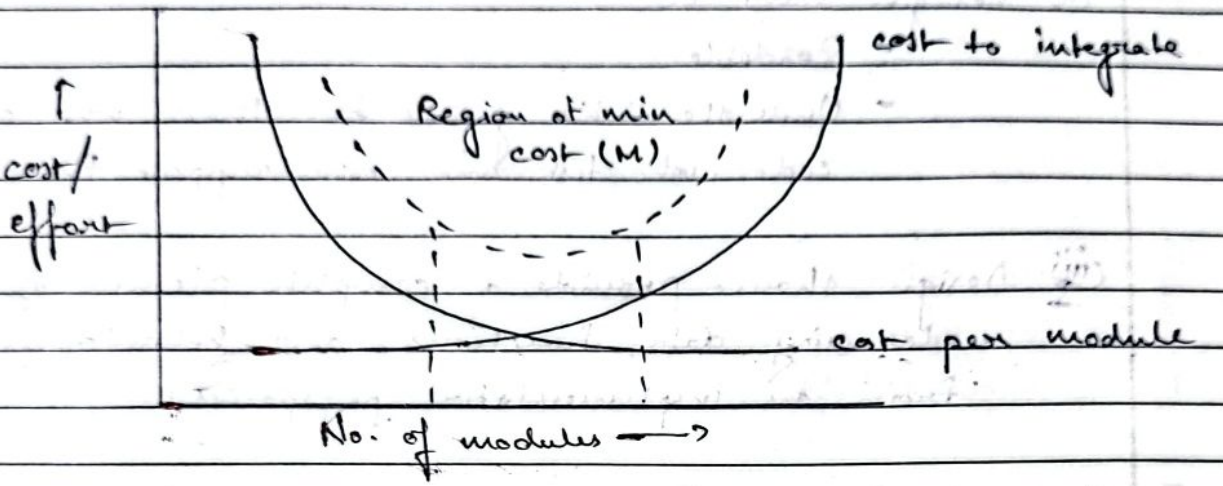
(I) Abstraction
- Procedural abstraction - It is a named sequence of instructions that has a specific and limited functions
- Data abstraction - Is a named collection of data describes a data object
- Control abstraction - It implies a program control mechanism without specifying internal details

(II) Refinement - It is process of elaboration

Abstraction & Refinement are complementary concepts

(iii) Modularity - Software architecture embodies modularity. The software is divided into separately named and addressable components (modules) that are integrated to satisfy problem requirements.

- Reduces complexity
- Facilitates changes



- Results in easier implementation by encouraging parallel development of different parts of the system.

(iv) Software architecture

It is the hierarchical structure of a program components (modules), the manner in which these components interact and the structure of data that are used by the component.

(v) Control hierarchy (program structure)

It represents the organisation of the program components (modules) and implies a hierarchy of control. It does not represent the procedural aspect of software.

(VI) Structural partitioning

For a hierarchical style of architecture the program module can be partitioned into:

(i) Horizontal
(ii) Vertical

- In horizontal partitioning, the control modules are used to communicate b/w functions and execute the function.

- In vertical partitioning, the functionality is distributed among the modules - in a top to down manner. The modules at top level perform the decision making and do little processing whereas the modules at the low level perform all input, computation and output tasks.

(VII) Data Structure

Represents the logical relationship among individual elements of data. It indicates the

(i) Organisation of data
(ii) Method of access
(iii) Degree of associativity
(iv) Process alternatives for information

(VIII) Software procedure

Program structure defines the control hierarchy, represents the organisation of program components (modules) and implies a hierarchy of control.

- Software focuses on the processing details of each module individually.
- Procedure must provide a precise specification of processing including:
  - Sequence of events
  - Exact decision points

(ix) Information Hiding - Modules can be characterized by design decision that each module hides from all others. i.e. information contained within a module is inaccessible to other modules that have no need for such information.

Cohesion

Cohesion is the measure of
- functional strength of a module
- A cohesive module performs a single task/func.

Coupling

Coupling b/w two modules is a degree of measure of interdependence or interaction between two modules

#A module having high cohesion and low coupling is a good design.

Advantages of functional interdependence

(i) Better understandability and good design
(ii) Complexity of design is reduced
(iii) Different modules easily understood in ~~iso~~ isolation since different modules are independent
(iv) Functional independence reduces error propagation
- Degree of interaction b/w modules is low
- An error existing in one module does not directly affect other modules
(v) Reuse of modules is possible

Degree of cohesion and coupling

There are no ways to quantitatively measure the degree of cohesion and coupling.

Page No:
Date:

Classification of cohesiveness

functional
sequential
communicational
procedural
temporal
logical
coincidental

↑ high
Degree of Cohesion
low ↓

* Coincidental Cohesion

Module performs a set of tasks which relate to each other very loosely, if at all. Data elements or instructions within module have no relation to each other.

* Logical Cohesion

All elements of the module perform similar operations. Components are logically related to each other. Modules which contains routines to support different types of input.

* Temporal Cohesion

The module contains tasks that are related by the fact that all the tasks must be executed in the same time span.

* Procedural Cohesion

A set of functions of the module are all the part of a procedure (algorithm)

* Communicational Cohesion

All functions of the module reference or update the same data structure. Different outputs are generated on the same input.

