Page No.
Date: / /

Software Engineering

It is the application of a systematic, disciplined, quantifiable approach to the development, operation, & maintenance of software.

OR

It is a discipline whose aim is the production of fault free software that satisfies the user's needs & that is delivered on time & within budget.

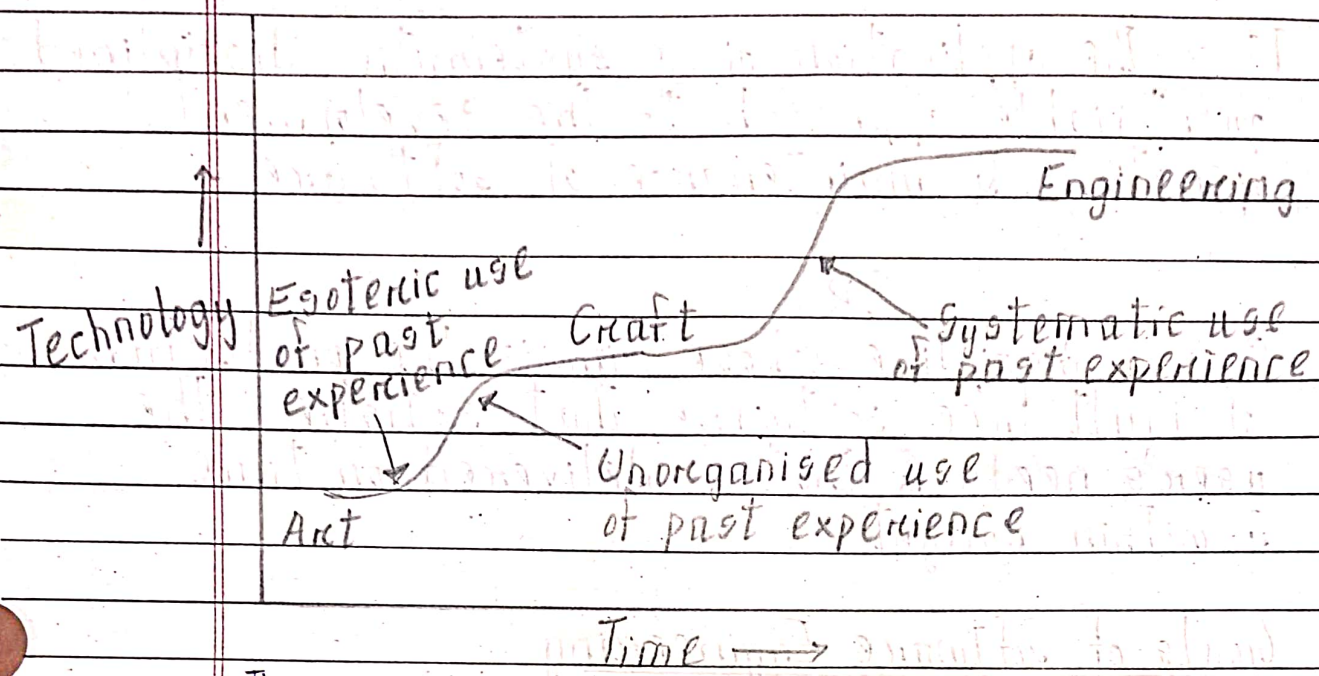
- **Goals of Software Engineering**

=> To improve the quality, reliability of software.
=> To increase productivity.
=> To increase the job satisfaction of software engineers, etc.

- **Why study Software Engineering?**

=> To acquire skills to develop large program
=> Ability to solve complex programming problems.
=> To acquire skills to be a better programmer.
=> learn techniques of - specification, design, testing, interface development etc.

- Evolution to an art to an Engineering discipline.



The graph emphasizes the progression over time & the increasing influence of technology & structured methods in transforming a creative art into a systematic engineering discipline.

- Software Crisis

When software is developing then during development many problems are raised up that set of problem is known as software crisis

Problems :

1) Schedule & cost estimates are often grossly inaccurate.

Page No.
Date: / /

2) The "Productivity" of software people hasn't kept pace with the demand for their services.

3) Communication betⁿ customer & software developer is often poor.

4) The quality of software is sometimes less than adequate.

5) The software complexity is harder to change.

- Program Vs Software Products

Programs

⇒ Combⁿ of source code & object code.

⇒ It is a subset of software & it becomes software only if documentation & operating procedure manuals are prepared.

⇒ Program = Source Code + Object Code

Software Products

⇒ Consists not only of the program code but also of all the associated documents, such as the requirements specification documents, the design & test documents, & the operating procedures, user manuals, etc.

Page No.
Date: / /

⇒ Software = Program + Documentation + Operating Procedures

Difference betⁿ Program & Software Products

	PROGRAM	SOFTWARE PRODUCTS
1	Programs are developed by individuals for their personal use.	Usually developed by a group of engineers working as a team.
2	Usually small in size.	Usually large in size.
3	Lack proper documentation.	Good documentation support.
4	Adhoc development	Systematic development
5	Have limited functionality.	Have more functionality.

- Emergence of Software Engineering

Early Computer Programming

⇒ Programs were usually written in assembly languages.

⇒ Programmers wrote programs without formulating any proper solⁿ, strategy, plan or design a jump to the terminal & start coding immediately on hearing

Page No.
Date: / /

The problem. This is called exploratory programming style.

=> Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code.

High Level Language Programming

=> At this time, high-level languages such as FORTAN, ALGOL and COBOL were introduced.

=> This considerably reduced the effort required to develop software & helped programmers to write larger problems.

=> However, programmers were still using the exploratory style of software development.

=> Typical programs were limited to sizes around a few thousand lines of source code.

Control Flow-Based Design

=> A program's control flow structure indicates the sequence in which the program's inst^ns are executed.

=> To design good control flow structures, the flow charting technique is developed.

Page No.
Date: / /

=> The flow charting technique is being used to represent & design algorithms; though the popularity of the flow charting technique has diminished due to more advanced techniques.

=> A program having a messy flow chart representation would indeed be difficult to understand & debug.

Structured Programming

=> A program is called structured when it uses only the following types of constructs:
i) Sequence
ii) Selection
iii) Iteration

=> Consists of neat set of modules with single entry & single exit program.

=> Easier to read, understand & maintain & require less effort for development.

=> Avoid unstructured control flows by restricting the use of GO TO statements.

Data Structure - Oriented Design

=> It is much more important to pay attention to the design of the important data

structures of the program than to the design of its control structure. Design techniques based on this principle are called data structure-oriented design techniques.

=> First the data structure is designed & then the program structure is derived out of the data structure design.

Eg. Jackson's Structured Programming (JSP)
Warnier-Orr Methodology

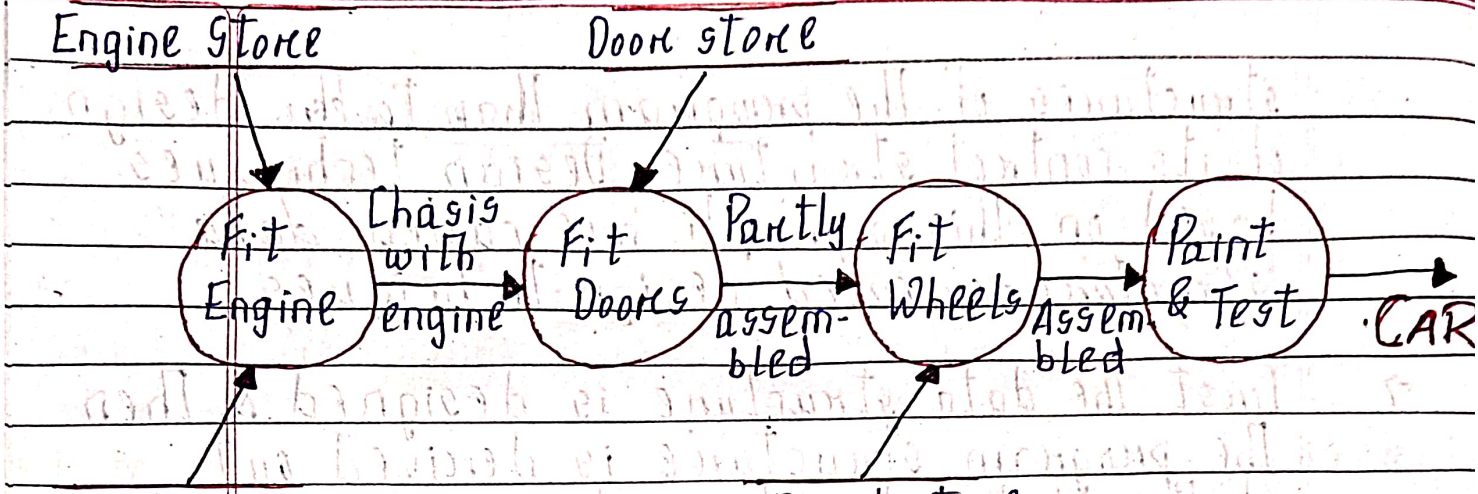
Data Flow-oriented Design

=> The data flow-oriented techniques advocate the following :-

i) Data items input to a system must be identified first.
ii) Processing required on the data items to produce the required outputs should be determined.

=> The func^n's & the data items that are exchanged bet^n the func^n's are represented in a diagram known as data flow diagram (DFD).

=> DFD is a generic technique which can be used to model all types of system other than software

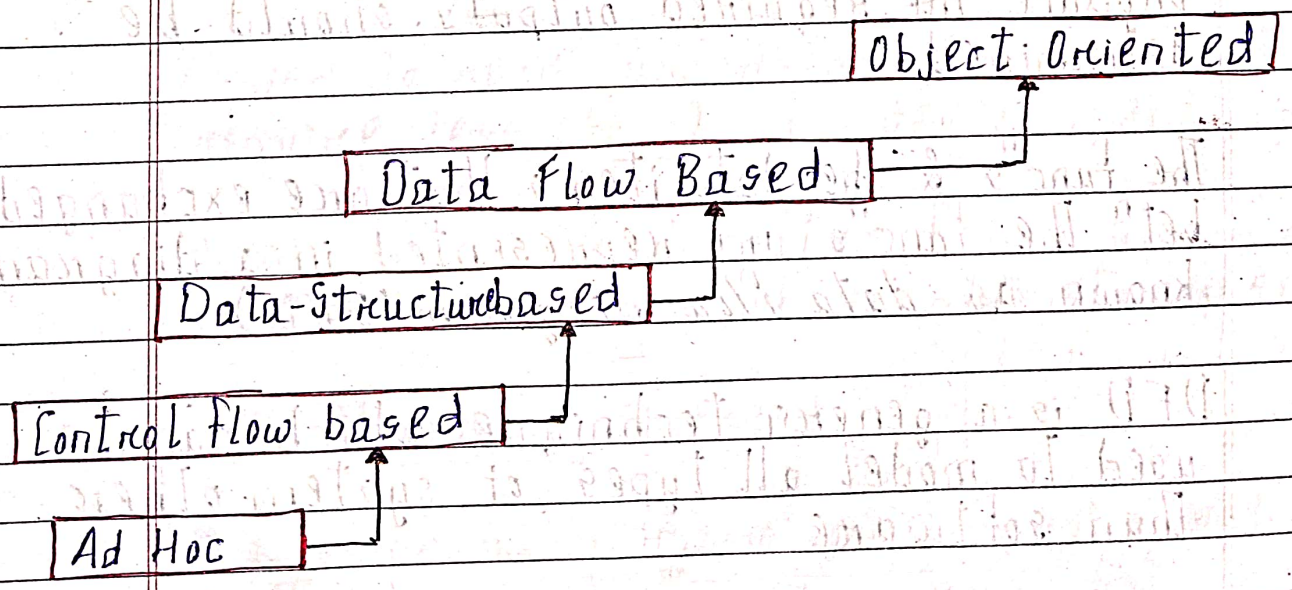


Object Oriented Design

⇒ It is an intutive approach where natural objects occurring in a problem are first identified & then the relationship betⁿ objects are determined.

⇒ Each object acts as a data hiding entity.

⇒ Advantages :- Simplicity, Lower development time, robust code.



Page No.
Date: / /

Changes in S/W Development Practices

=> The exploratory software development style is based on error correction while the software engineering techniques are based on the principles of error prevention.

=> In the exploratory style, coding was considered synonymous with software development. But in the modern software development style, coding is regarded as only a ~~part~~ small part of the software development activities.

=> In the modern software development style, there is distinct design phase where std design techniques are employed to yield coherent & complete design models.

=> In the modern software development style, periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment errors i.e. detect & correct errors as soon as possible.

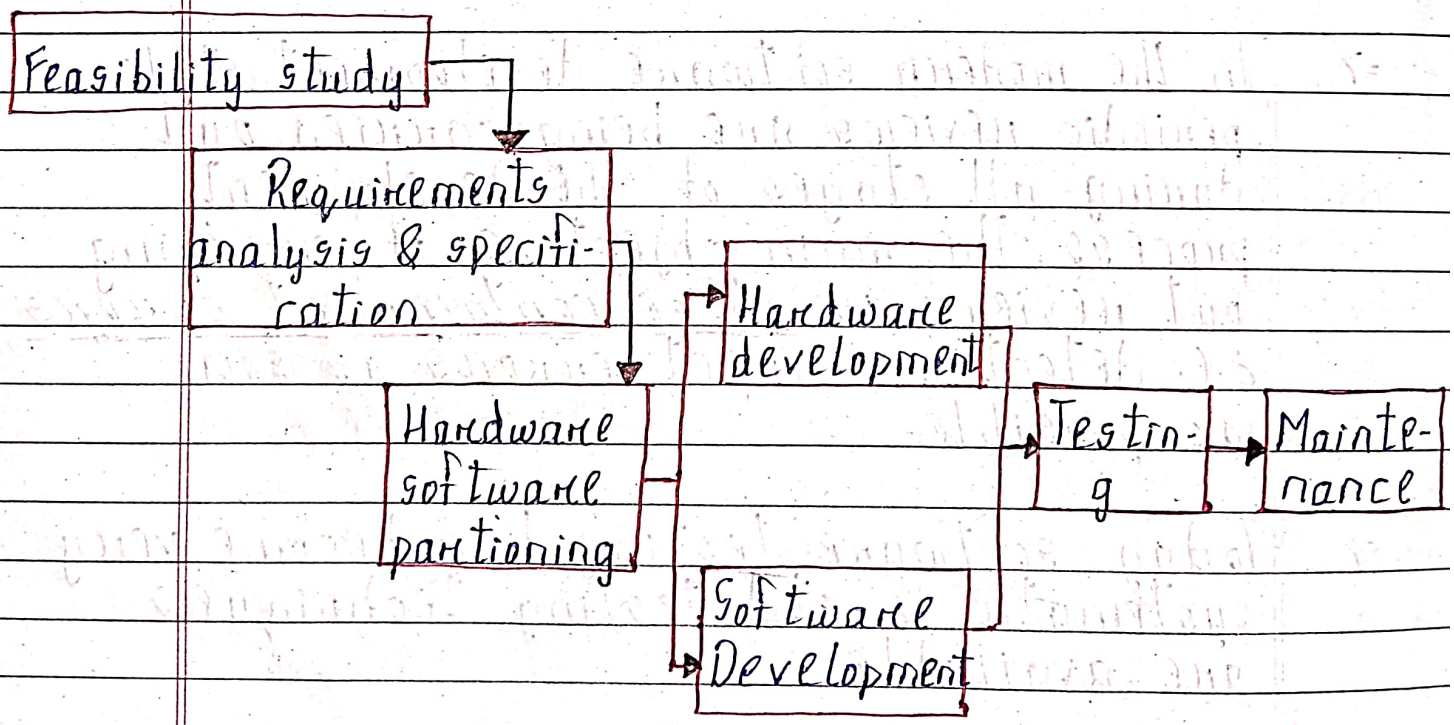
=> Today, software testing has become very systematic & std testing techniques are available.

⇒ In the exploratory style, the design & test activities, even if carried out, were not documented satisfactorily. But in modern software development, good quality documents are being developed.

⇒ Now, projects are being thoroughly planned. The primary objective of project planning is to ensure various development activities take place at the correct time & no activity is halted due to the want of some resource.

- Computer Systems Engineering

⇒ Computer Systems Engineering addresses development of such systems requiring development of both software & specific hardware to run the software.



Page No.
Date: / /

⇒ One of the important stages in systems engineering is the stage in which decision is made regarding the parts of the problems that are to be implemented in hardware & the ones that would be implemented in software.

⇒ The functionality implemented in hardware run faster. On the other hand, functionalities implemented in software is easier to extend.

⇒ After the hardware-software partitioning stage, development of hardware & software are carried out concurrently.

⇒ To test the software during development, it usually becomes necessary to develop simulators that mimic the features of the hardware being developed. The software is tested using these simulators.

Page No.
Date: / /

SOFTWARE LIFE CYCLE

=> A software life cycle model is a pictorial & diagrammatic representation of the software life cycle.

=> A life cycle model represents all the methods required to make a software product transit through its life cycle stages.

Need for Software Life Cycle

Without using an exact life cycle model, the development of a software product would not be in a systematic & disciplined manner.

When a team is developing a software product, there must be clear understanding among team representative about when & what to do.

A software life cycle model describes entry & exit criteria for each phase. A phase can begin only if its stage-entry criteria have been fulfilled. So, without a software life cycle model, the entry & exit criteria for a stage cannot be recognised.

Page No.
Date: / /

SDLC Cycle

It represents the process of developing software.

The stages of SDLC are as follows :-

Stage 1 : Planning & Requirement Analysis

=> The requirement is the 1st stage in the SDLC process.

=> Planning for the quality assurance requirement & recognition of the risks involved is also at this stage.

=> The information from this analysis forms the building blocks of a basic project.

=> The quality of the project is a result of this planning.

Stage 2 : Feasibility Study

=> Once the requirement analysis phase is completed the next SDLC step is to define & document software needs.

=> This process is conducted with the help of "Software Requirement Analysis" document OR SRS Document

Page No.
Date: / /

=> It includes everything which should be design & developed during the project life cycle.

Step 3 : Design

=> In this phase, the system & S/W design documents are prepared as per the requirement specification document. This helps define overall system architecture.

Stage 4 : Coding

=> Once the design phase is over, the next phase is coding.

=> In this phase, developers start to build the entire system by writing code using the chosen programming language.

=> The developers needs to follow certain pre-defined coding guidelines. They also need to use programming tools like compiler, interpreter, debugger to generate & implement the code.

Step 5 : Testing

=> After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed & gathered during the

requirements stage.

Stage 6 : Deployment

⇒ Once the software is certified, & no bugs or errors are stated, then it is deployed.

⇒ Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.

