Software Engineering

* It is a program/set of programs containing instructions which provide desired functionality.
* Also comprises of data structure that enable the program to manipulate information.

The process of designing & building something that serves a particular purpose.

SE: A systematic approach to the development, operation and maintainance of desired software.

Dual role of software :-

① As a product: It delivers the computing potential of a hardware (H/w).
- ↳ enable the H/w to deliver the expected functionality
- ↳ Acts as information transformer
  - ↳ Bussines info. → e.g. query retrieval s/w based on indirect data
  - ↳ Personal info. → e.g. Salary cheque generation.

② As a vehicle for delivering a product:
- ↳ helps in creation and control of other programs.
  e.g. Operating System.

Why software engineering important?

→ enables us to build complex system (S/W) in a timely manner.

→ ensures high quality of S/w.
→ Impose discipline to work that can become quite chaotic.

What is Work Product (Result/ outcome)

(★) **Software engineer** -
↳ the set of programs, the (data) content along with documentation that is part of software (S/w).

(★) **User / customer** -
↳ the functionality delivered by the software that improves the user experience.

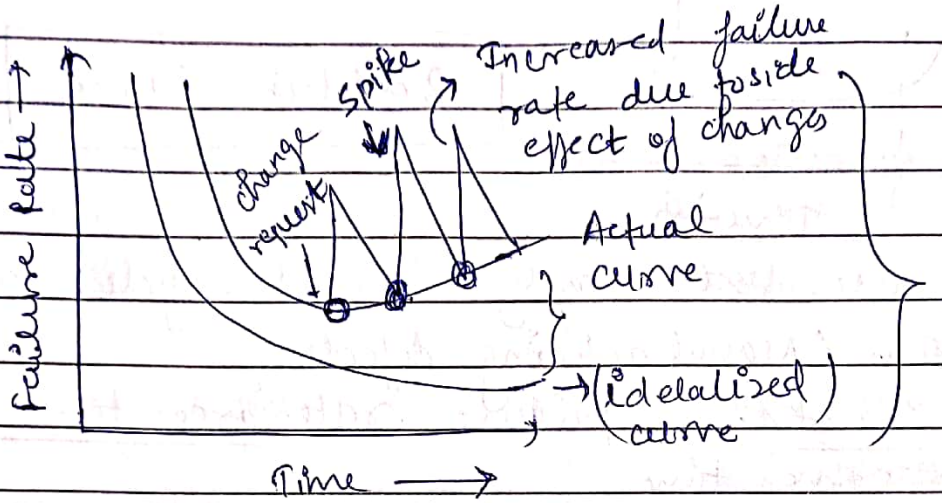
Software Engineering focuses on?

(★) **Quality** → {while S/w development}
- → **functional** : degree to which correct software is produced.
- → **Non-functional** : features other than functions of a S/w. like robustness.
  - (also known as structured attributes)

(★) **Maintainability** → {after the S/w has been delivered and developed} should be easily enhanced and adapt to changing requirements whenever required.

Software :-

↓ S/W does not wear out, it deteriorates



⇒ Not an idealized curve
↳ undetected defect

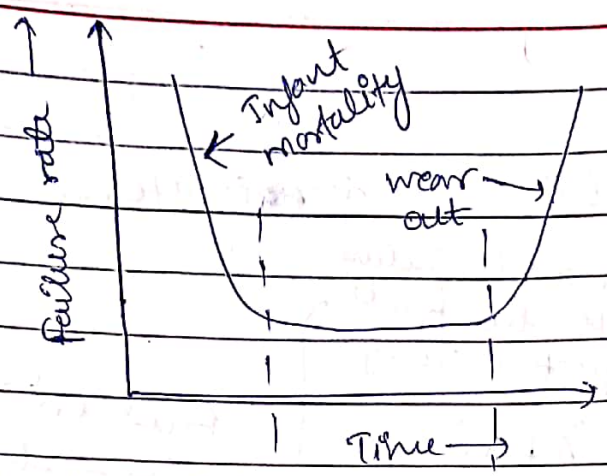
→ High failure rate in early life of S/w.
→ Defects corrected & curve flattens.
→ During its life, S/w undergoes change.
↳ new errors may be introduced
↳ cause spike (rise) in failure rate.
→ curve spikes with every change that caused new error.