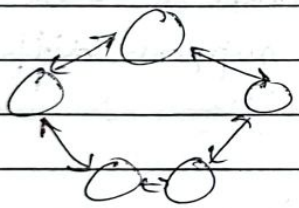
Page No:
Date:

* Sequential Cohesion
Elements of a module form different parts of a sequence i.e. the output from one element of the sequence is input to the next

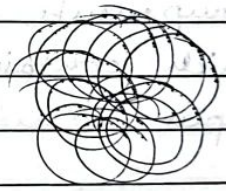
* Functional Cohesion
Different elements of a module cooperate to achieve a single function

Coupling indicates -
(i) How closely two modules interact ?
OR
(ii) How interdependent they are ?

- The degree of coupling b/w two modules depends on their interface complexity

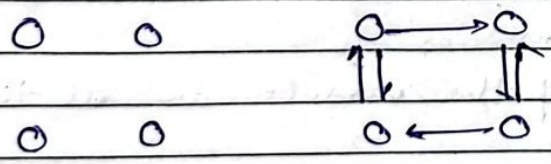


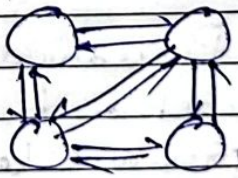
Loose coupling



~~this~~ tight coupling

- Module coupling





uncoupled	loosely coupled	highly coupled
(No dependencies)	(Some dependencies)	(Many dependencies)

Page No:
Date:

Classifications of Coupling

low

Data
Stamp
Control
Common
Content

Degree of cohesion

high

* Data coupling - Two modules are data coupled if they communicate via an elementary data item.
  Ex - An integer, a float, a character...
  The data item should be problem related, but not used for control purpose.

* Stamp Coupling - Two modules are stamp coupled if they communicate via composite data item.

* Control Coupling - Data from one module is used to direct the order of instruction execution in another.

* Common Coupling - Two modules are common coupled, if they share some global data

* Content Coupling - Content coupling exists between two modules share code.
  - Content coupling exists b/w two modules, when a module can directly access or modify or refer to the content of another module.

Page No:
Date:

Unified Modelling Language (UML)

The unified modelling language (UML) is a general purpose, developmental modelling language in the field of software engineering that is intented to ~~form~~ provide a standard way to visualize the design of the system.

- UML is a modelling language
- Used to document object oriented analysis and design results
- Independent of any specific design methodology

UML Model Views

① Users View

- The users view captures the functionalities offered by the system to its users.
- It is a black box view of the system where the internal structure, the dynamic behaviour of different system components, the implementation etc. are ignored

• Usecase Diagram

- Describes what a system does from the standpoint of an external observer.
- Emphasis on what a system does rather than how
- Scenario - An example of what happens when someone interacts with the system.
- Actor - A user or another system that interacts with the modelled system.

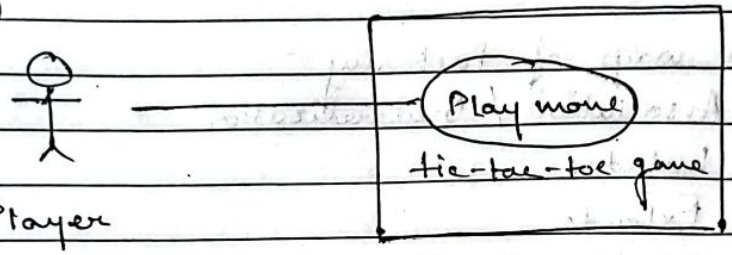
A usecase diagram describes the relationships b/w actors and scenarios
Provides system requirements from the user point of view.

Page No:
Date:

Representations of use cases

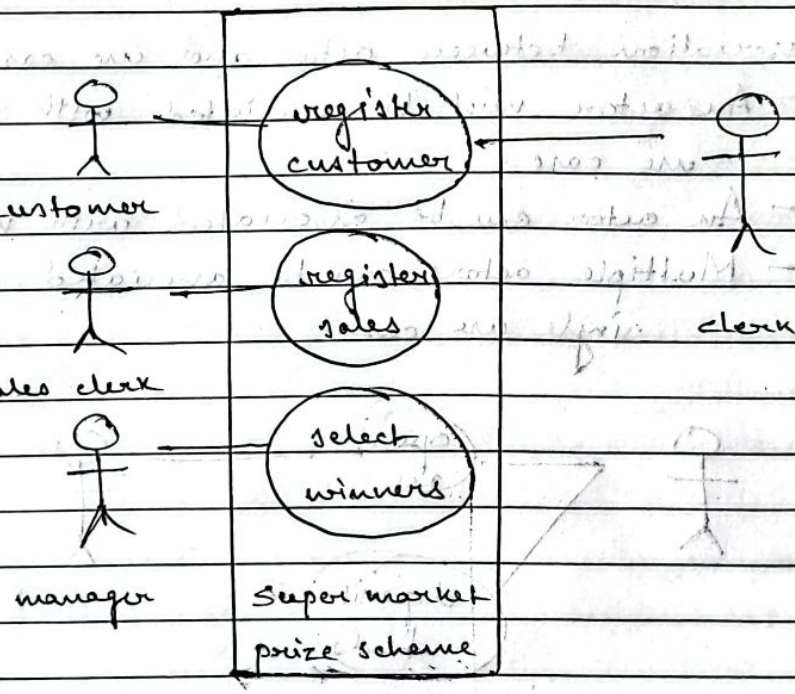
(i) A use case is represented by an ellipse
(ii) System boundary is represented by a rectangle.
(iii) Users are represented by stick person icons (actor)
(iv) Communication relationship b/w actor and use case by a line
(v) External system by a stereotype