Stage 7 : Maintenance

⇒ Once when the client starts using the developed systems, then the real issues come up & requirements to be solved time to time.

Requirement Analysis

Feasibility Study

Design

Coding

Testing

Maintenance

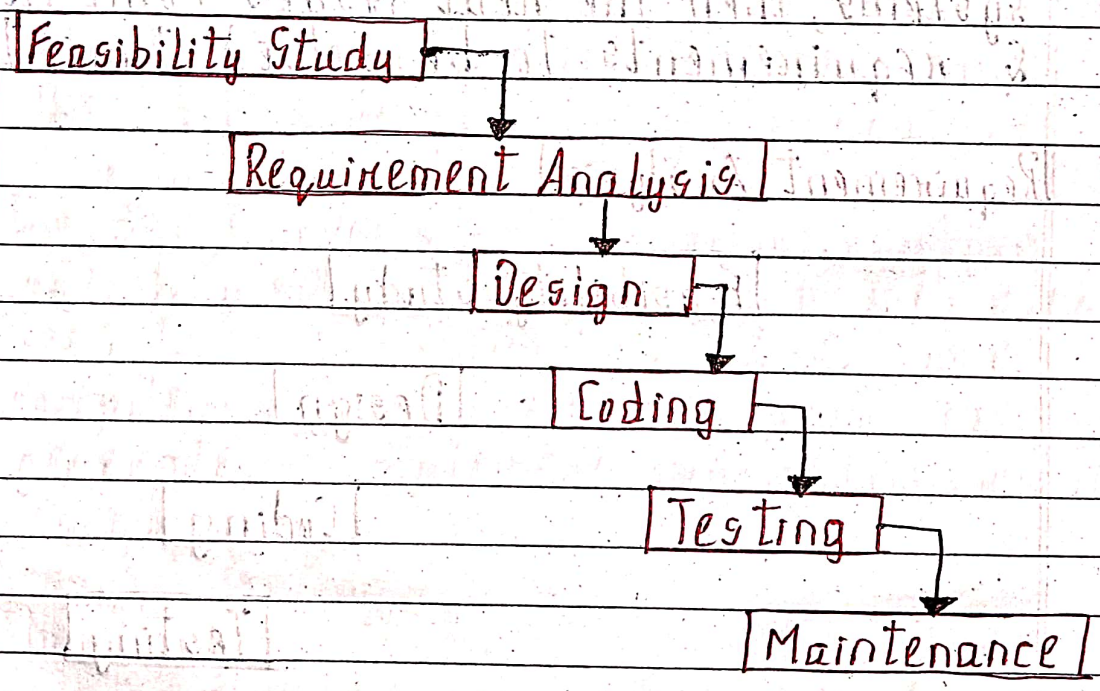
1) **WATERFALL MODEL**

⇒ Classical software development methodology.

⇒ Linear & sequential approach to software development that consists of several phases

- **Features**
  - → Sequential Approach
  - → Document - Driven
  - → Quality Control
  - → Rigorous Planning

- **Phases**



Page No.
Date: / /

i) Feasibility Study

=> The main goal of this phase is to determine whether it would be financially & technically feasible to develop the software.

=> Involves understanding the problem & then determining the various possible strategies to solve the problem.

=> The best sol^n is chosen after analysing all diff. identified sol^{n}s based on their benefits & drawbacks.

ii) Requirements Analysis & Specification

=> Aims to understand the exact requirements of the customer & document them properly.

=> It consists of 2 diff. activities :

Requirement Gathering & analysis : First all the requirements regarding the software are gathered from the customer & then the gathered requirements are analysed. The goal of the analysis part is to remove incompleteness & inconsistencies.

Requirement Specification : These analyzed requirements are documented in SRS document.

Page No.
Date: / /

iii) Design

The goal of this phase is to convert the requirements acquired in the SRS into a format that can be coded in a programming language.

A Software Design Document (SDD) is used to document all of this effort.

iv) Coding

In the coding phase, the S/W design is translated into source code using any programming language.

v) System Testing

α-Testing : Testing performed by the development team.
β-Testing : Testing performed by the set of customers.
Acceptance Testing : Customer performs acceptance testing to determine whether to accept it or reject it.

vi) Maintenance

The effort spent on maintenance is 60% of the total effort spent to develop a full software.

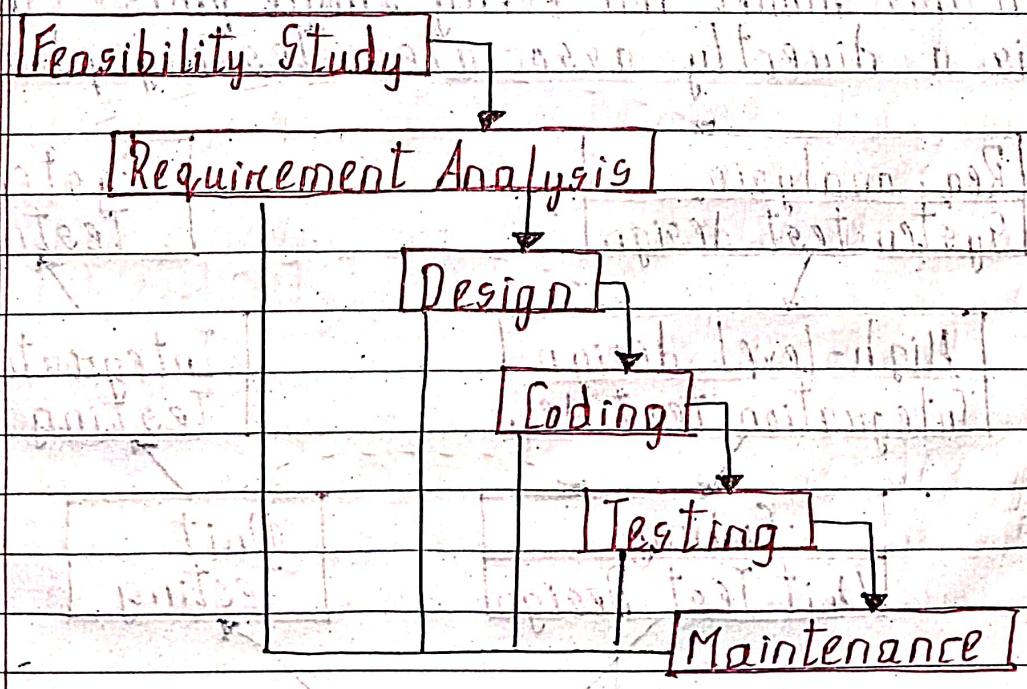
Corrective maintenance :- Carried out to correct errors that were not discovered during development phase.
Perfective maintenance :- Carried out to enhance the functionalities of the system based on customer's request.
Adaptive maintenance :- Required for porting the software to work in a new environment.

- Disadvantages

→ No feedback Path
→ Lengthy Development Cycle
→ Difficult to accomodate Change Requests.

2) ITERATIVE WATERFALL MODEL

⇒ Provides feedback paths from every phase to its preceding phase.



Phase containment of errors

The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors.

Disadvantages

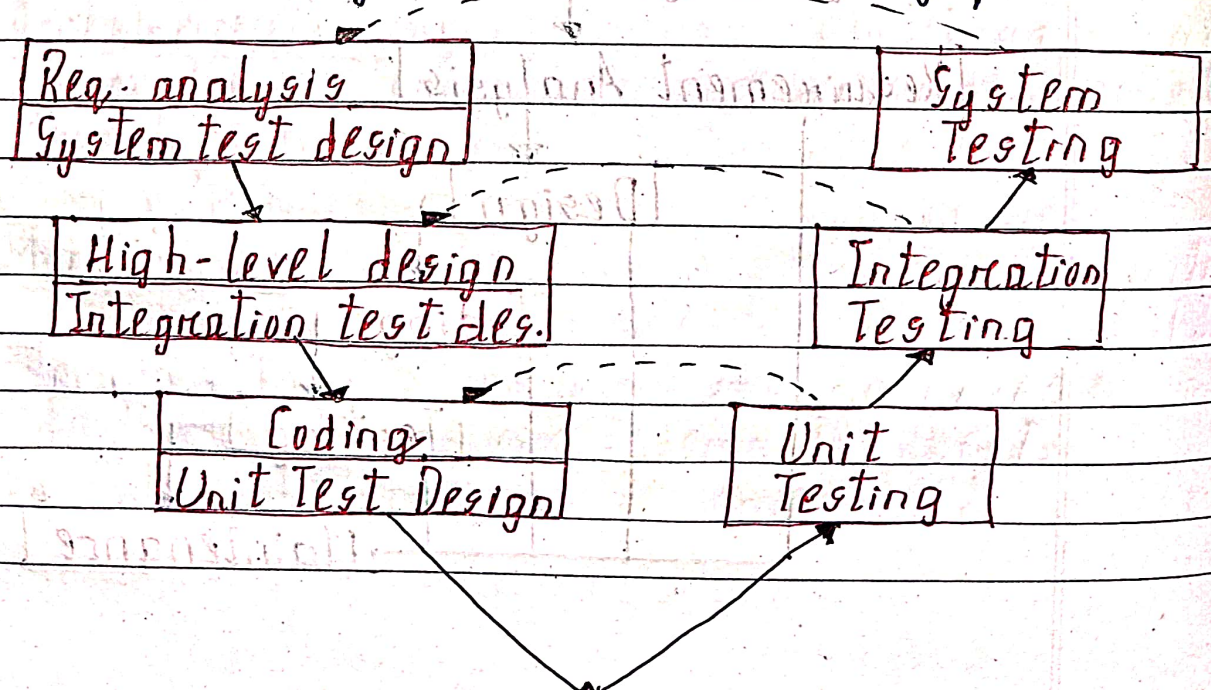
→ Difficult to accomodate change requests
→ Phase overlap not supported.
→ Incremental delivery not supported
→ limited customer interactions

3) V-MODEL

⇒ Execution of processes happens in a sequential manner in a V-shape

⇒ Also known as Verification & Validation model.

⇒ In this model, for every single phase, there is a directly associated testing phase.



Page No.
Date: / /

⇒ There are 2 main phases:
i) Development phase (Left)
ii) Validation phase (Right)

i) In development phase, along with the development of a work product, test case design & the plan for testing the work product carried out.

ii) In validation phase, testing is carried out in 3 steps - unit, integration & system testing. The purpose is to detect defects that arise in the corresponding phases.

Advantages

→ Clear & structured process
→ Emphasis on testing
→ Improved traceability
→ Better Communication.

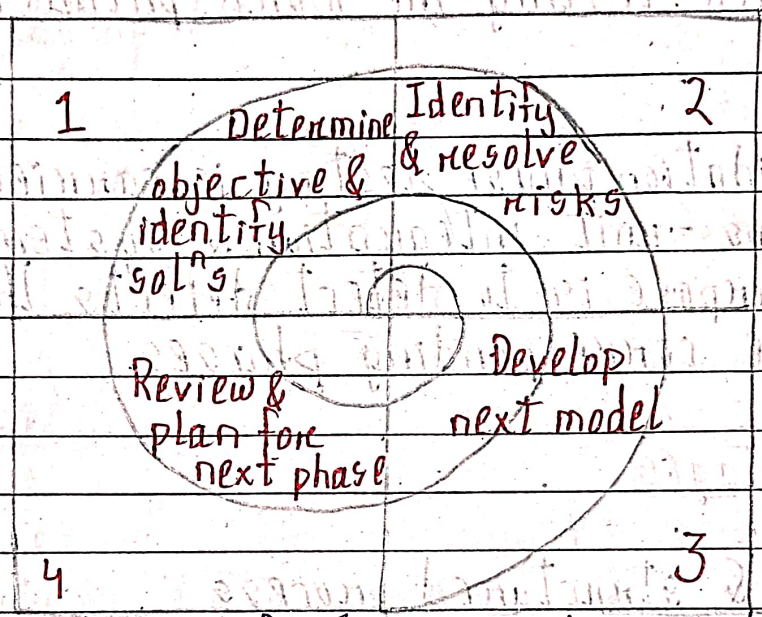
Disadvantages

→ Does not support iteration of phases.
→ Time consuming
→ Inflexibility.

4) SPIRAL MODEL

=> Combⁿ of the waterfall & iterative model.

=> Used for complex & large software development.



Each phase of the spiral model is divided into 4 quadrants.

Quadrant 1 : Requirements are gathered from the customers & the objectives are identified, elaborated, & analyzed at the start of every phase. Then alternative solⁿs possible for the phase are proposed.

Quadrant 2 : All the possible solⁿs are evaluated to select the best possible solⁿ. Then the risks associated with that solⁿ are identified & the risks are resolved using the best possible strategy. At the end of this

Page No.
Date: / /

quadrant, the prototype is built for the best possible sol^n.

Quadrant 3: The identified features are developed & verified through testing. At the end of this quadrant, the next version of the software is available.

Quadrant 4: The customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

Risk Handling in Spiral Model

The most important feature of the spiral model is handling these unknown risks after the project has started.

=> The spiral model supports coping with risks by providing the scope to build a prototype at every phase of software development.

=> In each phase, the features of the product dated & analysed, & the risks at that point in time are identified & are resolved through prototyping.

Page No.
Date: / /

Why Spiral Model is called Meta Model?

The spiral model is called meta model because it subsumes all other SDLC models.

=> A single loop spiral actually represents The Iterative Waterfall Model.

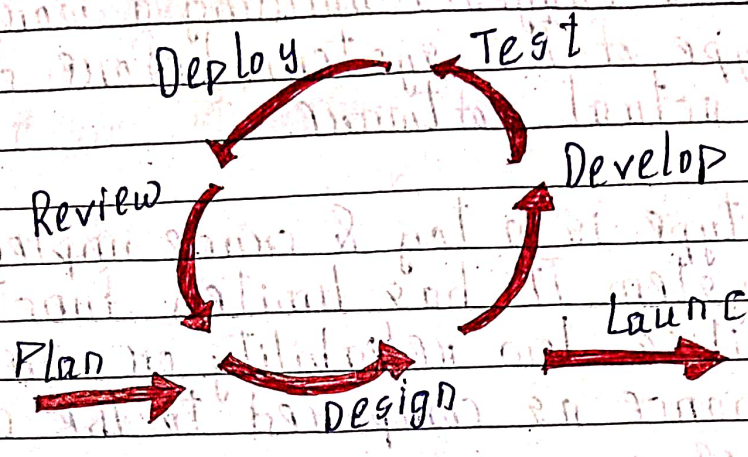
=> It incorporates The stepwise approach of The Classical Waterfall Model

=> It uses The approach of The Prototyping Model by building a prototype at The start of each phase as a risk-handling technique.

=> It can be considered as ~~the~~ supporting The Evolutionary Model - The iterations along The spiral can be considered as evolutionary levels through which The complete system is built.

5) AGILE MODEL

=> The Agile Model refers to a group of development processes. These processes share some basic characteristics but do not have certain subtle differences among themselves.



⇒ In The Agile model, The requirements are decomposed into many small parts That can be incrementally developed.

⇒ The Agile model adopts Iterative development Each incremental part is developed over an iteration. At a time, one iteration is planned, developed & deployed to The customers.

⇒ The time required to complete an iteration is known as Time-Box. It refers to The max^m amount of time needed to deliver an iteration to customers. So, The end date for an iteration does not change.

Page No.
Date: / /

6) PROTOTYPING MODEL

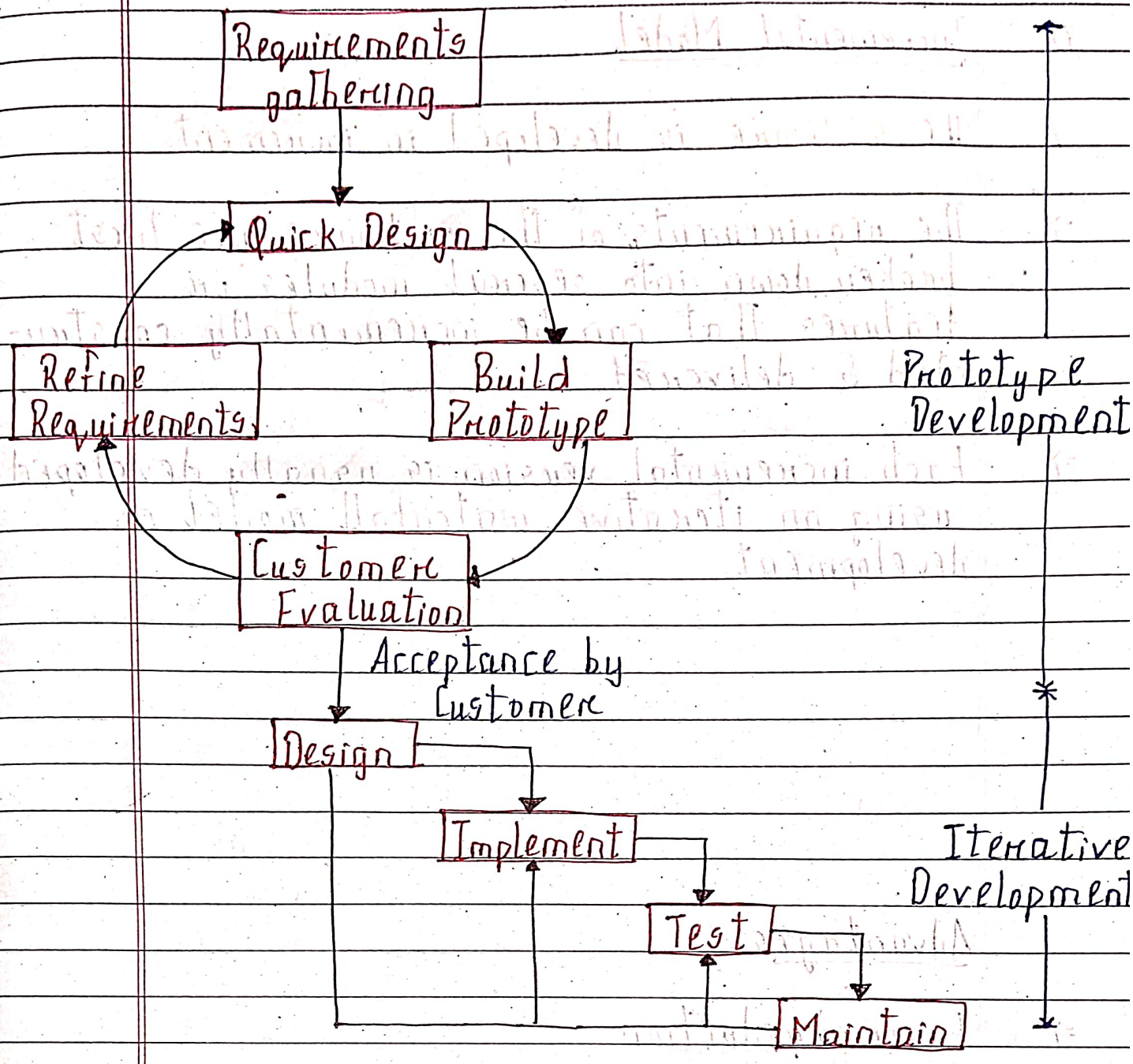
⇒ This model suggests building a working prototype of the system, before development of the actual software.