Ex 1)



Use Case Model

Ex 2)



Use-cases

(i) Different ways in which a system can be used by the users
(ii) Corresponds to the high level requirements
(iii) Represents transaction b/w the user and the system
(iv) Defines external behaviour without revealing internal structure of the system
(v) Set of related scenarios tied together by a common goal

Page No:
Date:

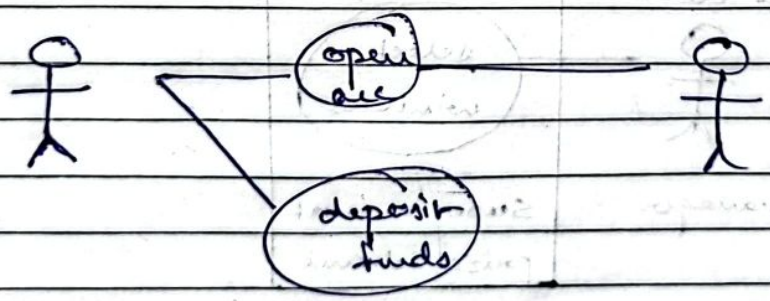
Factoring use cases

- Two main reasons for factoring –
  (i) Complex use cases need to be factored into simpler use cases.
  (ii) To represent common behaviour across different use cases

- Three ways of factoring –
  (i) Association / Generalization
  (ii) Includes
  (iii) Extends

Use case diagram relationships

(i) Association between actor and use case
- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases
- Multiple actors can be associated with a single use case.



(ii) Generalization of an actor
- It means that one actor can inherit the role of the other actor
- The descendent inherits all the use cases of the ancestor.
- The descendant has one or more use cases that are specific to that role

Page No:
Date:

(iii) <<Extend>> between two use cases
- The extending use case is dependent on the extended (base) use case
- The extending use case is usually optional and can be triggered conditionally.
- The extended (base) use case must be meaningful on its own

(iv) <<include>> between two use cases
- Include relationship show that the behaviour of the included use case is part of the including (base) use case
- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

(v) Generalization of a use case
- This is similar to the generalization of an actor.
- The behaviour of the ancestor is inherited by the descendant
- This is used when there is a common behaviour b/w two use cases and also specialized behaviour specific to each use case.

Class Diagram

A class diagram in UML is a type of static structure diagram that describes the structure of a system by showing the system's
- class
- their attributes
- operations (or methods)
- relationships among objects

Class diagrams can directly mapped with object oriented languages and thus widely used at the time of construction

Class diagrams are static - display what interacts but not what happens when interaction occurs

- A class represents a concept which encapsulates state (attributes) and behaviour (operations).
  - Each attribute has a type
  - Each operation has a signature
- The class name is the only mandatory information

Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

+ : public attributes / operations
- : private attributes / "
: protected " / "

Parameter Directionality

Each parameter in an operation (method) may be denoted as in, out or inout which specifies its direction wrt caller. This dimensionality is shown before the parameter name.

Object Diagram

- Describe the static structure of a system at a particular time.
- Whereas a class model describes all posible situations, an object model describes a particular situation.
- Object diagram contain the following elements -
  - Objects - represent particular entities. These are instances of a class
  - Links - Represent particular relationships b/w objects. These are instances of associations

Page No:
Date:

Interaction Diagram

- Models on how groups of objects collaborate to realize some behaviour
- Typically each interaction diagram realizes behaviour of a single use case.

• Two types of interaction diagram —

① Sequence diagram – Shows interaction among objects as a two dimensional chart. The chart is read from top to bottom.

- Objects are shown as boxes at top.
- If object created during execution then shown at appropriate place.
- Objects existence are shown as dashed lines (lifeline)
- Objects activeness shown as rectangle on lifeline.

- Messages are shown as arrows
- Each message labelled with corresponding message name.
- Each message can be labelled with some control information.
  - condition ([ ])
  - iteration (*)

② Collaboration Diagram – Shows both structural and behavioural aspects. Objects are collaborator, shown as boxes. Message b/w objects shown as a solid line. A message is shown as a labelled arrow placed near the link. Messages are prefixed with sequence number

Page No:
Date:

Activity Diagram

It describes the workflow behaviours of a system.

- A fork notation in a UML Activity Diagram is a control node that splits a flow into multiple concurrent flows.

- A branch specifies alternate paths taken based on some Boolean expression which is also known as guard condition.

- A merge node is a node in an activity at which several flows are merged into single flow.

- A join node joins multiple concurrent flows back into a single outgoing flow

State Chart Diagram

It is used to model the dynamic nature of system. It defines the different states of an object during its lifetime and these states are changed by events

- It describes the flow of control from one state to another state.

- States are defined as a condition in which an object exists and it changes when some event is triggered.

- The most important purpose of state chart diagram is to model lifetime of an object from creation to termination.

Page No:
Date:

Package diagram

Package diagram, a kind of structural diagram, shows the arrangement and organisation of model elements in middle to large scale project.

Package diagram can show both structure and dependencies b/w sub systems and modules.

Package diagram can be used to simplify complex class diagrams, it can group classes into packages. A package is a collection of logically related UML elements. Packages are depicted as file folders and can be used on any of the UML diagrams.

Component diagram

- Captures the physical structure of the implementation (code components)
  - Executables, library, table, file, documen-

- Captures the physical structure of the implementation
- Built as part of architectural specification
- Purpose -
  (i) Organise source code
  (ii) Construct an executable release
  (iii) Specify a physical database
- Developed by architects and programmers

Page No:
Date:

TESTING STRATEGIES

- **Testing :**

Software testing is the process of accessing the functionality of a software program.

Software testing is the process of finding errors in the developed product.

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

- MISTAKE - An action performed by an individual leading to incorrect result.

- FAULT - It is the outcome of mistake. It can be a wrong step, wrong definition etc. It may or may not result into failure.

- FAILURE - It is the outcome of fault

- ERROR - Amount of deviation from from correct result

- Positive Testing - It determines that your application works as expected. If an error is encountered during positive testing, the test fails.

Positive testing involves validating an applications functionality with valid i/p's to ensure that it performs as expected.

Page No:
Date:

Negative testing - It ensures that your application can gracefully handle input or unexpected user behaviour. Negative testing explores the systems behaviour when it is subjected to invalid inputs or unexpected conditions

CODING PHASE

Benefits of coding standard

- Gives a uniform appearance of the code written by different software engineers.
- Provide sound understanding of the code.
- Encourage good programming practice

Representing coding guidelines

- Rule for limiting the use of global (else high coupling results)
- Content of header preceeding code for different modules.
- Naming convention for Global, local variables and CONSTANTS.
- Error handling convention and exception handling mechanisms
- Don't use coding style that is too difficult to understand and too clever.
- Avoid obscure side effects (passing reference variables or updating global variables)

- Don't use identifier for multiple purpose
- Code should be well documented
- Length of any function should not exceed 10 lines
- Do not use GO TO statements

Code Review

Code review is systematic examination of computer source code.

Prepared after the module successfully compiled and all syntax errors are eliminated.

Intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and developer skills. Used for reducing coding error.

- Code Walk through

After the module is successfully compiled a few members of the development team are given the code to read and understand it. Each member selects some test cases and simulate execution by hand. The main objective is to discover the algorithmic and logical errors in the code. The member note down their finding to be discussed in walk through meeting

- The team performing code walk through meeting should not be too big nor too small (b/w 3 to 7)
- Discussion should focus on discovering errors not on how to fix the discovered errors
- Managers should not attend code walk through meetings

Benefits

- Earlier bugs found in life cycle product
- Other developers are likely to find mistakes missed.

- Code inspection

Classical programming errors
- Use of uninitialized errors
- Jumps into loops
- Non terminating loops
- Incompatible assignments
- Array indices out of bound
- Improper storage allocation & de-allocation
- Mismatches b/w actual & formal parameters
- Use of incorrect logical operations and incorrect precedence among operators
- Improper modifications of loop variables
- Comparision of quality of floating point values etc.

- Software Documentation

- Purpose of good documentation
  - Enhance understandability & maintainability of software product
  - Helps the user in effectively exploiting system
  - Effectively overcoming the manpower turn over problem
  - Helps managers in effectively tracking the progress of the project

Software testing - Verification & Validation

* Verification (process oriented)
- Is the process of determining whether the output of one phase of software development confirms to that of its previous phase
- It is concerned with phase containment of errors
- Requires white box testing strategies
  - Component testing - Unit, module
  - Integrated testing - Subsystem, System

Page No:
Date:

* Validation (product oriented)
- Is the process of determining whether the fully developed system confirms to its requirement specifications.
- Aim is to ensure that the final product is error free.
- Requires black box testing techniques
  - User testing - Acceptance

Black Box Testing

Test cases are designed from an examination of the I/O values and no knowledge of design or code is required
It treats the system as a black box.
Also called as:
- Functional testing
- Specification testing
- Behavioural testing
- Data driven testing
- I/O driven testing

- Equivalence Class Partitioning
Domain of I/P value to program is partioned into a set of equivalence classes and its done in such a way that the behaviour of the program is similar to every input data belonging to the same equivalence class.

- Validating an applications functionality with valid inputs to ensure that it performs as expected is called positive testing

- Explores the systems behaviour when it is subjected to invalid inputs or unexpected conditions is called negative testing.

Page No:
Date:

(Ex) Linear functions $y = mx + c$

Design the equivalence class test cases for program that reads two integer pairs $(m1, c1)$ and $(m2, c2)$ defining two straight lines of the form $y = mx + c$. The program computes the intersection points of two straight lines and display the point of intersection.

The equivalence classes are the following:

(i) Parallel lines: $(m_1 = m_2, c_1 \neq c_2)$

(ii) Intersecting lines: $(m_1 \neq m_2)$

(iii) Coincident lines: $(m_1 = m_2, c_1 = c_2)$

(Ex) College Admission Process

A college gives admissions to students based upon their percentage. Consider % field that will accept % only between 50% to 90%, more and even less than not be accepted and application will redirect user to error page.

If % entered by the user is less than 50% or more than 90%, that equivalence partitioning method will show as invalid %

If % entered is b/w 50% to 90%, then the equivalence partitioning method will show valid %.