⇒ A prototype is a toy & crude implementation of a system. It has limited functional capabilities, low reliability or inefficient performance as compared to the actual software.

Necessity of the Prototyping Model

⇒ Through the use of prototype, it becomes easier to illustrate the data input formats, message reports & the interactive dialogs to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs.

⇒ It is especially useful when the exact technical sol^{n}s are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development.


Requirements gathering
Quick Design
Refine Requirements
Build Prototype
Customer Evaluation
Acceptance by Customer
Design
Implement
Test
Maintain
Prototype Development
Iterative Development

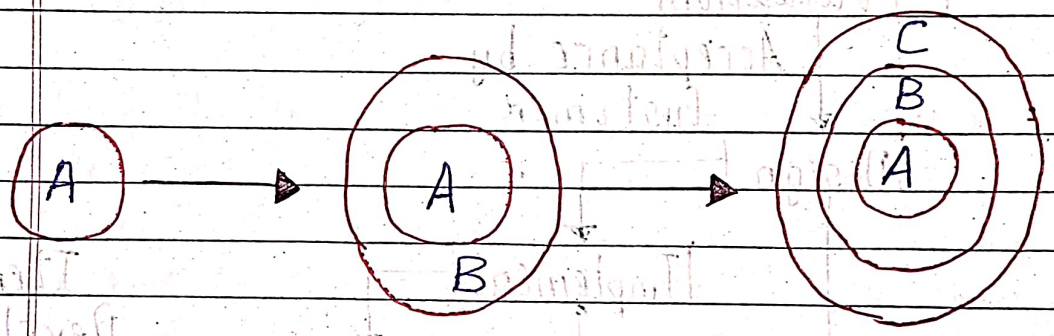
Page No.
Date: / /

7) Incremental Model

=> The software is developed in increment.

=> The requirements of the software are first broken down into several modules or features that can be incrementally constructed & delivered.

=> Each incremental version is usually developed using an iterative waterfall model of development.



Advantages

=> Error reduction

=> Incremental resource deployment.

Page No.
Date: / /

A Comparison of Different Life Cycle Models

The classical waterfall model can be considered as the basic model & all other life cycle models are embellishments of this model. However, the classical model cannot be used in practical development projects, since this model supports no mechanisms to correct the errors that are committed during any of the phases but detected at a later phase.

The iterative waterfall model is the most widely used software development model. This model is simple to understand & use. However, this model is suitable ~~for development~~ only for well-understood problems, & is not suitable for development of very projects & projects that suffer from large no. of risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood, however all risks can be identified before the project starts. This model is popular for development of user interface part of projects.

The evolutionary model is suitable for large problems which can be decomposed into a set of modules for incremental development & delivery. Used widely for object oriented development projects.

Page No.
Date: / /

The Spiral Model is suitable for development of technically challenging & large software that are prone to several kind of risks that are difficult to anticipate at the start of the project.

99 percent complete' Syndrome

It refers to a common issue where a project appears to be almost finished but experiences unexpected delays or complications that prevent its finalization.

Page No.
Date: / /

REQUIREMENT ANALYSIS AND SPECIFICATION

The goal of the requirements analysis & specification phase is to clearly understand the customer requirements & to systematically organise the requirements into a document called Software Requirements Specification (SRS) Document.

- **Requirements Gathering**

=> Also popularly known as requirements elicitation.

=> The primary objective is to collect the requirements from the stakeholders.

=> Gathering documents turns out to be especially challenging if there is no working model of the software to be developed.

=> If the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input & output forms & the details of the operational procedures.

=> Following are the ways which an experienced analyst gathers requirements :-

Page No.
Date: / /

i) Studying existing documentation :- The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. These documents might discuss issues such as the context in which the software is required, the basic purpose, features of any similar software developed elsewhere etc.

ii) Interview :- It is important for the analyst to first identify the diff. categories of users & then determine the requirements of each. To systematise this, the Delphi technique is followed. In this technique, the analyst consolidates the requirements as understood by him in a document & then circulates it for the comments of the various categories of users. Based on their feedback, he refines his document.

iii) Task Analysis :- A service supported by a software is called a task. ~~Task~~ The analyst tries to identify & understand the tasks to be performed by the software. For each identified task, the analyst tries to ~~identify & understand~~ formulate the diff. steps necessary to realize the required functionality in consultation with the users.

Page No.
Date: / /

iv) Scenario Analysis :- A task can have many scenarios of operation. The diff. scenarios of a task may take place when the task is invoked under diff. situations. For various identified tasks, the possible scenarios of execution are identified & details of each scenario is identified in consultation with the users.

v) Form Analysis :- It is an important & effective activity that is undertaken by the analyst, when the project involves automating an existing manual system.

- Requirements Analysis

=> The main purpose of the requirements analysis is to analyse the gathered requirements to remove all ambiguities, incompleteness & inconsistencies from the gathered customer requirements & to obtain a clear understanding of the software to be developed.

=> During requirements analysis, the analyst needs to identify & resolve 3 main types of problems in the requirements :-

a) Anamoly
b) Inconsistency
c) Incompleteness

Page No.
Date: / /

a) Anomaly : When a requirement is anamalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anamalous requirement can be interpreted in several ways during development.

b) Inconsistency : Two requirements are said to be inconsistent, if one of the requirements contradict the other.

c) Incompleteness : An incomplete set of requirements is one in which some requirements have been overlooked. Incompleteness is caused by the inability of the customer to visualise the system to be developed & to anticipate all the features that would be required.

- Software Requirements Specification (SRS)

=> SRS document is a complete description of requirements of the software that need to be fulfilled for the successful development of the software system.

=> It contains all the user requirements in a structured informal form.

Page No.
Date: / /

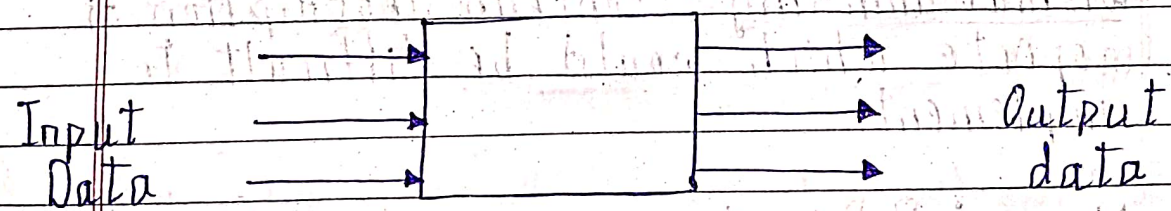
Users of SRS Document :-

1) Users, customers, & marketing personnel
2) Software Developers
3) Test Engineers
4) User documentation writers
5) Project Managers
6) Maintenance Engineers

Uses of a SRS Document :-

→ Forms an agreement betn the customers & the developers.
→ Reduces future reworks
→ Provides a basis for estimating costs & schedules
→ Provides a baseline for validation & verification.

The SRS document is known as <u>black box specification</u> as it describe the system to be developed as black box & should specify only the externally visible behaviour of the system.



Page No.
Date: / /

Characteristics of good SRS Document :

=> It should be concise without ambiguity.
=> It should specify what the system must do & not how to do it.
=> It should be consistent.
=> It should be complete.
=> It should be traceable.
=> Easy to change i.e. it should be well-structured.

Characteristics of Bad SRS Document :

=> Over-specification : It occurs when the analyst tries to address the "how to" aspects in the SRS document. Restricts the freedom of the designers in arriving at a good design sol^n.

=> Forward Reference : One should not refer to aspects that are discussed much later in the SRS document. Reduces readability of the specification.

=> Wishful Thinking : Concern description of aspects which would be difficult to implement.

=> Noise : Refers to presence of material not directly relevant to software development process.

An SRS document should clearly document the following aspects of a software:-

i) Functional requirements.
ii) Non-functional requirements
iii) Goals of implementation

ii) <u>Functional Requirements</u>

=> Capture the functionalities required by the users from the system.

$f : I \rightarrow O$ — output
Input

Input — f_i — Output

=> Describe each functionality that the system would support along with the corresponding input & output data set.

=> In order to document the functional requirements of a system, it is necessary to identify the high-level func^{n}s of the systems by reading the informal documentation of the gathered ~~docume~~ requirements.

=> Each high level requirement involves accepting some data from the user, transforming it to the required response & displaying the system response in proper format.

Page No.
Date: / /

High level Func^{n}s :-

=> Involves a series of interactions bet^n the system & one or more users.

=> For any given high-level func^n, there can be diff. interaction sequences or scenarios due to users selecting diff. options or entering diff. data items.

ii) Non-functional Requirements

=> These are non-negotiable obligations that must be supported by the software.

Following are the characteristics of the system which cannot be expressed as func^{n}s :-

→ Reliability issues
→ Performance issues
→ Security, maintainability
→ Human computer interface issues
→ Interface with other external systems.

iii) Goals of Implementation

=> Might document issues such as easier revisions to the system functionalities that may be required in the future, easier support for new devices to be supported in future etc.

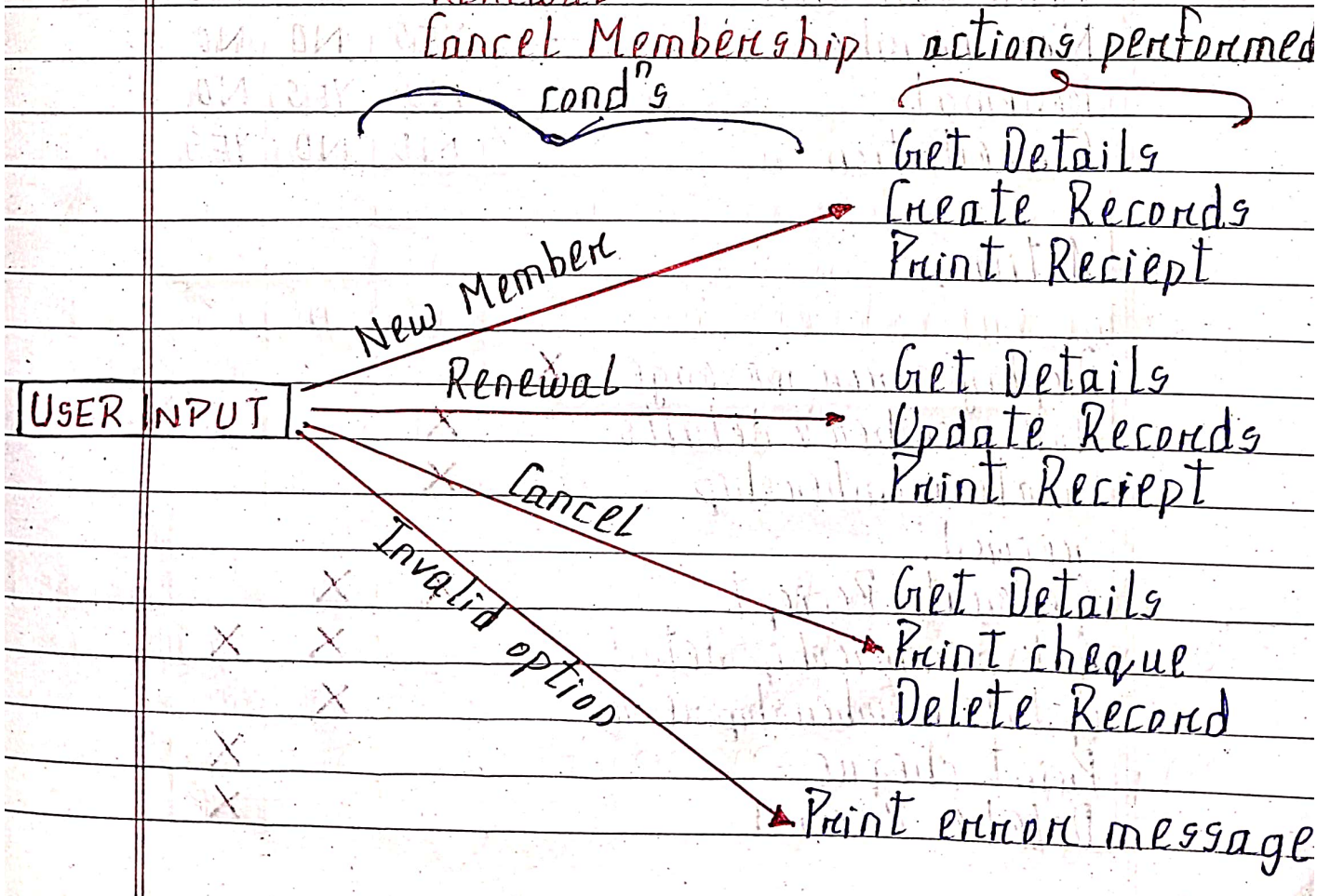
Decision Tree

=> Gives graphic view of the processing logic involved in decision making & the corresponding actions taken.

=> The edges of a decision tree represent cond^{n}s & the leaf nodes represent the actions to be performed depending on the outcome of testing the cond^{n}s.

Case Study :-

A library Membership automation Software (LMS) should support the following 3 options :- New member
Renewal
Cancel Membership



Decision Table

=> Shows the decision making logic & the correspondi actions taken in tabular or matrix form.

=> The upper rows specify the variables / cond^{n}s to be evaluated & the lower rows specify the action to be taken when the corresponding cond^{n}s are satisfied.

=> A column in the table is called rule. A rule implies : if cond^n is true, then execute the corresponding action.

Cond^{n}s				
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 member's details		X		
Create membership record.		X		
Generate Reciept		X	X	
Ask membership details			X	X
Update membership record			X	
Print cheque				X
Delete Record				X

SOFTWARE DESIGN

=> During the software design phase, the design document is produced, based on the customer requirements as documented in a SRS document.

SRS document → Design Document

Outcome of Design Process

The following items are designed & documented during the design phase :-

=> Different modules required :- The diff. modules in the solⁿ should be identified. Each module is a collection of funcⁿs & data shared by these funcⁿs.

=> Control relationships among modules :- Arises due to funcⁿ calls across the 2 modules. The control relationship existing among various modules should be identified in the design document.

=> Interfaces among diff. modules :- The interface betⁿ 2 modules identifies the exact data items that are exchanged betⁿ the 2 modules when one module invokes a funcⁿ of the other module.

Page No.
Date: / /

⇒ Data structures of the individual modules :
Suitable data structures for storing & managing the data of a module need to be properly designed & documented.

⇒ Algorithms required to implement the individual modules.

- Classification of Design Activities

Classified into 2 stages :-
1) High level design
2) Low level design

1) High Level Design

⇒ In high level design, a problem is decomposed into a set of modules, where control relationships among the modules & the various interfaces among the modules are identified.

⇒ The outcome of high level design is called software architecture.

2) Detailed Design

⇒ In detailed design, each module is examined carefully to design its data structures & the algorithms.

Page No.
Date: / /

=> The outcome of detailed design is ~~call~~ in the form of module specification (MSPEC) document.

- **Characteristics of Good Software Design**

1) Correctness :- A good design should correctly implement all the functionalities of a system.

2) Understandability :- A good design should be easily understandable.

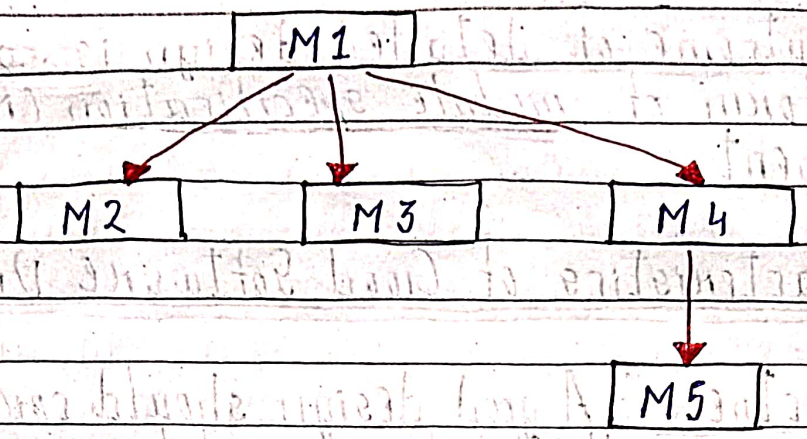
3) Efficiency :- A good design sol^n should adequate address resource, time, & cost optimisation issues.

4) Maintainability :- A good design should be easy to change, since change requests usually keep coming from the customer even after product release.

- **Modular Design**

=> A modular design is an approach where a system is divided into smaller, independent & reusable modules.

=> Each modules performs a specific func^n & interacts with other modules through well-defined interfaces.



A design solⁿ is said to be highly modular, if the ~~sol~~ diff. modules in the solⁿ have high cohesion & their inter-module coupling are low.

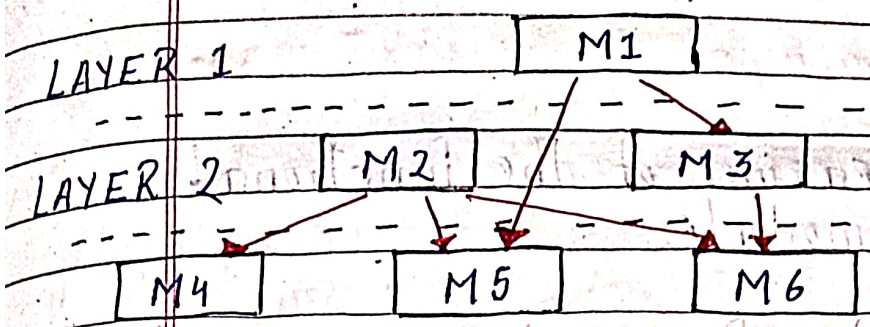
- Layered Design

=> The modules are arranged in a hierarchy of layers.

=> Considered to be implementing control abstraction, since a module at lower layer is unaware of higher layer modules.

Fan-in :- Indicates the no. of modules that directly invoke a given module. High fan-in represents code reuse & is desirable in a good design.

Fan-out :- Measures the no. of modules that are directly controlled by a given module.



	Fan-in	Fan-out
M1	0	2
M2	0	3
M3	1	1
M4	1	0
M5	2	0
M6	2	0

Functional Independence

=> A functionally independent module has minimum interaction with other modules.

=> A module having high cohesion & low coupling is said to be functionally independent of the other modules.