Equivalence partitioning		
Invalid	valid	Invalid
<= 50	50 - 90	>= 90

• Boundary value analysis (BVA)

It is based on testing the boundary values of valid and invalid portions. The behaviour at the edge of each equivalence partition is more likely to be incorrect than the behaviour within the partition, so boundaries are an area where testing is likely to yeild defects

• Advantages

- It is easier and faster to find defects using this technique. This is because the density of defects at boundaries is more.
- Instead of testing with all set of test data, we only pick the one at the boundaries. So the overall test execution time reduces.

• Disadvantages

- The success of the testing using this technique depends on the equivalence class identified which further depends on the expertise of the tester and his knowledge of the application. Hence incorrect identification of equivalence classes leads to incorrect boundary value testing.

White Box Testing

White box testing techniques analyze the internal structure the used data structures, internal design, code structure and the working of the software rather than just the Functionality as in black box testing. It is also known as glass box testing clear box testing or structural testing.

White box testing also known as transparent testing or open box testing

Page No:
Date:

What does white box testing focus on?

(i) Path checking - It examines the different routes the program can take when it runs. Ensures that all decisions made by the program are correct, necessary and efficient.

(ii) Output validation - Tests different i/p's to see if the function gives the right output each time.

(iii) Security testing - Uses techniques like static code analysis to find and fix potential security issues in the software. Ensures the s/w is developed during secure practices

- Types of white box testing

(i) Unit testing - Checks if each part or function of the application works correctly. It ensures the application meets design requirements during development.

(ii) Integration testing - Examines how different parts of the application work together. Done after unit testing to make sure components work well both alone and together.

(iii) Regression testing - Verifies that changes or updates don't break existing functionality. It ensures the application still passes all existing tests after updates.

Page No:
Date:

- White box testing techniques

① Statement Coverage

In this technique, the aim is to traverse all statements at least once. Hence each line of code is tested. In the case of a flowchart, every node must be traversed at least once. Since all lines of codes are covered, it helps in pointing out faulty code.

② Branch coverage

In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.

③ Condition Coverage

In this technique, all individual conditions must be covered as in the following example code

```
READ X,Y
IF (X==0 || Y==0)
PRINT '0'
```

TC1 - X = 0, Y = 55

TC2 - X = 5, Y = 0

④ Multiple condition coverage

In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once

```
READ X,Y
IF (X==0 || Y==0)
PRINT '0'
```

TC1 - X = 0, Y = 0

TC2 - X = 0, Y = 5

TC3 - X = 55, Y = 0

TC4 - X = 55, Y = 5

⑤ Basis path testing

Control flow graphs are made from code / flowchart and then cyclomatic complexity is calculated which defines the no. of independent paths so that the minimal number of test cases can be designed for each independent path.

Steps:

(i) Make the corresponding control flow graph.
(ii) Calculate the cyclomatic complexity
(iii) Find the independent paths
(iv) Design test cases corresponding to each independent path.

V(G) = P + 1 where P is the no. of predicate nodes in the flow graph

V(G) = E - N + 2 where E is the no. of edges and N is the total no. of nodes

V(G) = No. of overlapping regions in the graph.

#P1 : 1 - 2 - 4 - 7 - 8
#P2 : 1 - 2 - 3 - 5 - 7 - 8
#P3 : 1 - 2 - 3 - 6 - 7 - 8
#P4 : 1 - 2 - 4 - 7 - 1 - .... - 7 - 8

⑥ Loop testing

Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginings and end of loops.

- Simple loops : For simple loops of size n, test cases are designed that

Page No:
Date:

1. Skip the loop entirely
2. Only one pass through the loop
3. 2 passes
4. m passes, where $m < n$
5. $n-1$ and $n+1$ passes

- Nested loops : For nested loops, all the loops are set to their minimum count, and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

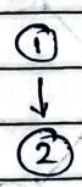
- Concatenated loops : Independent loops, one after another. Simple loop tests are applied for each. If they are not independent, treat them like nesting.

How to draw control flow graph ?

1. Number all the statements of a program
2. Numbered statements
   - Represent nodes of the CFG
3. An edge from one node to another node exists:
   - If execution of the statement, representing the first node, can result in transfer of control of the other node

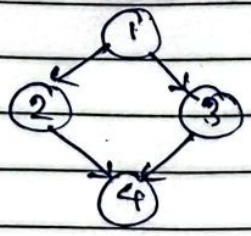
Sequence

```
1   a = 5;
2   b = a * b - 1;
```



Selection

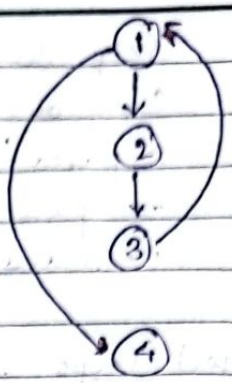
```
1   if (a > b) then
2       c = 3;
3   else c = 5;
4   c = c * c;
```



Page No:
Date:

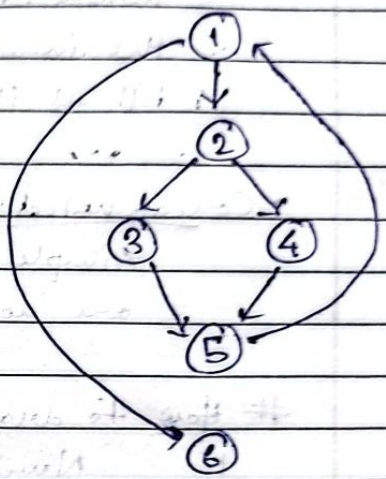
Iteration

```
1   while (a > b) {
2       b = b * a;
3       b = b - 1; }
4   c = b + d;
```

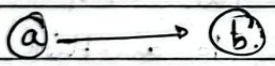


Ex

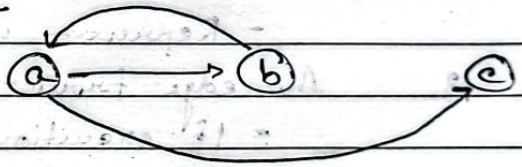
```
int f1(int x, int y) {
    while (x != y) {
        if (x > y) then
            x = x - y;
        else y = y - x;
    }
    return x; }
```



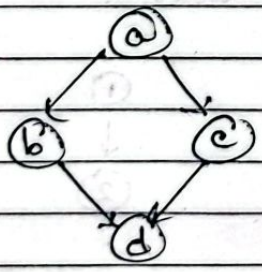
Notion of drawing CFG



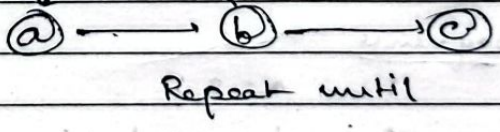
Sequence



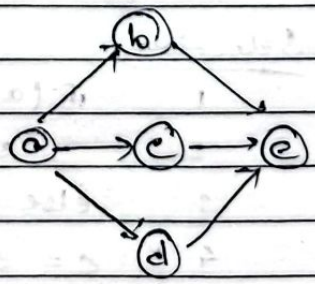
while loop



if then else



Repeat until



case

Page No:
Date:

McCabes Cyclomatic Complexity

Cyclomatic complexity is a software metric used to indicate the complexity of the program.
It is a quantitative measure of the no. of linearly independent paths through a programs source code.
Cyclomatic complexity is calculated using the control flow ~~diagram~~ graph of the program:
It may also be used for individual functions, modules, methods, classes within a program.

- The cyclomatic complexity can be calculated using any one of the following

$V(G)$ = No. of simple decisions + 1
$V(G)$ = E - N + 2
$V(G)$ = No. of enclosed areas + 1

Steps to carry out path coverage based testing

1. Draw the CFG
2. Determine the cyclomatic complexity (gives min no. of test cases required for path coverage)
3. Repeat
   Test using a randomly designed set of test cases.

Other use of cyclomatic complexity

* Estimation of structural complexity
* Estimation of testing effort
* Estimation of program reliability

Page No:
Date:

Unit tests help to fix bugs early in the development cycle and save costs.

It helps the developers to understand the testing code base and enables them to make changes quickly.

Good unit tests serve as project documentation.

Unit tests help with code reuse. Migrate both your code and your tests to your new project.

Stubs and drivers are the dummy programs in the integration testing used to facilitate the software testing activity.
These programs act as a substitutes for the missing models in the testing.
They do not implement the entire programming logic of the software module but they stimulate data communication with the calling module while testing.

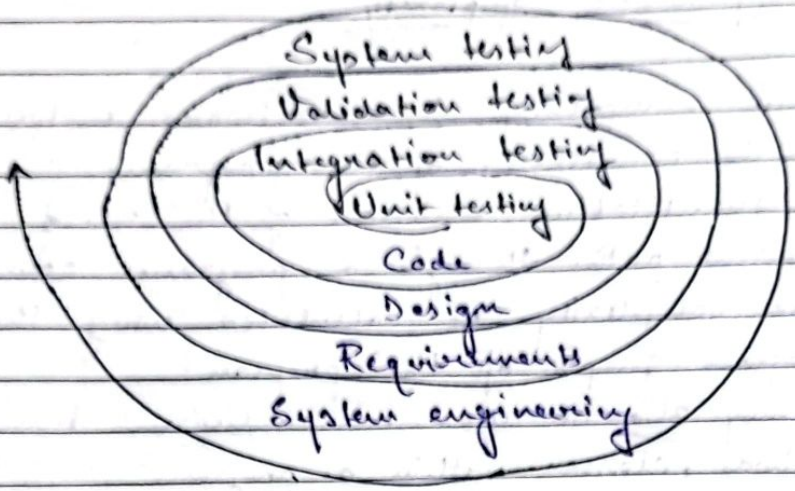
Stub: Is called by the module under test
Driver: Calls the module to be tested.

Integration Testing

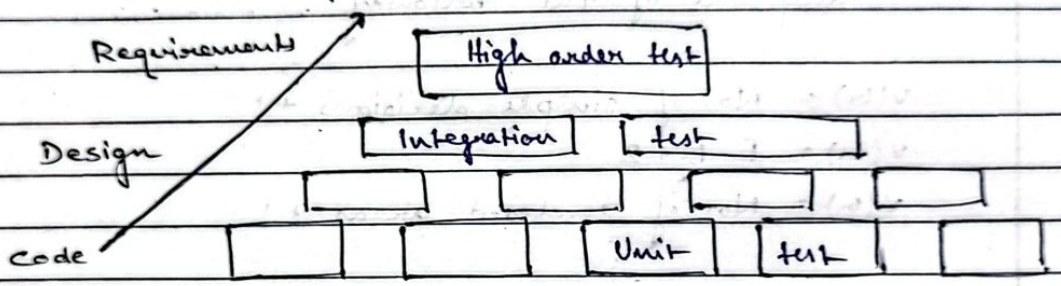
Integration testing is the process of testing the connectivity or data transfer between the couple of unit tested modules.