Advantages :-

=> Better understandability & good design.

=> Complexity of design is reduced.

=> Reduces error propagation.

Cohesion

=> Cohesion is a measure of the functional strength of a module.

=> When the func^n's of a module co-operate with each other for performing a single objective, then the module has good cohesion.

=> If the func^n's of a module do not cooperate with each other to perform a single piece of work, then the module has poor cohesion.

Classification of Cohesion

- Functional — High
- Sequential
- Communicational
- Procedural
- Temporal
- Logical
- Coincidental — Low

Page No.
Date: / /

1) Coincidental Cohesion :- A module is said to ~~be~~ have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.

2) Logical Cohesion :- A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output etc.

3) Temporal Cohesion :- When a module contains func^{n}s that are related by the fact that these func^{n}s are executed in the same time span, then the module is said to possess temporal cohesion.

4) Procedural Cohesion :- A module is said to possess procedural cohesion, if the set of func^{n}s of the module are executed one after other, though these func^{n}s may work towards entirely diff. purposes & operate on very diff. data.

5) Communicational Cohesion :- A module is said to have communicational cohesion, if all func^{n}s of the module refer to or update the same data structure.

6) Sequential Cohesion :- A module is said to possess sequential cohesion, if the diff. func^{n}s of the module, execute in a sequence, & the output

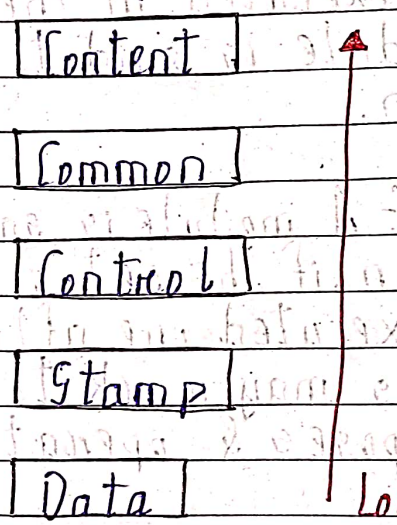
from one func^n is input to the next sequence.

7) Functional Cohesion :- A module is said to posses functional cohesion, if the diff func^{n}s of the module co-operate to complete a single task.

- Coupling

The coupling bet^n 2 modules is a measure of the degree of interaction bet^n the 2 modules.

Classification of Coupling



1) Data Coupling :- Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter bet^n the two e.g. an integer, a float etc.

2) Stamp Coupling :- Two modules are stamp coupled, if they communicate using a composite item such as a record in PASCAL.

Page No.
Date: /

3) Control Coupling :- Control coupling exists betn two modules, if data from one module is used to direct the order of instn execution in another.

4) Common Coupling :- Two modules are common coupled, if they share some global data items.

5) Content Coupling :- Content coupling exists betn two modules, if they share code. That is, a jump from one module into the code of another module can occur.

FUNCTION ORIENTED SOFTWARE DESIGN

=> FO design techniques are very popular & currently being used in many software development projects.

=> Top-down decomposition is used to denote the successive decomposition of a set of high level func^n's into more detailed func^n's.

FO Design

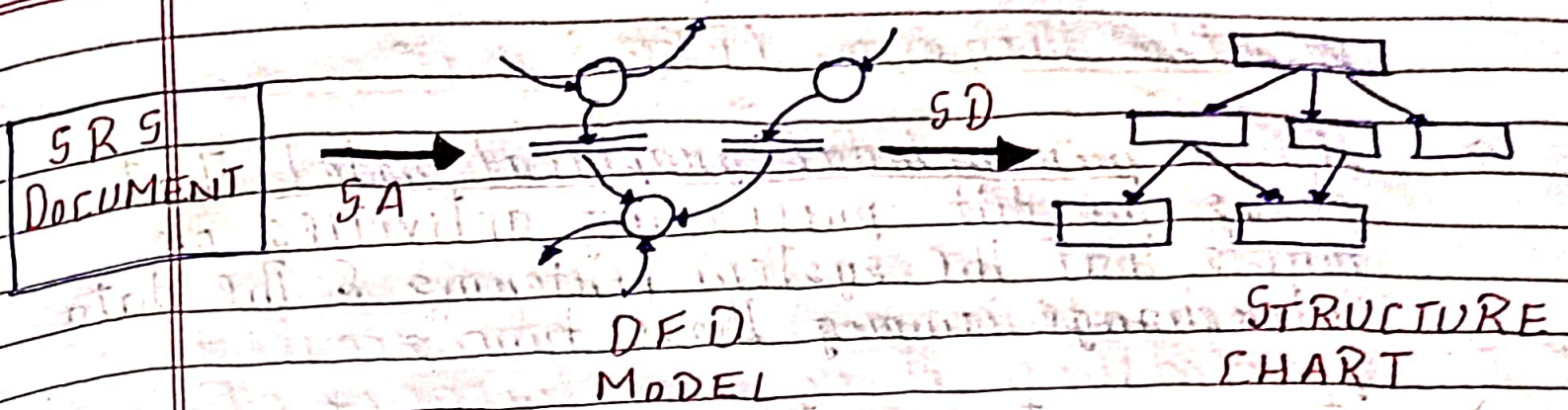
Structure Analysis (SA) | Structure Design (SD)

=> SA/SD technique is drawn from many methodology :-

1) Constantine & Yourdon
2) Hatley & Pirbhai
3) Gane & Sarson
4) DeMarco & Yourdon

=> The SA activity transforms the SRS document into a graphical model called the DFD model.

=> During SD, all func^n's identified by SA are represented using a structure chart.



NOTE The purpose of SA is to capture the detailed structure of the system as perceived by the user.

The purpose of SD is to define the structure of the sol^n that is suitable for implementation in some programming languages.

Structured Analysis

⇒ Based on the following principles :-

→ Top-down decomposition approach.

→ Application of divide & conquer principle.

→ Graphical representation of the analysis results using DFDs

⇒ The results of SA technique can be easily understood by the user.

Page No.
Date: / /

Data Flow Diagrams (DFDs)

=> It is a hierarchical graphical model that shows the diff. processing activities or func^n's that the system performs & the data interchange among those func^n's.

=> It represent a system in terms of the input data to the system, various processing carried out on those data, & the output data generated by the system.

DFD Symbols

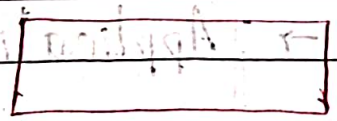
1) Function Symbol

=> Represented using a circle.

=> known as process OR bubble.

2) External Entity

=> Represented by a rectangle.

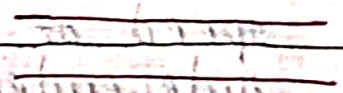


=> These are the physical entities which interact with the system by inputting the data to the system or by consuming the data produced by the system.

3) Data Flow Symbol

=> Represented by directed arc.

=> Represents data flow in the dir^n of the arrow.

4) Data Store Symbol 

=> Represented using 2 parallel lines.

=> Represents a logical file i.e. a data structure OR a physical file on disk.

=> Each data store is connected to a process by means of a data flow symbol.

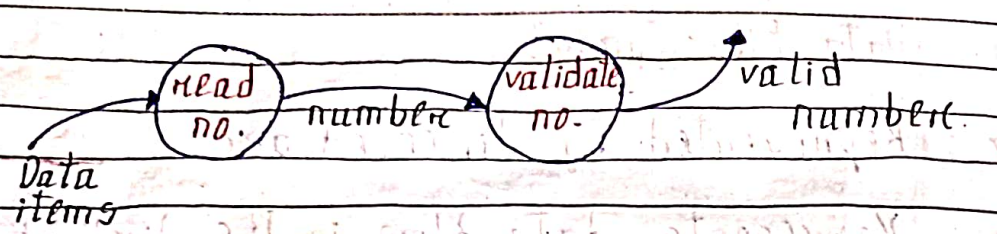
=> Dir^n of data flow arrow shows whether data is being read from OR written into it.

5) Output Symbol

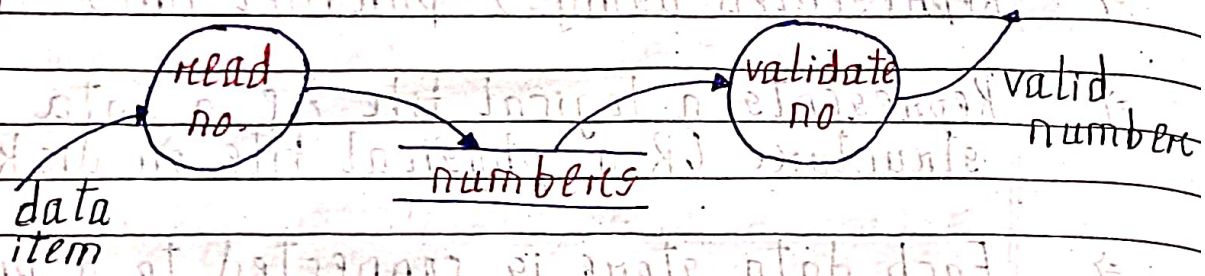
=> Used when a hard copy is produced.

Synchronous & Asynchronous Operation

=> If two bubbles are directly connected by a data flow arrow, then they are synchronous i.e. they operate at the same speed.



=> If two bubbles are connected through a data store, then it is asynchronous i.e. the speed of operation of the bubbles are independent.



Data Dictionary

=> A data dictionary lists the purpose of all data items & the definition of all composite data items in terms of their component data items.

=> Every DFD model must be accompanied by a data dictionary.

=> A data dictionary lists all data items that appear in DFD.

Eg: grossPay = regularPay + overtimePay

Page No.
Date: / /

⇒ The dictionary plays a very important role in any software development process for the following reasons :-

- A data dictionary provides a std. terminology for all relevant data for use by developers working in a project.

- It helps the developers to determine the definition of diff. data structures in terms of their component elements while implementing the design.

- It helps to perform impact analysis i.e. it is possible to determine the effect of some data on various processing activities & vice-versa.

Data Definition

+ : composition of 2 data items.
  Eg. a + b

[, ,] : selection i.e. any one of the data items listed inside the bracket can occur.
  Eg. [a, b] → either a or b occurs

( ) : represents optimal data which may or may not appear.
  Eg. a + (b) → either a OR a+b

Page No.
Date: / /

{} :- represent iterative data definition.
Eg. {name}5 → 5 name data

= :- equivalence
Eg. a = b + c

* * :- comment

Context Diagram

⇒ Most abstract data flow representation of a system.

⇒ Represents the entire system as bubble.

⇒ Also called as the level 0 DFD.

⇒ Establishes the context in which the system operates; i.e. who are the users, what data do they input to the system, & what data they recieve from the system.

Level 1 DFD

⇒ Represent each high level func^n as bubble

Represent data input to every high level func^n.

Represent data output from every high level func^n.

Page No.
Date: / /

High level DFD

=> Each high level func^n is separately decomposed into sub func^{n}s :-

- Identify the sub func^{n}s of the func^n
- Identify the data input to each sub func^n.
- Identify the data output from each sub func^n

Decomposition

=> Also known as factoring OR exploding a bubble.

=> Each bubble is decomposed into 3-7 bubble.

=> It should be carried on until a level is reached at which the func^n of a bubble can be described using simple algorithm.

Numbering of Bubbles

=> Helps in uniquely identifying any bubble from its bubble no.

At context level → 0

At level 1 → 0.1, 0.2, 0.3...

Bubble numbered x → $x.1$, $x.2$, $x.3$, ...

Balancing DFDs

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is called a balancing a DFD.