The primary objective of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameters passing, when one module invokes another module.

In integration testing, different modules making up the system are integrated in a planned manner using an integration plan.



- Software testing steps



- Unit testing

It focusses on the verification effort on the smallest unit of software design i.e. the software component or modules.

It focusses on the internal processing logic and data structure within the boundaries of a component

Unit testing is simplified when high cohesion is designed.

Unit testing is a white box testing technique that is usually performed by the developer.

Two strategies:

i) Phased integration testing
ii) Incremental integration testing

A group of related modules are added, each time, to the partial system under test.

Only one new module is added, each time, to the partial system under test.

- **BIG BANG APPROACH**
  - Least used and least effective approach
  - All modules / components are combined together at a time and then tested in one go.

Disadvantages

- Defects present at the interfaces of components are identified at very late stage.
- Very difficult to isolate the defects found.
- High probability of missing some critical defects.

- **TOP-DOWN INTEGRATION**

Top down integration testing is a method in which integration testing takes place from top to bottom following the control flow of software system.

The higher level modules are tested first and then lower level modules are tested and integrated in order to check the software functionality.

Stubs are used for testing if some modules are not ready.

Page No:
Date:

Advantages
- Fault localization is easier
- Possibility to obtain an early prototype
- Critical modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages
- Needs many stubs
- Modules at a lower level are tested inadequately.
- Functionality of the complete system can be demonstrated early.

• BOTTOM UP INTEGRATION

Bottom up integration testing is a strategy in which the lower level modules are tested first.

These tested modules are then further used to facilitate the testing of higher level modules.

The process continues until all modules at top level are tested. Once the lower level modules are tested and integrated, then the next level of modules are formed.

Advantages
- Fault localization is easier
- No time is wasted waiting for all modules to be developed unlike Big bang approach.

Disadvantages
- Critical modules which control the flow of application are tested last and may be prone to defects
- An early prototype is not possible

Page No:
Date:

• SANDWICH INTEGRATION

Sandwich integration is a type of testing that consist of two parts, they are

i) top down approach
ii) bottom up approach

It combines the advantages of both Bottom up testing and top down testing at a time. Here the top level modules are tested with low level modules at the same time lower modules are integrated with top modules and tested as a system.

Advantages

- Sandwich approach is useful for very large projects having several subprojects.
- When the software product under test has modules, the size of which is comparable with that of the main product itself, then also sandwich testing has been found to be extremely handly.
- There is lesser requirements of stubs & drivers
- It takes time less to complete testing.

Disadvantages

- This testing is not suitable where system or the product is interdependences b/w the different components / modules.
- It can be more complicated to test
- In this approach, it is difficult to localize faults
- Mixed testing requires high cost
- This approach is not suitable for small projects

Page No:
Date:

Validation Testing

Like all other testing steps, validation testing tries to uncover errors but focus is at the requirements level i.e., on things that will immediately apparent to the end user.

- Validation test criteria
- Configuration Review
- Alpha and Beta testing

It ~~def~~ demonstrates the conformity with requirements

A test plan and a test procedure to be outlined in order to ensure that

- All functional requirements are satisfied
- All behavioural characteristics are satisfied
- All performance requirements are attained
- Documentation is correct
- Usability and other requirements are met

Alpha testing	Beta testing
The alpha test is conducted at the developers site by end users.	The beta test is conducted at the end users site and done by the end user.
The alpha test is conducted in controlled environment	Beta test is a live application of the software in an env. that can not be controlled by the developer

Page No:
Date:

System testing

It is a actually a series of different tests whose primary purpose is to fully exercise the computer based system.

- Recovery testing - It forces the software to fail in a variety of ways and verifies that the recovery is properly performed.
- Security testing - It verifies that protection mechanism built in a system will in fact protect it from improper penetration
- Stress testing - It is designed to confront programs with abnormal situations.
- Performance testing - It is used to test the runtime performance of ~~system~~ software within the context of an integrated system.

Regression testing

It is performed in maintenance or development phase when some modification is made in the software for either its enhancement or to remove some caught bugs It is selective ~~wrote~~ retesting of a system or the component to verify that modifications have not caused unintended effects and that the system or component is still complies with its specified requirements.

Regression testing is required when:
(i) Performance issue fix
(ii) Defect fixing
(iii) Change in needs & code is modified according to the condition
(iv) New feature added to software

Page No:
Date:

Mutation testing

It is the technique to ensure the quality of test suit. Here, the software is altered to intuse an error in it. Such intentionally altered faulty software is called mutant. Now the test suit is runn over the mutant. If the fault of the mutant is caught, mutant is said to be killed, else it is alive.

Data flow testing

It is based on the use of data structure and flow of data in the program.