RMS Calculating Software

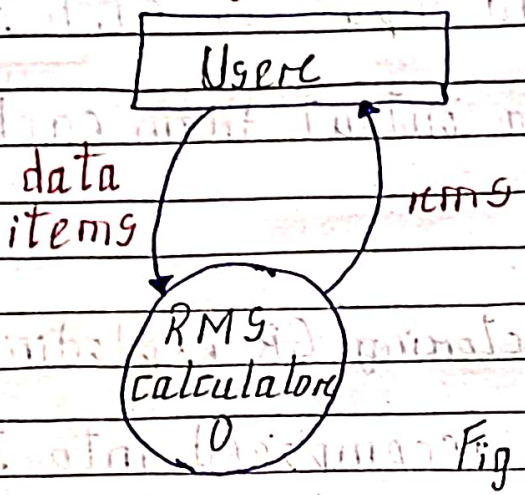


Fig. Context Diagram

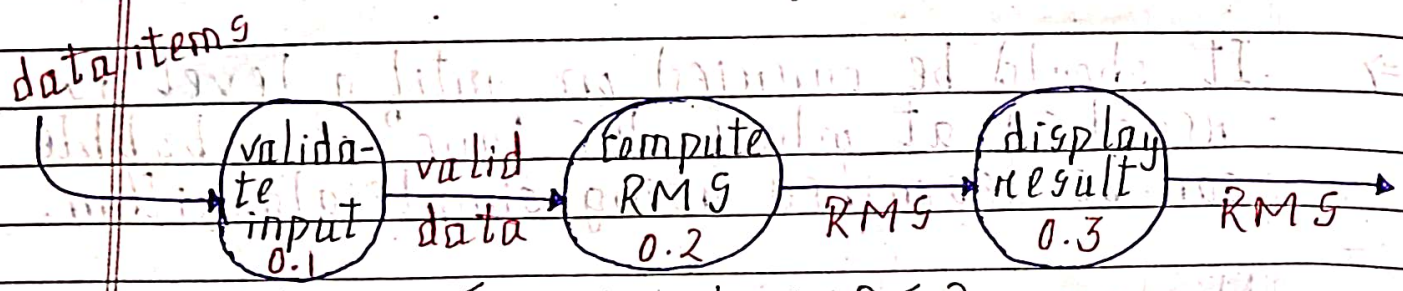


Fig. Level 1 DFD

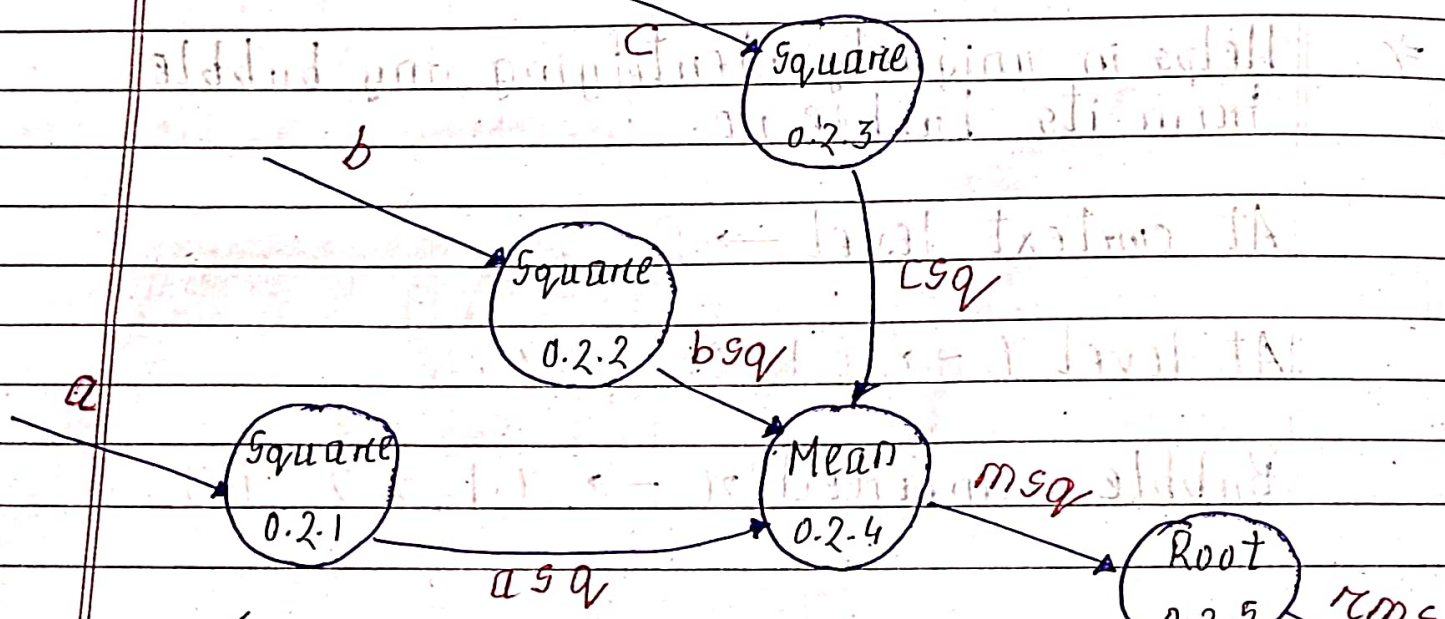


Fig. Level 2 DFD

Tic-Tac-Toe Computer Game

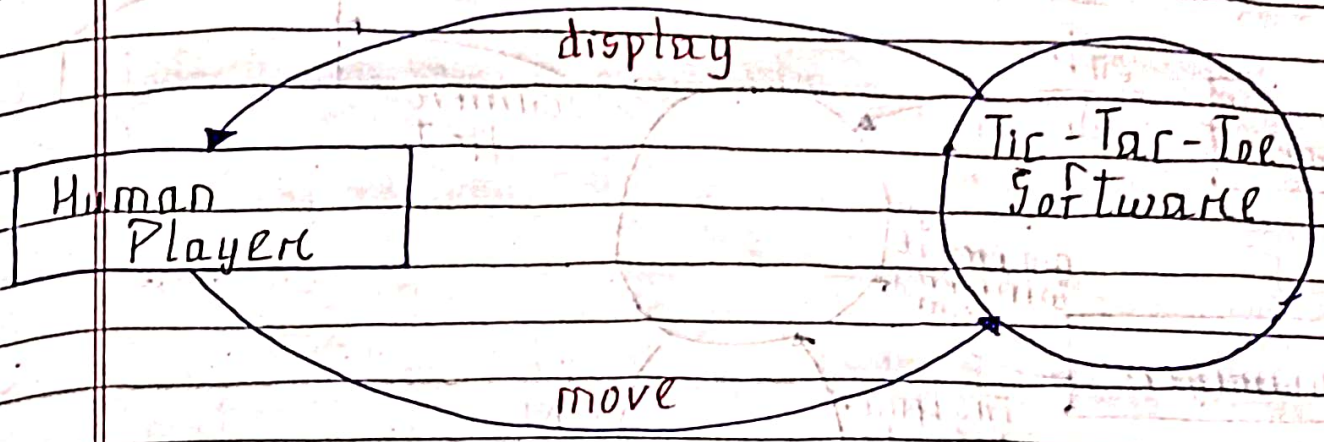


Fig. Context Diagram

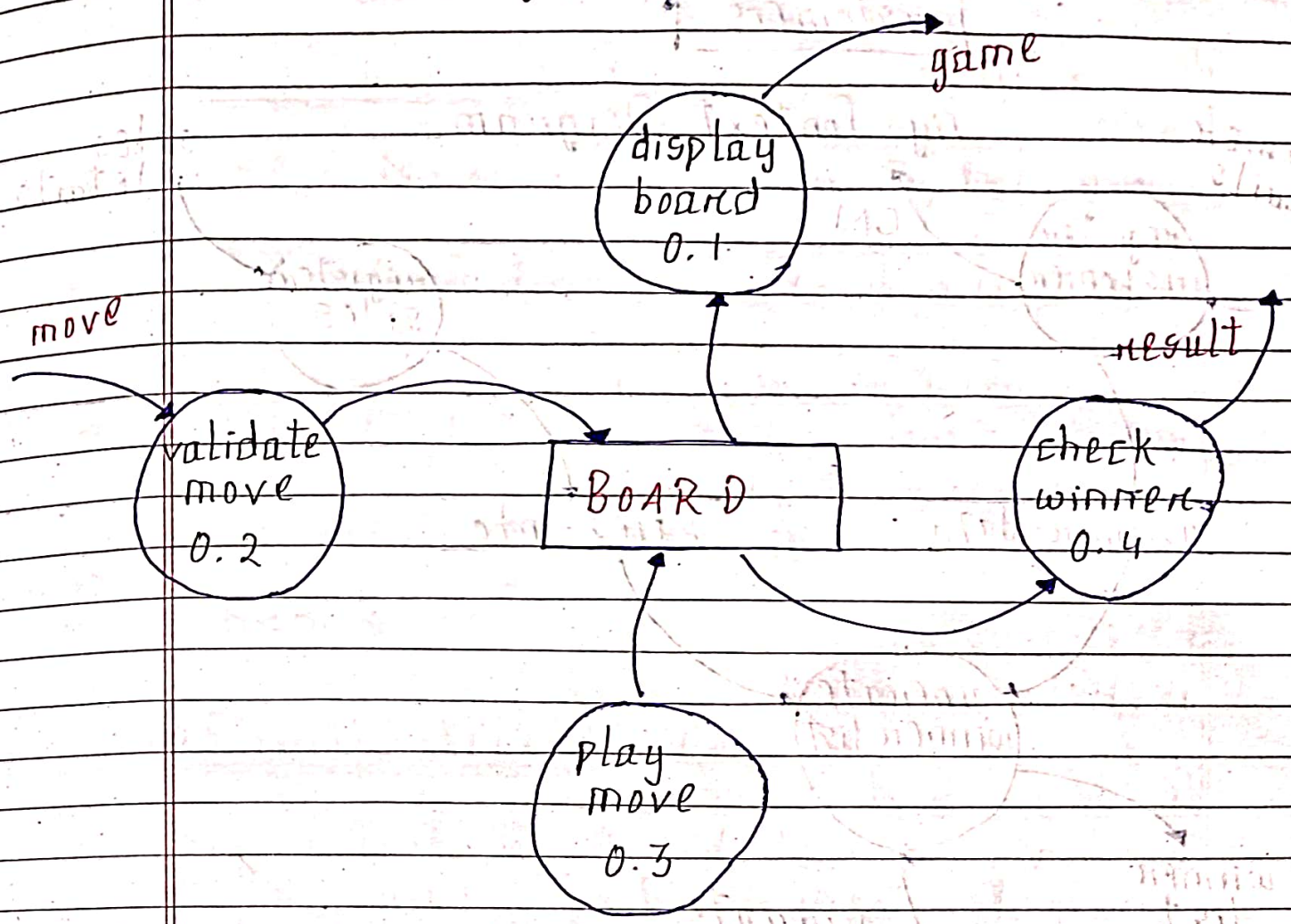


Fig. level 1 DFD

Supermarket Prize Scheme

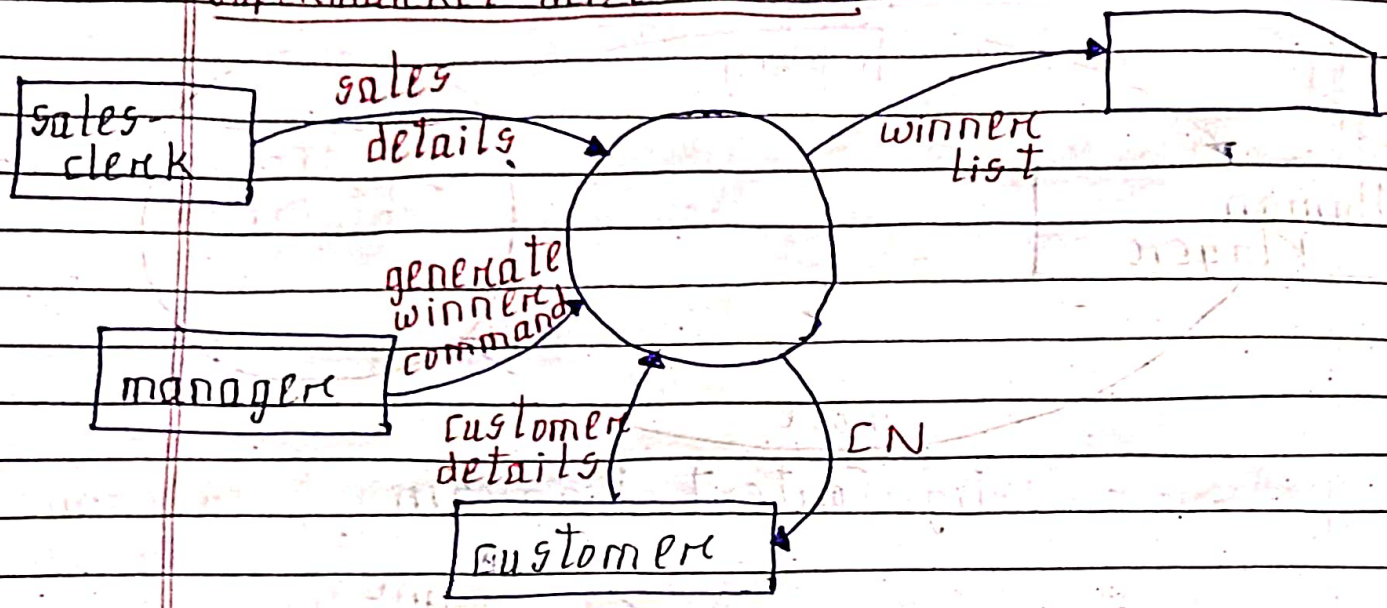


Fig. Context Diagram

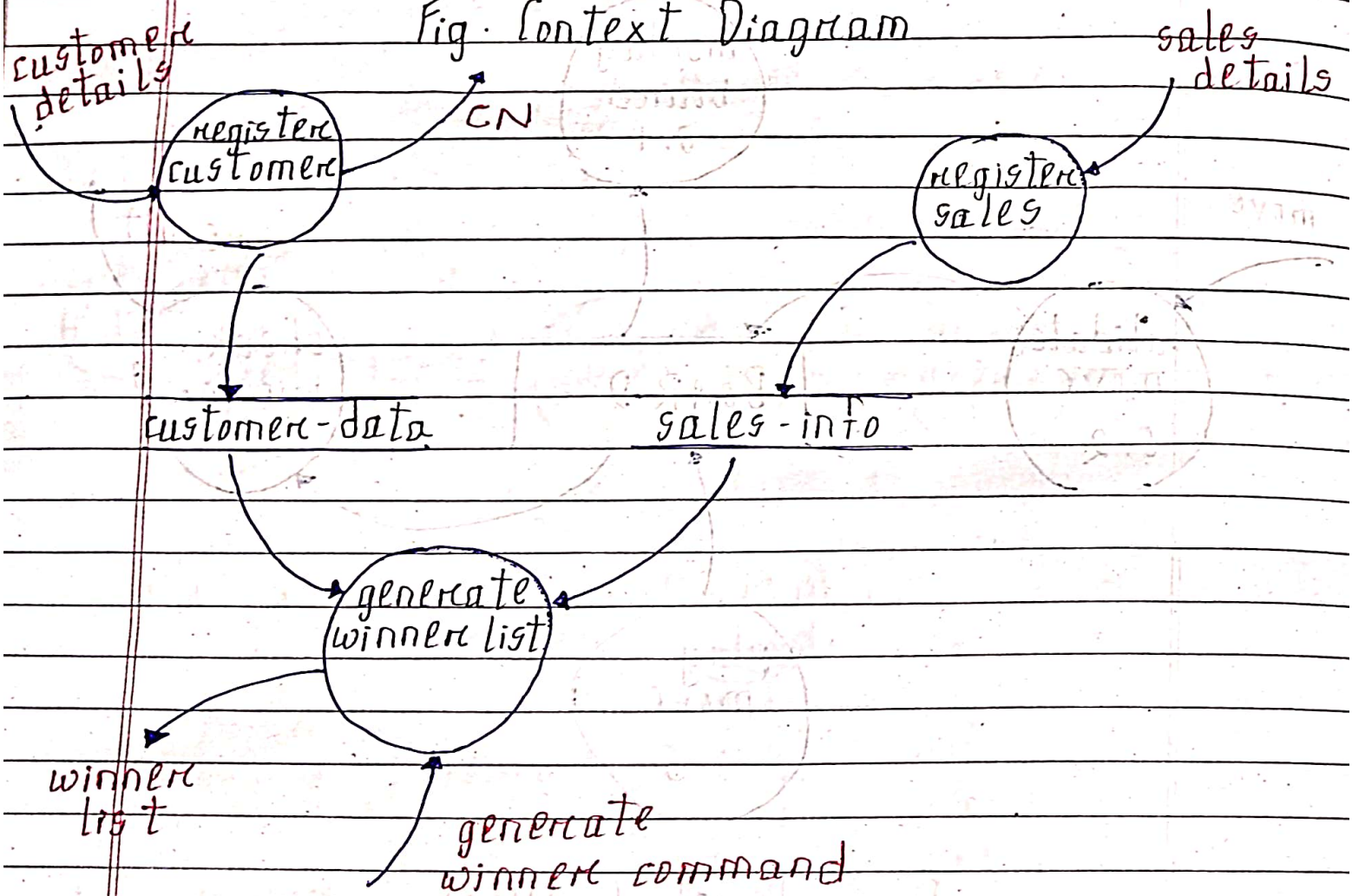


Fig. level 1 DFD

Page No.
Date: / /

Errors while constructing DFDs

=> More than one bubble in the context diagram.

=> External entities appear at higher levels leaving the DFD's unbalanced.

=> Not constructing the data dictionary for the DFDs.

Shortcomings of DFD Model

=> DFD's leave ample scope to be precise.

=> Many DFD diagram for a single requirement specification possible.

=> Difficult to differentiate betn 2 DFD's in terms of which is better.

=> Control aspects are not defined by the DFD.

Structured Design

=> The aim of structured design is to transform the result of DFD model into a structure chart.

Page No.
Date: / /

⇒ Represents the software architecture.

⇒ Easily implementable using programming languages.

Basic Building Blocks

1) Rectangular box

⇒ Represents a module.

2) Module Invocation arrows

⇒ During execution, control is passed from one module to the other in the dirⁿ of the arrow.

3) Data Flow Arrows

⇒ Small arrows appearing along side the module invocation arrows.

⇒ Data passes from one module to the other in the dirⁿ of the arrow.

4) Library Modules

⇒ Represented by a rectangle with double edges.

⇒ Simplifies drawing when a module is called by several modules.

5) Selection

=> The diamond symbol represents one module or several module connected to the symbol is invoked depending on some cond^n.

6) Repetition

=> A loop around control flow arrows denotes that the concerned modules are invoked repeatedly

NOTE There is only one module at the top → root module.

There is at most one control relationship bet^n any 2 modules :-
if module A invokes module B, module B cannot invoke module A.

The main reason behind this restriction is that we can consider the modules in a structure chart to be arranged in layers or levels

Flow Chart Vs Structure Chart

=> It is difficult to identify modules of a software from its flow chart representation

Page No.
Date: / /

⇒ Data interchange among the modules is not represented in a flow chart.

⇒ Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

Steps to generate Structure Chart

⇒ To generate the structure chart level 1 DFD is considered.

⇒ STEP 1 Divide the DFD into 3 parts
a) Input
b) Processing
c) Output

a) The input portion includes the processes which convert input data from physical to logical form.
Each input portion is called Afferent branch.

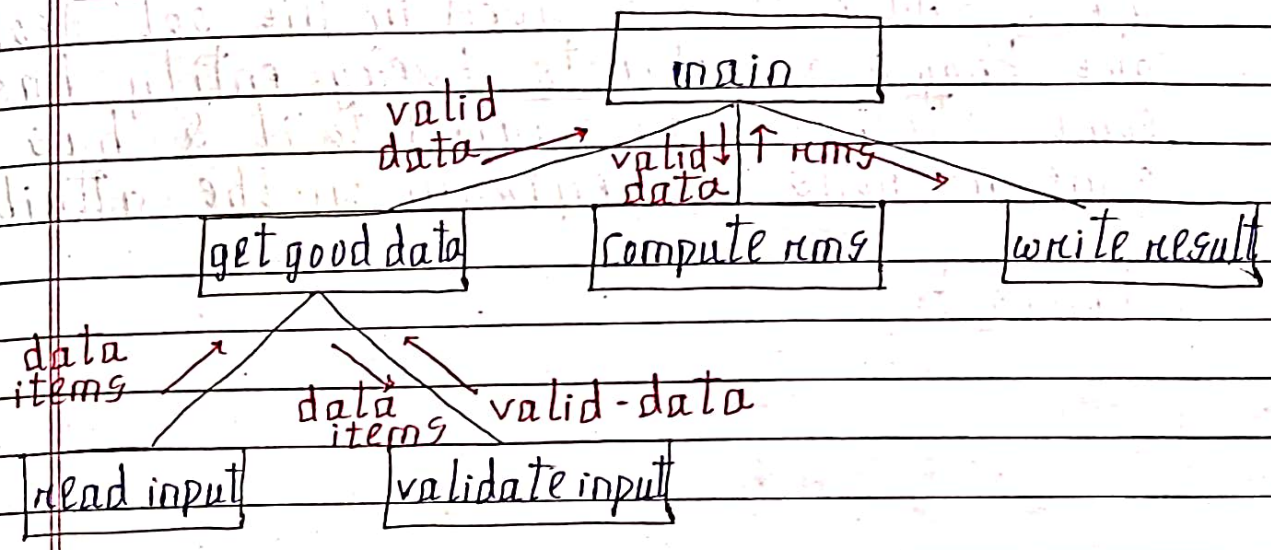
b) The output portion transforms output data from logical to physical form.
Each output portion is called Efferent branch.

) The remaining portions is called central transform.

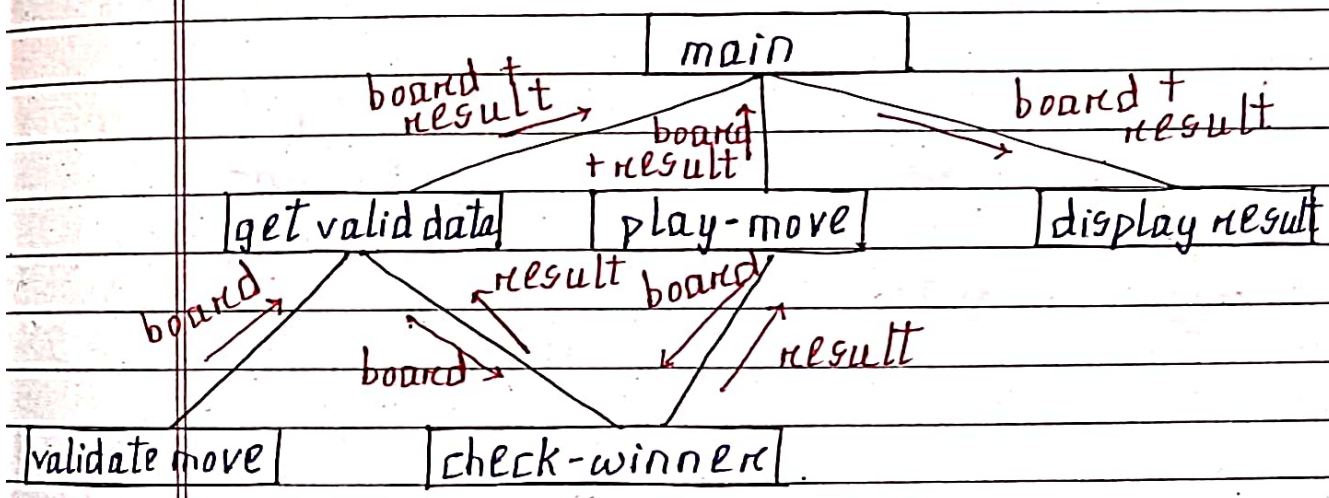
STEP 2 The structure chart is derived by drawing one functional component each for the central transform; the afferent & efferent branch.

STEP 3 The structure chart is refined by adding subfunc^n's required by each of the high-level functional components.

Structure Chart for RMS



Structure Chart for Tic-Tac-Toe



Page No.
Date: / /

OBJECT ORIENTED DESIGN USING UML

OOD (Object Oriented Design)

OOD works around the entities & their characteristics instead of func^n involved in the software system.

Concepts of OOD

1) Objects : All entities involved in the sol^n design are known as objects. Every entity has some attributes associated to it & has some methods to perform on the attributes.

2) Classes : An object is an instance of class. It defines all attributes, which an object can have & methods, which defines the functionality of object.

3) Encapsulation : In OOD, the attributes & methods are bundled together is called encapsulation. It also restrict access of the data & methods from the outside world.

4) Polymorphism : Single interface performing tasks for different types.

5) Inheritance : Sub-classes can import, implement & reuse variables & methods from their immediate super classes.

Page No.
Date: / /

Advantages of OOD

→ Code & design reuse
→ Increased productivity
→ Ease of testing & maintenance
→ Better code & design understandability

Disadvantages of OOD

→ Since the data that is centralised in a procedural implementation, gets scattered across various objects in object-oriented implementation. Thus, the spatial locality of data becomes weak & this leads to higher cache ratios & consequently to larger memory access times.

UML (Unified Modelling Language)

⇒ UML is a language for documenting models.