Defination - The statement where data / variable defined

```
i=1; sum=0
```

Basic block - Set of consecutive statements w/o branching

```
sum = sum+1;
billvalue = billvalue + sum;
next++;
```

- C-use (computation use) : A path represents the defination and the computation of a variable called dc path.
- P-use (predicate use) : The path b/w the statement where variable is defined and the statement where it appears in a predicate called dp path.
- All use - Paths b/w the defination and all use of a variable
- do-use - Path can be identified starting from defination of a variable and ending at a point where its used but its value is not changed, called du path.

- the no. of potential backward paths increases
- becomes unmanageably large for complex programs

- **Cause Elimination Method**

Determine a list of causes
- which could have possibly contributed to error symptom
- tests are conducted to eliminate each.

A related technique of identifying error by examining error symptoms.
- software fault tree analysis

- **Program slicing**

- This technique is similar to back tracking.
- However the search space is reduced by defining slices
- A slice is defined for a particular variable at a particular statement
  - Set of source lines preceding this statement which can influence the value of the variable

Page No:
Date:

The Art of Debugging

It is the process of identifying the location of error
It occurs as a consequence of successful testing
That is when test cases uncover an error, debugging is an action that results removal of error

- **Brute force method**

print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the bugs.

- **Advantages**
  - Simplest way to test & most common approach
  - A sequence of print statements describes the dynamics of variable changes.

- **Disadvantages**
  - Cumbersome to use on large ~~probles~~ programs
  - If used indiscriminately, they can produce copious data to be analysed, much of which are superfluous

- **Backtracking**

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed.
  - Source code is traced backwards until the error is discovered
- Unfortunately as the no. of source lines to be traced back increases -

Page No:
Date:

Software Reliability and Software Maintainance

Quality engineering focusses on ensuring that products meet user expectations and predefined standards

Software Quality - Software products require more than just meeting functional requirements. They must also excel in areas such as usability, maintainability and efficiency making them more complex to define and evaluate.

- Quality Control (QC) — Focuses on detecting defects in the product through testing and inspection
  - Example: testing software for bugs before release.

- Quality Assurance (QA) — Ensures that the processes used to create the product prevent defects from occuring
  - Example: Implementing coding standards and peer reviews during development

Software Quality Factors

A high quality software product is evaluated on multiple factors:

(i) Correctness: Accurately implements the requirements specified in the Software Requirements Specifications (SRS) document.
- Ensures functional accuracy such as generating correct outputs for given inputs.

(ii) Portability : The software can run seamlessly on different operating systems, hardware platforms or in combination with other software.
Example : A web application that works on both Windows and macOS.

(iii) Reusability : Software modules can be reused in different projects with changes.
Example : A library of user authentication functions reused in multiple applications.

(iv) Usability : Users, whether experienced or novices, find the software intuitive and easy to use.
Example : A clear, well designed UI in mobile app

(v) Maintainability : Errors are easy to fix, and updates or new features can be integrated without much hassle.
Example : Clear and modular code structure makes it easy to add new functionalities.

ISO 9000 Standards

ISO 9000 is a set of international standards focused on quality management.

- ISO 9000 Series overview
  (i) ISO 9001 : Applies to organisations involved in design, development and production. Relevant for software development organisations

  (ii) ISO 9002 : For organisations only involved in production (Eg. car manufacturing)

  (iii) ISO 9003 : For organisations focused on installation and testing.

- Purpose
  Emphasizes process standardization to ensure repeatable quality. Addresses organisational aspects like resources, responsibilities and reporting.

- Benefits
  Enhances customer confidence, particularly in international markets. Ensures a well documented & structured development process, improving efficiency and cost effectiveness

- Limitations
  Does not guarantee the process quality itself or provide guidance for defining a suitable process

  Certification norms may vary due to to absence of a global accrediation body.

SEI Capability Maturity Model (CMM)

The CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University to help organisations improve their software development processes systematically.

- Levels of maturity
  (i) Initial
  - No formal process; success depends on individual efforts.
  - Ad hoc and chotic practices.

  (ii) Repeatable
  - Basic project management practices (cost, schedule and functionality tracking)
  - Success in previous projects can be replicated

(iii) Defined
- Standardized process are documented and understood across the organisation.
- ISO 9001 certification aligns with this level.

(iv) Managed
- Quantitative measures are established to track software quality and process performance.
- Measurement results are used to evaluate project outcomes

(v) Optimizing
- Continous process improvement through statistical analysis and lessons learned.
- Innovations and best practices are shared across the organisation.

• Application of CMM
(i) Capability Evaluation - Accesses an organisation ability to handle projects and helps clients select contractors

(ii) Software Process Assessment - Used internally to improve processes and enhance overall capability

ISO 9001	SEI CMM
• General quality management	• Software specific quality management
• Process standardization	• Gradual improvement of maturity levels
• All industries	• Software development
• Repeatable process	• Continous quality improvement
• Corresponds to level 3 of CMM	• Provides a roadmap to reach level 5.

Page No:
Date:

Key Process Areas (KPA's)

Each maturity level focusses on specific areas of improvement.

* **Level 2 KPA's**
  - Software project planning (cost, size, schedule)
  - Configuration and subcontract management

* **Level 3 KPA's**
  - Defined processes and documentation
  - Employee training programs

* **Level 4 KPA's**
  - Establishing quantitative measurements for quality
  - Using data to manage processes

* **Level 5 KPA's**
  - Defect prevention
  - Technology and process change management.