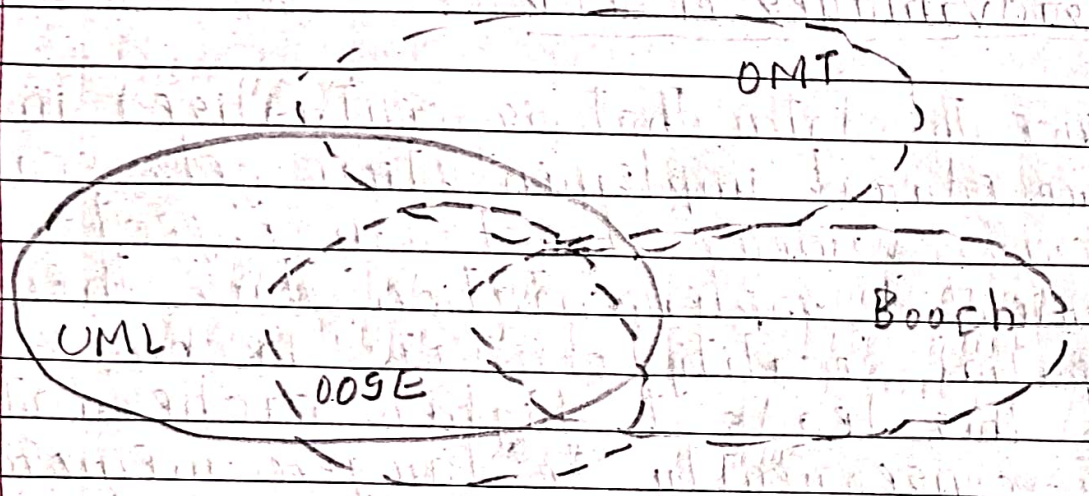
⇒ It is used to document object oriented analysis & design results.

⇒ It is independent of any specific design methodology.

Page No.
Date: / /

Origin of UML

→ OMT (Rumbagh 1991)
→ Booch's Methodology (Booch 1991)
→ OOSE (Jacobson 1992)
→ Odell's Methodology (Odell 1992)
→ Shlaer & Mellor Methodology (Shlaer 1992)



UML as a Standard

⇒ UML was adopted by Object Management Group (OMG) as a de facto std. in 1997.

⇒ OMG aims to promote consensus notations & techniques with the hope that if the usage becomes widespread, then they would automatically become stds.

⇒ ISO adopted UML a std (ISO 19805) in 2005, & with this UML has become an official std.

Page No.
Date: / /

What is Model?

⇒ It is an abstraction of a real problem & is constructed leaving out unnecessary details.

⇒ UML is primarily a graphical modelling tool which is easy to understand & construct.

Why construct a Model?

→ Manage the complexity in a problem.
→ Provides good sol^n & reduce design costs.

UML Diagrams

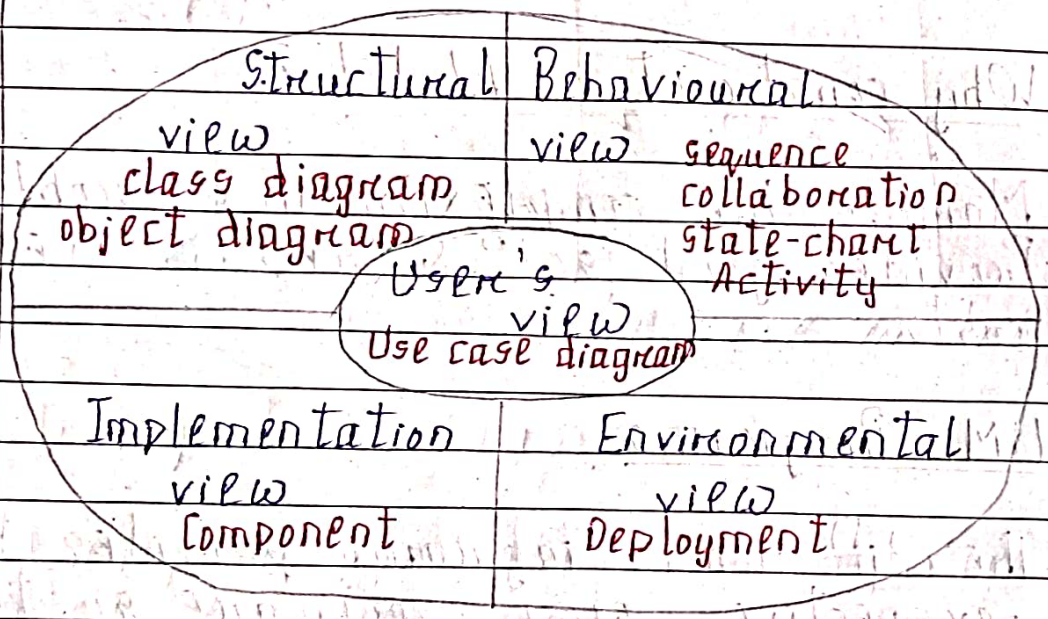
⇒ The diff UML diagrams provides diff. perspectives of a software system to be developed & facilitate comprehensiv understanding of the system.

NOTE If a single model is made to capture all the required perspectives, then it would be as complex as the original problem.

Page No.
Date: / /

UML Model Views

1) User's view
2) Structural view
3) Behavioural view
4) Implementation view
5) Environmental view



Are All views Required for Developing a typical system?

No, it is not required.

Use case & class diagram → interaction diagram for a simple system.

State chart diagram → class state change.

Page No.
Date: / /

Deployment diagram → large components used to develop the system.

Use Case Diagram

⇒ User's view of the system.

⇒ Used to visualise, specify, construct & document the behaviour of the system, during requirement gathering & analysis.

⇒ Contains use cases, actors & their relationships.

Use Cases

⇒ Represent the diff ways in which a system can be used by the users.

⇒ Represent transaction betn the user & the system.

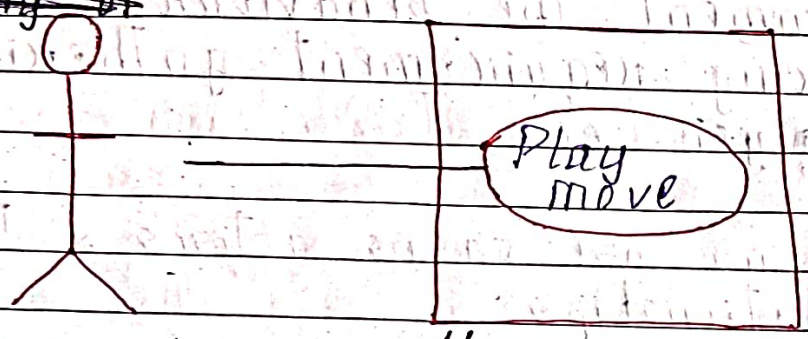
⇒ Involves a sequence of interactions betn the user & the system.

⇒ Defines a piece of coherent behaviour without revealing the internal structure of the system.

Representation of Use Cases

Use case → Ellipse
System boundary → Rectangle
Users → Stick person icons
Communication relationship → Line
External system → << >>

~~Factoring of~~



Use case model

Factoring of Use Cases

Factoring of use cases are required under 2 situations :-

=> Complex use cases need to be factored into simpler use cases.

=> Need to be factored whenever there is common behaviour across diff. use cases.

=> 3 ways :- i) Generalisation
ii) Includes
iii) Extends

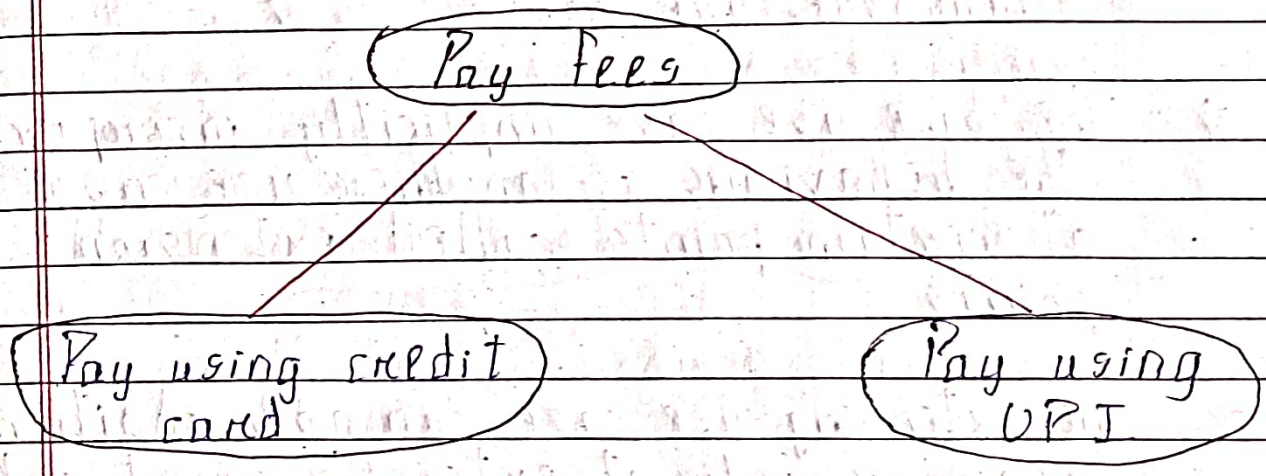
Page No.
Date: / /

i) Generalization

=> Can be used when you have one use case that is similar to another, but does something slightly different.

=> The child use case inherits the behaviour & meaning of the present use case.

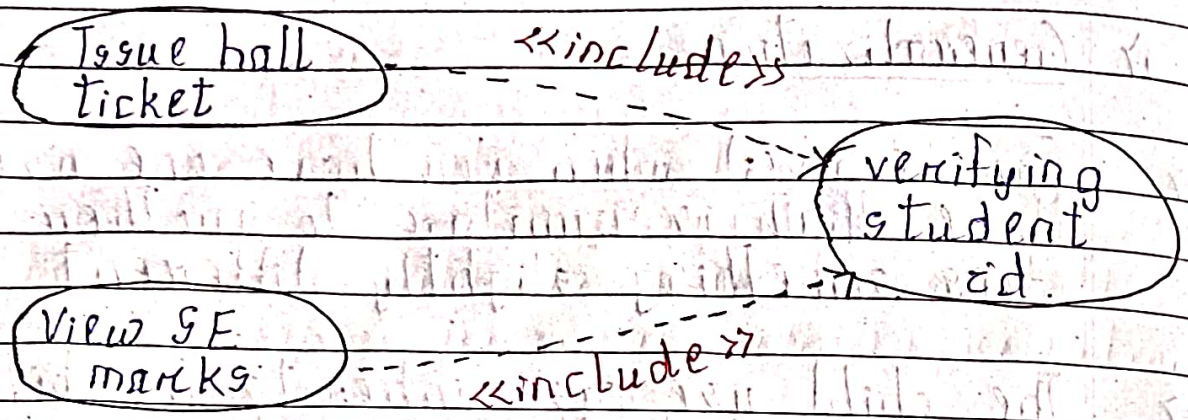
=> The child may add to or override the behaviour of its parent.



ii) Includes

=> Implies one use case includes the behaviour of another use case in its sequence of events & actions.

=> Explores the issues of reuse by factoring out the commonality ~~of~~ across use cases.

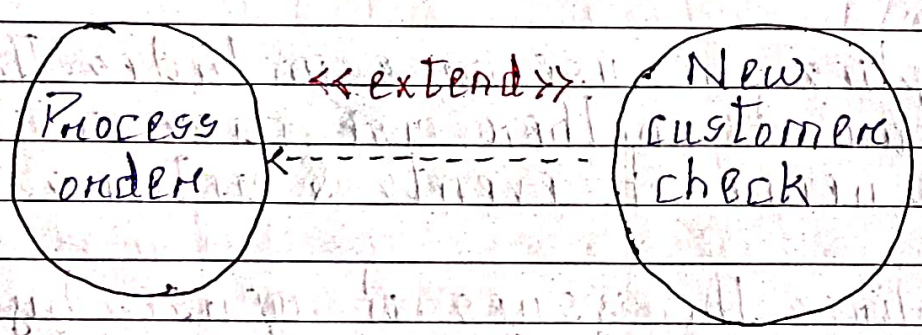


iii) <u>Extends</u>

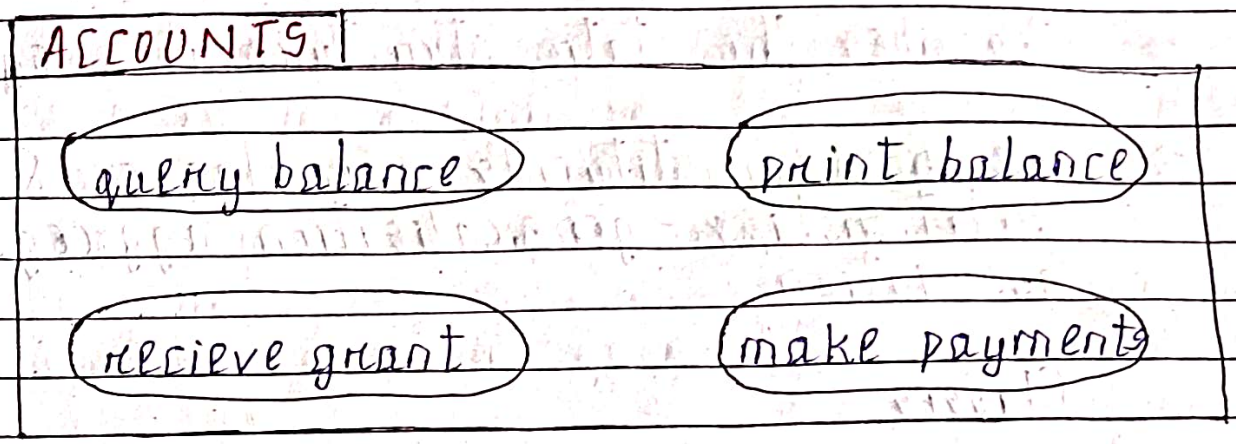
⇒ Use cases that extends the behaviour of other some use cases.

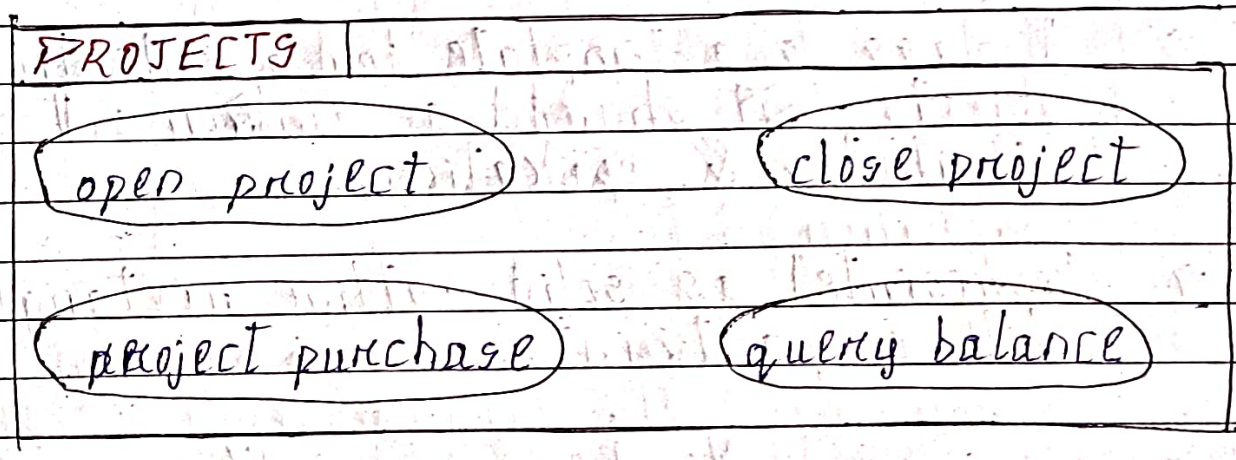
⇒ The base use case implicitly incorporates the behaviour of another use case at certain points called extension points.

⇒ The extending use case can add additional behaviour only at extension point only when certain cond^n^s are satisfied.

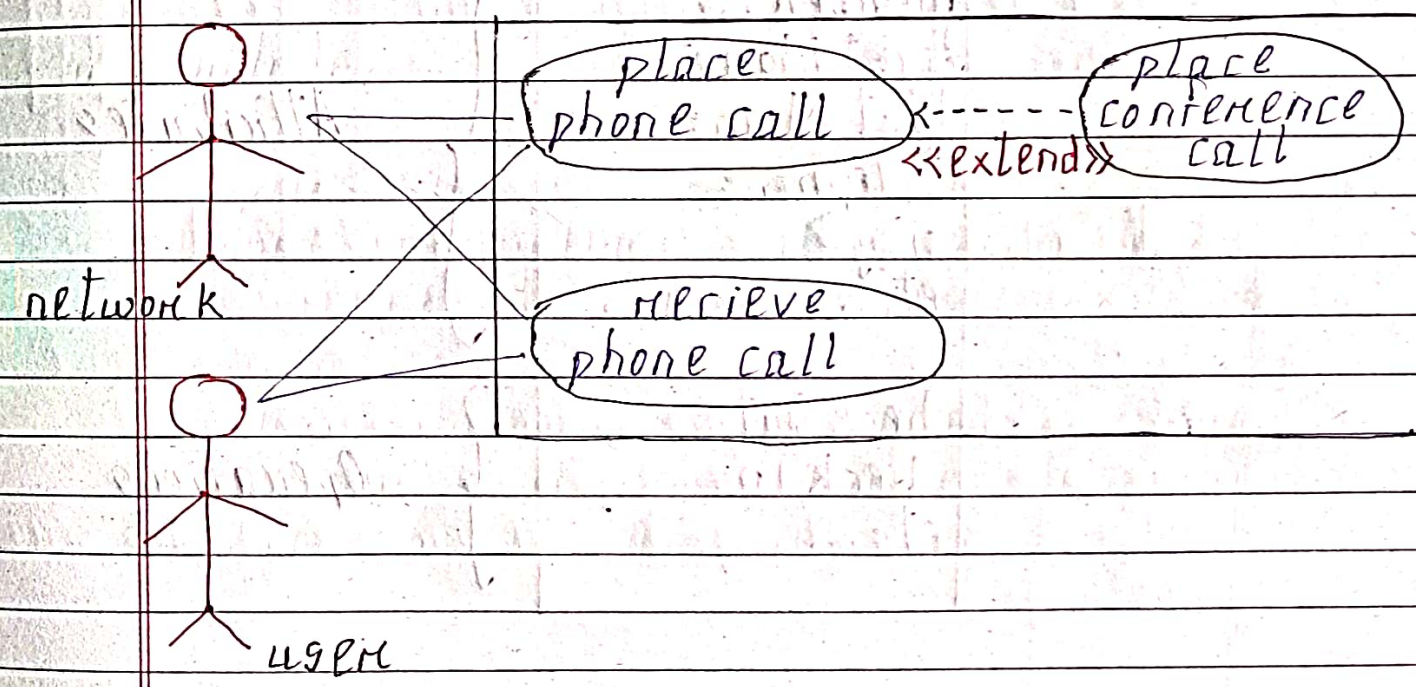


Use Case Packaging





Eg. of Use Case



Class Diagram

⇒ Describes the static structure of a system.

⇒ The main constituents are classes & their relationships - generalisation, aggregation, association.

Classes

⇒ A class is a template for a collection of objects that shared a common set of attributes & operations.

⇒ Represented as solid outline rectangles with compartments.

Member	Class name
memberNo First Name Last Name telephone city etc...	Attributes
checkOut checkIn etc...	Operations

Association

=> Association betn 2 classes is represented by drawing a st. line betn the concerned classes.

=> The name of the association is written along side the association line.

=>

Library Member	1	◄ Borrowed by	* Book

=> On each side of the association relation, the multiplicity is noted as an individual no. or as a value range.

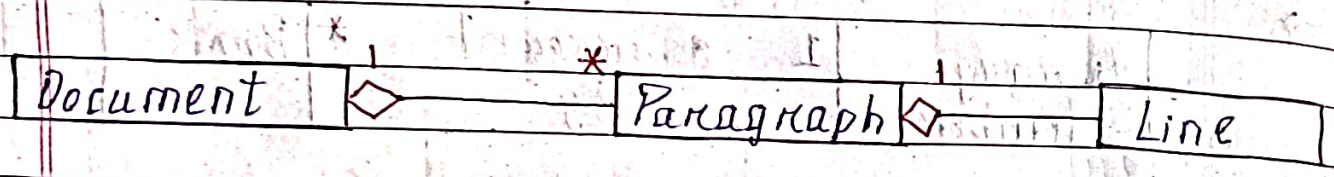
=> The multiplicity indicates how many instances of one class are associated with the other.

Exactly one → 1
Zero or more → 0... *
One or more → 1... *
Zero or one → 0... 1
Specified range → 2... 4

Aggregation

⇒ It is a special type of association relation where the involved classes are not only associated with each other, but a whole-part relationship exist betⁿ them.

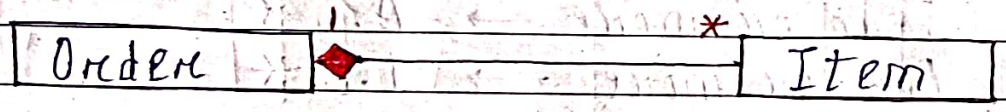
⇒ Represented by an empty diamond at the aggregate end of a relationship.



Composition

⇒ It is a stricter form of aggregation, in which the parts are existence-dependent on the whole.

⇒ Represented as a filled diamond drawn at the composite-end.



In this case, as soon as the order object is created, all the other items in it are created. & as soon as the order object is destroyed, all order items are destroyed.

Page No.
Date: / /

Aggregation Vs Composition

=> When the components can dynamically added & removed from the aggregate. Then the relationship is aggregation.

=> If the components cannot be dynamically added / delete then the relationship is composition.

Interaction Diagram

=> These are models that describe how groups of objects interact among themselves through message passing to realise some behaviour.

=> Typically, each interaction diagram realises the behaviour of a single use case.

=> Sequence diagram is an example of interaction diagram.

Sequence Diagram

=> Shows interaction among objects as a 2D chart.

Objects → boxes at top

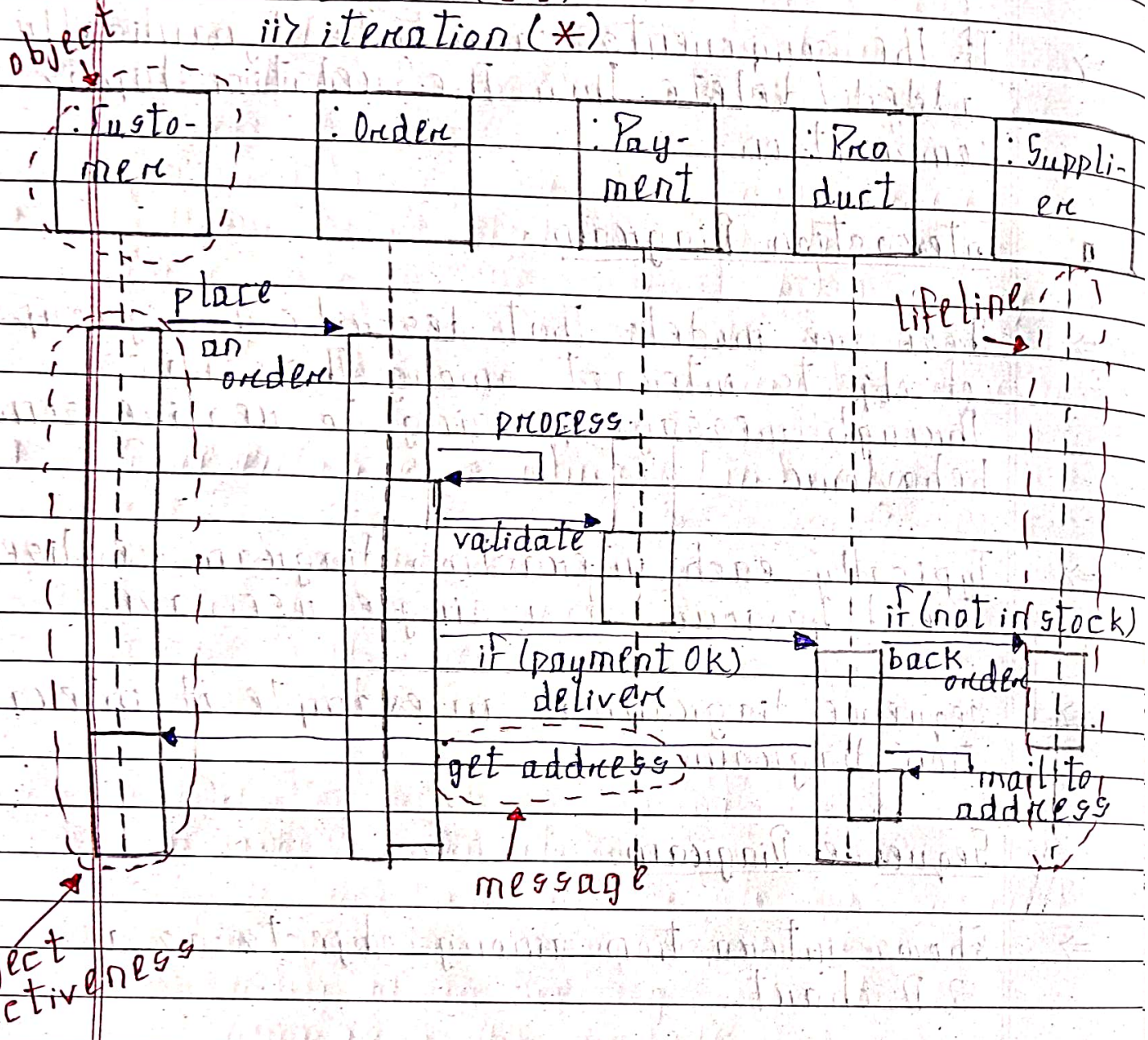
Objects existence → dashed lines

Page No.
Date: / /

Object activeness → rectangle

Messages → arrows

Two types of control information :-
i) cond^n ([ ])
ii) iteration (*)



UNIT - III

Page No.
Date: / /

CODING

=> Coding is undertaken once the design phase is complete & the design documents have been successfully reviewed.

=> The objective of the coding phase is to transform the design of a system into code in a high-level language.

Coding Standards

=> Good software development organisations require their programmers to adhere to some well-defined & std style of coding which is called their coding stds.

=> These software development organisations formulate their own coding stds that suit them the most & require their developers to follow the stds rigorously because of the significant business advantage it offers.

Advantages

=> Gives uniform appearance to the codes written by diff. engineers.

=> Promotes good programming practices.

=> Facilitates code understanding & code reuse.

Page No.
Date: / /

- **Difference betn Coding Guideline & Coding Standard**

⇒ Coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

⇒ It is mandatory for the programmers to follow the coding stds. Compliance of their code to the coding stds is rejecte during ~~coding~~ code review & the code is reworked by the concerned programme

- **Representative ~~of~~ Coding Standards**

⇒ Naming conventions for
  a) global variables
  b) local variables
  c) const identifiers.

⇒ Contents of headers of diff modules
  a) name of the module
  b) date on which the module was created.
  c) author's name
  d) modification history
  e) synopsis of the module

Page No.
Date: / /

Coding WalkThrough

⇒ It is an informal code analysis technique.

⇒ The main objective is to discover the algorithmic & logical errors in the code.

⇒ Governed by the developer's experience & ability.

⇒ The purpose should be error discovery & not how to fix errors.

Code Inspection

⇒ Its purpose is to find common made errors by going through the code.

⇒ Good companies record all the common errors made by a developer which becomes a checklist for future.

⇒ Following is a list of some classical programming errors which can be checked during code inspection :-

→ Use of uninitialized variables.
→ Jumps into loops.
→ Non-Terminating loops.
→ Array indices out of bounds.

Page No.
Date: / /

→ Incompatible assignments
→ Improper storage allocation & deallocation.
→ Mismatches betn actual & formal parameter in procedure calls.
→ Use of incorrect logical & precedence operators.
→ Improper modification of loop variables.
→ Comparison of equally of floating point variables.

- **Software Documentation**

⇒ When a software is developed, in addn to the executable files & the source code, several kinds of documents such as User's manual, SRS document, text document, etc. are developed as part of the software engineering process.

It is classified into 2 categories :-
i) Internal
ii) External

i) **External Documentation**

⇒ It is the code comprehension features provided in the source code itself.

⇒ Following are the types of internal documentation :-

→ Comments embedded in the source code
→ Code structuring
→ Code indentation
→ Meaningful variable names
→ Const. identifiers
→ User-defined data types

ii) External Documentation

⇒ Provided through various types of supporting documents such as-
a) user's manual
b) SRS document
c) Design document
d) Test document

⇒ Good external documentation is consistency with the code.

■ Gunning's Fox Index

→ no. of years of schooling to understand

$$F = 0.4 * \frac{\text{No. of words}}{\text{No. of sentences}} + \%\text{ of words of 3 or more syllables}$$

NOTE A document is easy to understood if
→ all sentences are small
→ use only 4-5 words each
→ not exceeding 5-6 characters per word

Page No.
Date: / /

TESTING

How To Test a Program ?

1) Input test data to the program.

2) Observe the output check if the program behave as expected.

3) If the program doesn't behave as expected note the cond^{n}s under which it failed for further debugging & correction.

Terminologies

1) Mistake : It is a programmer action that later shows up as an incorrect result during program execution.

2) Error : Result of mistake committed by a developer in any of the development activities.

3) Failure : Denotes an incorrect behaviour exhibited by the program during its execution.

4) Test Case : A test case is a triplet [I, S, O] where

I → data input to the program

S → state of the program at which the data is to be input.

O → Expected Output produced by the program

5) Test Suite : Set of all test cases that have been designed to test a given program

NOTE A test case is said to be a +ve test case if it is designed to test whether the software correctly performs a required functionality.

A test case is said to be a -ve test case if it is designed to test whether the software carries out something that is not required by the system.

Verification Vs Validation

⇒ Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas Validation is the process of determining whether the fully developed software conforms to its requirement specification.

⇒ Verification does not require execution of the software whereas validation requires execution of software.

⇒ Verification is concerned with phase containment of errors whereas Validation aims

to check whether the deliverable software is error free.

Error Detection Techniques = Verification Techniques + Validation Techniques

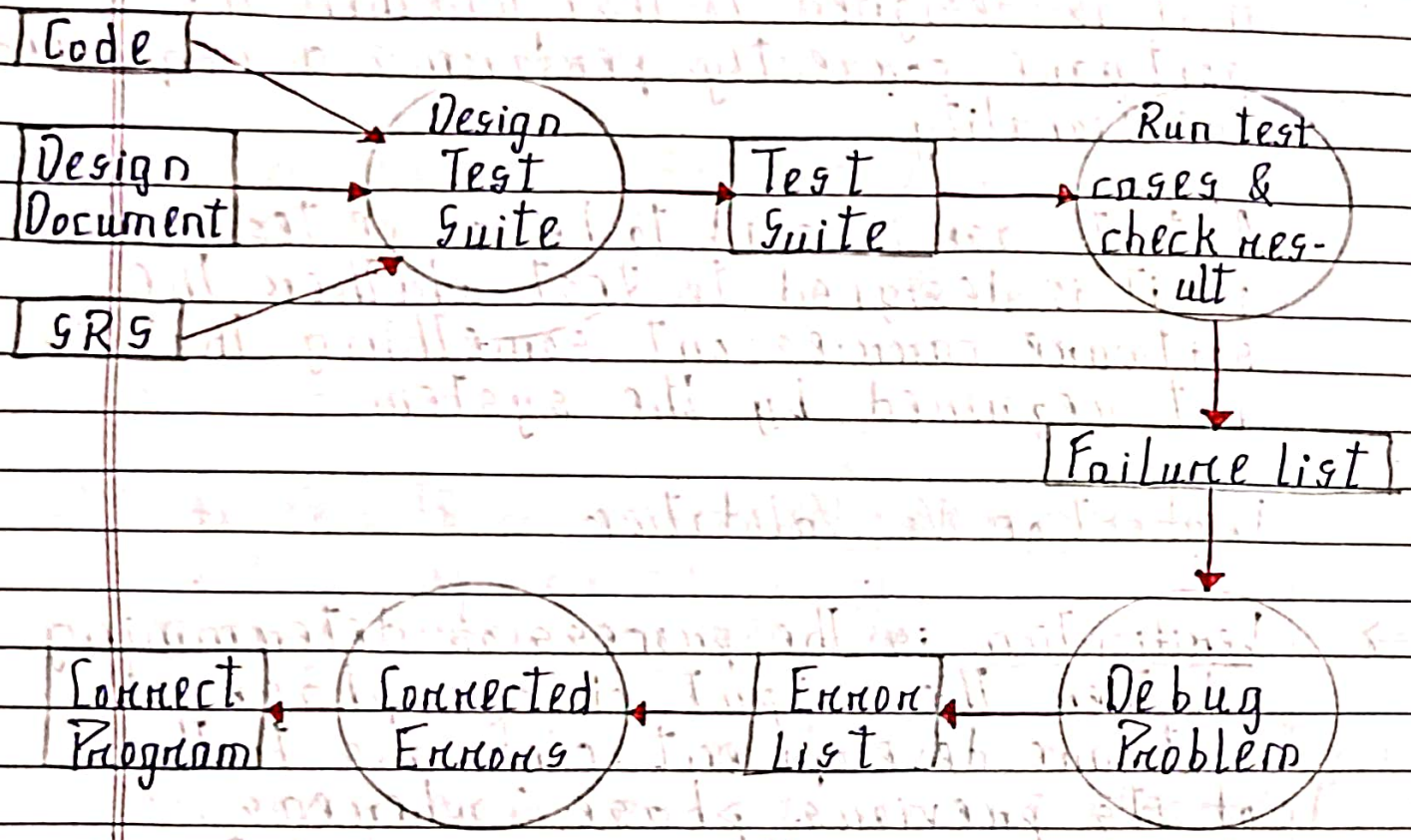


Fig. Testing Process

Why Design Test Cases?

=> When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite.

Page No.
Date: / /

=> A minimal test suite is carefully designed set of test cases ~~that~~ such that each test case helps detect different errors.

Black Box Testing

=> It is concerned with the software's internal knowledge or implementation details but rather focuses on validating the functionality based on the provided requirements.

OR

Test cases are designed using only functional specification of the software and without any ~~internal~~ knowledge of internal structure of the software.

=> Also known as functional testing

Black Box Testing Approaches

1) Equivalence Class Partitioning :

=> In this approach, the input values to a program are partitioned into equivalence classes.

=> The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

Page No.
Date: / /

Guidelines for designing the equivalence classes :-

1) If the input data values can be specified by a range of values, then one valid & 2 invalid equivalence classes can be defined.

2) If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values & another equivalence class for the invalid ~~output~~ input values should be defined.

2) Boundary Value Analysis :-

⇒ Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the diff. equivalence classes.

⇒ To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values.

Eg. Consider a program unit that takes an input integer values in range 0-5000 & compute its square root. Determine the equivalence classes & the black box test suite for the program unit.

Page No.
Date: / /

A- 3 set of equivalence classes :-
$[-\infty, 0]$, $[0, 5000]$, $[5000, \infty]$

So, The possible test suite is -
$\{-5, 500, 6000\}$

Determine the boundary test suite.

A- $\{0, -1, 5000, 50001\}$

White Box Testing

=> Test cases ~~where~~ which are designed using the knowledge about the internal structure of the software.

=> Also called structural testing

Diffrence betn Black Box & White Box Testing

Page No.
Date: / /

Debugging

=> After a failure has been detected, it is necessary to first identify the program statement that are in error & are responsible for the failure. The error can then be fixed.

Debugging Methods

1) Brute Force Method

=> Least effective method.

=> Print statements are inserted throughout the program to print the intermediate values in a hope that some of the values will help to identify the error.

=> Becomes more systematic with the use of a symbolic debugger.

=> Using a symbolic debugger, values of diff. variables can be easily checked & modified

=> Single stepping is used to execute one inst^n at a time.

Page No.
Date: / /

2) Backtracking

=> Fairly common approach.

=> Starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.

=> Unfortunately, in the presence of decision statements & loops, the no. of source lines to be traced back increases & the no. of potential backward paths increases which becomes unmanageably ~~complex~~ large for complex programs.

3) Cause Elimination Method

=> Once a failure is observed, the symptoms of the failure are noted.

=> Based on the failure symptoms, the causes which could possibly have contributed to the symptom is identified & tests are conducted to eliminate each.

4) Problem Slicing

=> Similar to backtracking.

=> A slice is defined for a particular variable

Page No.
Date: / /

at a particular statement is the set of source lines preceding this statement which can influence the value of the variable.

Types of Testing

1) Unit Testing

=> It means testing each module of a program separately in isolation.

=> It makes debugging easier by reducing the effort needed to find & fix errors.

=> Programmers usually do unit testing immediately after they finish coding a module.

2) Integration Testing

=> It is carried out after all the modules have been unit tested.

=> The objective is to detect the errors at the module interface.

=> At each integration step, the partially integrated system is tested.

=> It has 4 approach to perform the testing :-

Page No.
Date: / /

i) Big Bang Approach

=> The unit tested modules of the system are simply linked together & tested

=> Can be meaningfully used for very small systems.

Disadvantage :-

→ Once an error is detected, it is very difficult to localise the error.

→ Debugging errors are very expensive to fix.

ii) Bottom Up Approach

=> The bottom-level modules are integrated & tested.

Advantage :-

→ Disjoint subsystems can be tested simultaneously.

→ Low level modules get tested thoroughly.

Disadvantage :-

The system has large no. of small subsystems at same level which complex the testing.

Page No.
Date: / /

iii) Top-Down Approach

=> Starts with the root module in the structure chart & one or 2 subordinate modules of the root module.

=> After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it & tested.

=> Testing waits till all top-level modules are coded & unit tested.

iv) Mixed Approach

=> Comb^n of top-down & bottom-up approach.

=> Testing can start as & when modules become available after unit testing.

=> Most commonly used approach.

3) System Testing

=> Designed to validate a fully developed system to assure that it meets its requirements.

=> The testing cases are designed based on the SRS document.

Page No.
Date: / /

It consists of 3 kinds :-

a) Alpha Testing : System testing carried out by the test team within the developing organisation.

b) Beta Testing : System testing performed by a select group of friendly customers.

c) Acceptance Testing : System testing performed by the customer to determine whether the system should be accept or not.

Stress Testing

=> Impose a range of abnormal & even illegal input cond^ns so as to stress the capabilities of the software.

=> Input data volume, input data rate, processing time, utilisation of memory etc. are tested beyond the designed capacity.

Page No.
Date: / /

MAINTENANCE, RELIABILITY, QUALITY

Maintenance

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

=> It is becoming an important activity of a large no. of organisations.

Types of Software Maintenance

1) Corrective Maintenance
It is necessary to overcome the failures observed while system is in use

2) Adaptive Maintenance
When the customers need the product to run on new platforms, on new OS or when the product need to interface with new H/W or S/W.

3) Perfective Maintenance
Support adding new features & to change diff. functionalities of the system.

Page No.
Date: / /

Lehman's Laws

1st law : A software product must change continously otherwise they will become progressively less useful.

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

3rd law : The rate at which code is written or modified is approximately the same during development & maintenance

Software Maintenance Process Models

METHOD 1

=> This method is suitable when the changes required in the software are minimal & not complex in nature

=> The updates might include fixing minor bugs, updating a user interface element or making small adjustments to improve usability.

=> Do not require deep analysis or major modifications to the software's functionality.

Page No.
Date: / /

Gather change Requirements
↓
Analyse change Requirements
↓
Devise code change strategies
↓
Apply code changes to old code
↙ ↘
Update documents | Integrate & test

METHOD 2

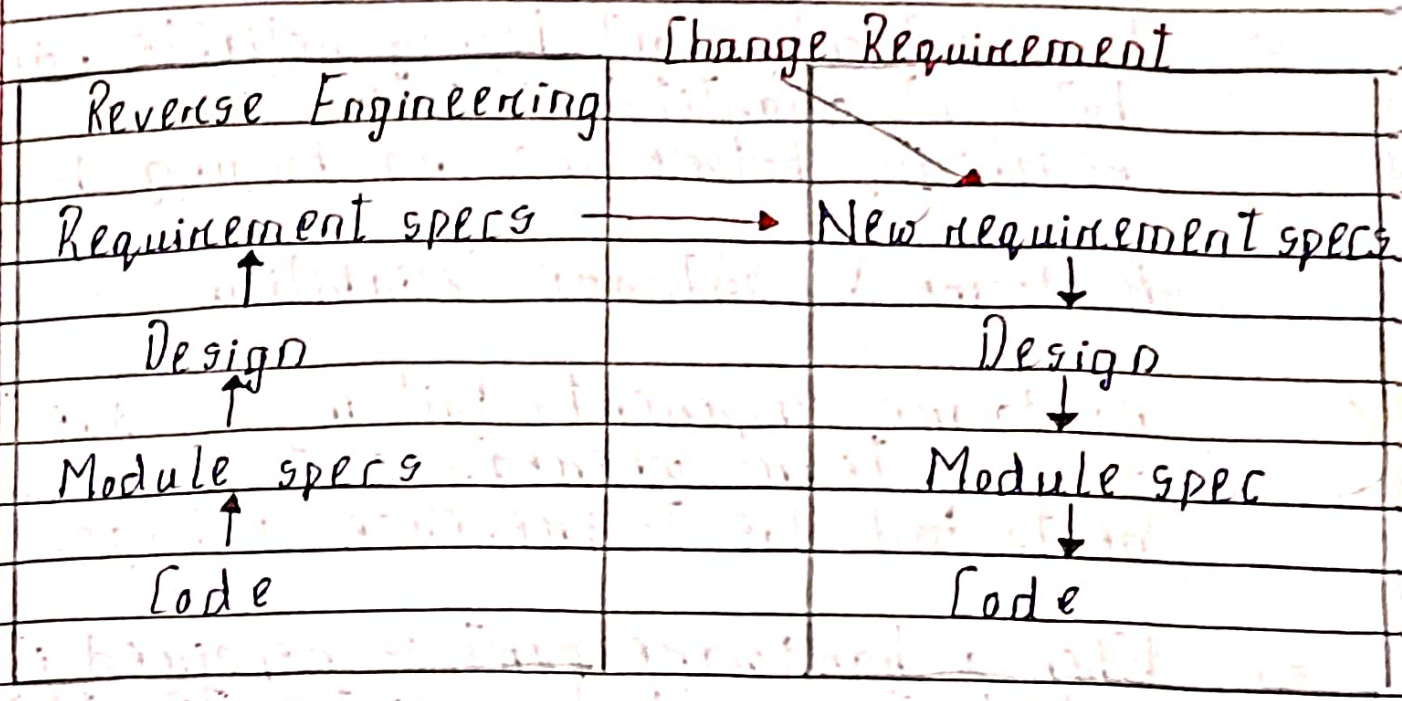
⇒ Used when the software is either outdated or poorly documented.

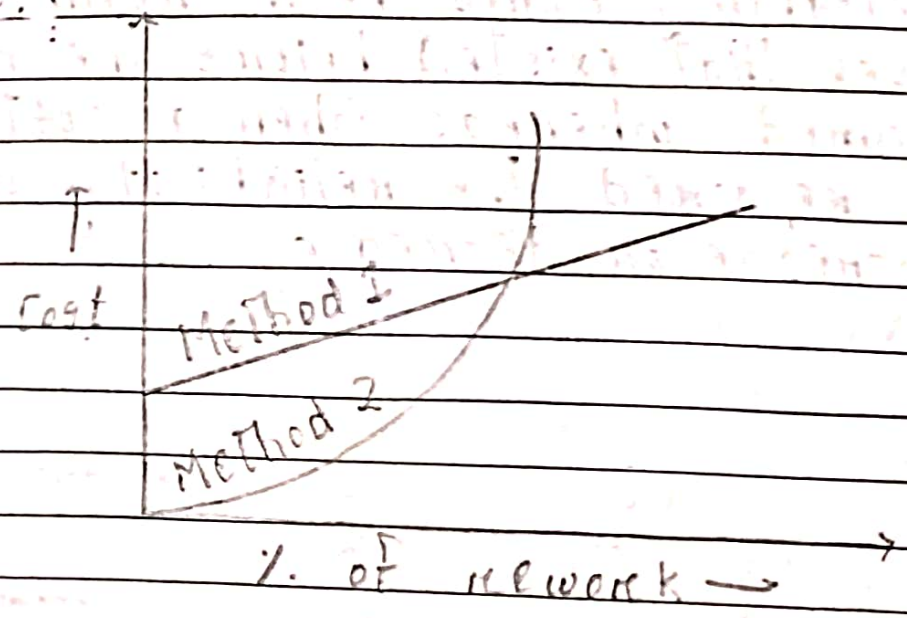
⇒ Changes may involve major redesigns, new functionalities or integration with other systems.

⇒ Original system may not have proper documentation.

⇒ Reverse Engineering : Analysing the existing system to discover its components & how they interact.

Forward Engineering : Used to redesign or improve the system.





Reliability

⇒ Denotes the trustworthiness or dependability of a software.

⇒ Can be also defined as the probability of the product working 'correctly' over a given period of time.

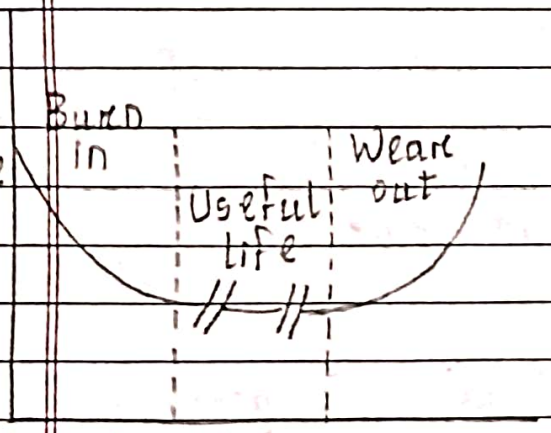
Page No.
Date: / /

=> Users not only want highly reliable products but want quantitative estimation of reliability before making buying decision.

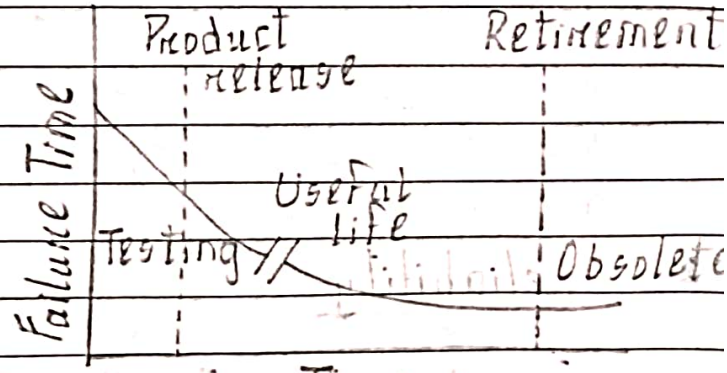
Hardware Vs Software Reliability

=> Hardware components fail mostly due to wear & tear, whereas software components fail due to presence of bugs.

=> When a hardware part is repaired its reliability would be maintained at the level that existed before the failure occured, whereas when a software failure is repaired, the reliability may either increase or decrease.



a) Hardware Product



b.) Software Product

Page No.
Date: / /

Reliability Metrices

1) Rate of occurrence of failure (ROCOF)

=> Measures the frequency of occurrence of failures.

=> Measured by dividing the total no. of failures observed by the duration of observation.

2) Mean Time to Failure (MTTF)

=> It is the time betn 2 successive failures avg. over a large no. of failures.

Let the failures occur at the time instants $t_1, t_2, \ldots, t_n$.

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

=> Only ~~tim~~ run time is considered in the time measurements.

3) Mean time to Repair (MTTR)

=> Once the failure occurs, it tracks avg. time to track the errors causing the failure & fix them.

Page No.
Date: / /

4) Mean time betn failure (MTBF)

$$MTBF = MTTF + MTTR$$

MTBF of 300 hrs indicates that once the failure occurs, the next failure is expected after 300 hrs.

5) Probability of failure on demand (POFOD)

=> Measures the likelihood of the system failing when a service request is made.

POFOD of 0.001 mean that 1 out of every 1000 service requests would result in failure.

6) Availability

=> It is a measure of how likely would the system be available for use ~~for a~~ over a given period of time.

=> Important for systems such as telecommunication systems, OS, etc. which are supposed to be never down.

Page No.
Date: / /

Quality

=> A good quality product does exactly what the users want it to do, since for almost every product, fitness of product is determined in terms of SRS Document.

=> Several quality factors are associated with a software product :-

1) Correctness
2) Reliability
3) Portability
4) Usability
5) Maintainability
6) Reusability

Software Quality Management System

=> It is the methodology used by organisations to ensure that the products they develop have the desired quality.

Quality System Activities :-

→ Auditing of projects
→ Development of stds, procedures & guidelines.
→ Review of the quality system
→ Produce reports for the top management summarising the effectiveness of the quality system of organisation.

Page No.
Date: / /

ISO 9001

=> Specifies requirements for a quality management system where an organisation needs to demonstrate its ability to consistently provide services that meet customer & regulatory requirements.

=> Includes design & development, production, installation & servicing.

ISO 9002

=> Covered quality assurance in production, installation & servicing but excluded design & development.

ISO 9003

=> Covered quality assurance in final inspection & testing only.

=> Applied to organisations that did not do design, production or servicing.

Page No.
Date: / /

SOFTWARE PROJECT MANAGEMENT

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

Responsibility of Project Manager

A project manager's activities can be broadly classified into :-
1) Project Planning
2) Project monitoring & control

1) Project Planning

=> It involves estimating several characteristics of a project & then planning the project activities based on the estimates made.

=> Commitments to unrealistic time & resource estimates result in :-
i) irritating delays
ii) customer dissatisfaction
iii) adverse affect on team morale.
iv) poor quality work.
v) project failure

=> Following are the activities of project planning :-

Page No.
Date: / /

1) Estimation (Cost, Duration, Effort)
2) Project Scheduling
3) Staffing
4) Risk identification (Analysis of risks)
5) Miscellaneous plans (Quality assurance, Configuration managemen

2) Project Monitoring & Control

The focus is to ensure that the software development proceeds as per plan.

SPMP Document

Once the project planning is complete, project managers document their plans in a software project management plan (SPMP) document.

Organisation of SPMP Document

1) Introduction :- a) Objectives
b) Major Func^n^s
c) Performance issues
d) Management & Technical constraints.

2) Project Estimates :- a) Historical data
b) Estimation Technique
c) Effort, cost, duration estimates

Page No.
Date: / /

3) Schedule :- a) Work breakdown structure
b) Task Network
c) Gantt Chart
d) PERT Chart

4) Project Resources :- a) People
b) H/W & S/W
c) Special resources

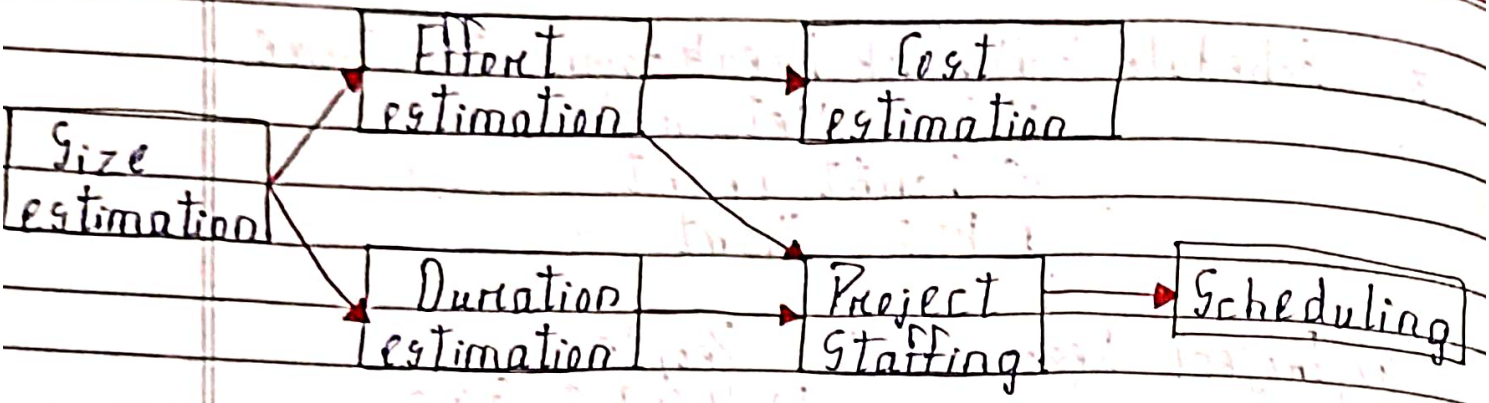
5) Staff organisation :- a) Team structure
b) Management reporting

6) Risk Management :- a) Risk analysis
b) Risk identification
c) Risk estimation
d) Risk Abatement

7) Project Tracking & control :-
a) Metrics to be tracked
b) Tracking plan
c) Control plan

8) Miscellaneous plans :- a) Process Tailoring
b) Quality Assurance

Software Cost Estimation



Software Size Metrices

LOC (Line of Code)

=> Simplest & most widely used metric.

=> Measures the size of a project by counting the no. of source inst^n's in the program & ignoring the comments & blank lines

Disadvantages

→ Size can vary with coding style.
→ Focuses on coding activity alone.
→ Correlates poorly with quality & efficiency.
→ Difficult to estimate LOC from problem description.
→ Does not access the issue of structural or logical complexity.

Function Point (FP) Metric

Based on the idea that a software product supporting many features would

Page No.
Date: / /

certainly be of large size than a product with less no. of features.

=> Overcomes many of the ~~meta~~ disadvantages of LOC.

Page No.
Date: / /

Cost Estimation Techniques

3 Types :- 1) Empirical
2) Heuristic
3) Analytical

1) Empirical Estimation Techniques

=> Based on making an educated guess of the project parameters.

a) Expert Judgement :-

=> Expert makes an educated guess about the problem size.

=> Estimates the cost of diff. components of the product by dividing it & then combines the estimates. to get overall estimate.

=> Disadvantages :- Suffers from human errors & individual bias.

b) Delphi Estimation :-

=> Carried out by a team comprising a group of experts & co-ordinator.

=> Estimators complete their individual estimates anonymously & submit them to the co-ordinator.

Page No.
Date: / /

=> No discussions among the estimators is allowed during this process.

=> The coordinator prepare the summary of all the responses of the estimators & again it is being re-estimated.

COCOMO Model

=> Constructive COst MOdel was proposed by Bohem in 1981.

=> It is a 3 stage process for project estimation
a) Organic (Application)
b) Semidetached (Utility)
c) Embedded (Systems)

=> Application programs :- Data Processing
Scientific Programs

=> Utility programs :- Compilers
Linkers.
Editors.

=> System programs :- OS
Real-time system

=> For the 3 categories, Bohem provided 2 basic expressions to predict the effort & project duration in PM & months respectively.

Page No.
Date: / /

Basic Expression

$$Effort = a_1 \times (KLOC)^{a_2} \ PM$$

$$Tdev = b_1 \times (Effort)^{b_2} \ months$$

where

KLOC → size of product in kilo lines of code

a_1, a_2, b_1, b_2 → const.s

Tdev → estimated time to develop software

Effort → Total effort required to develop software

Development Effort

Organic → $2.4 (KLOC)^{1.05}$ PM

Semi-detached → $3.0 (KLOC)^{1.12}$ PM

Embedded → $3.6 (KLOC)^{1.20}$ PM

Development Time

Organic → $2.5 (Effort)^{0.38}$ months

Semi-detached → $2.5 (Effort)^{0.35}$ months

Embedded → $2.5 (Effort)^{0.32}$ months