Minimum failure rate rises.

Hardware :-

→ Hardware wears out with time.

→ Also known as Bath Tub Curve.



⇒ failure rate as a function of time for H/W.

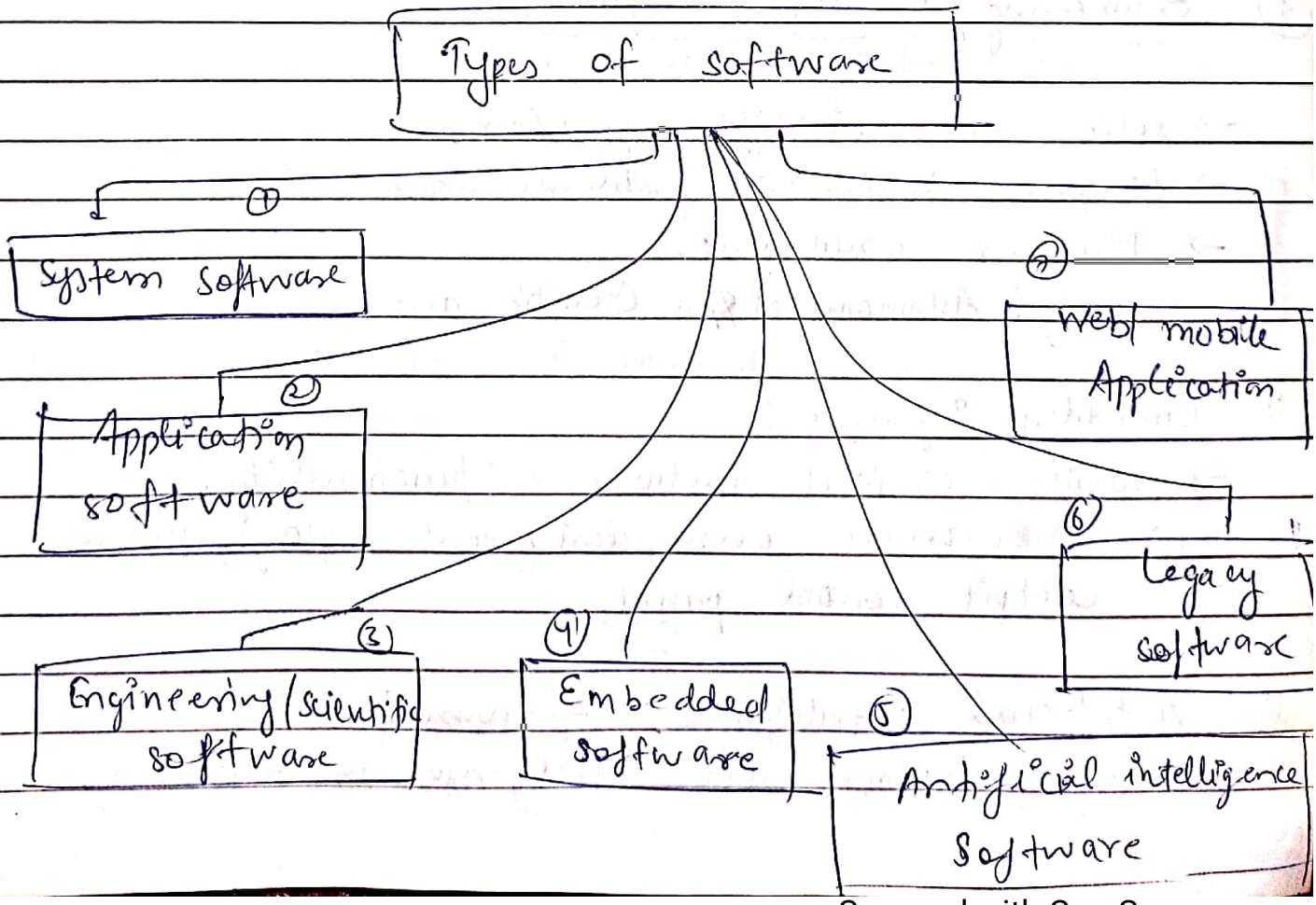
Bathtub Curve

→ High failure rate early in life cycle.
  ↳ Design / Manufacturing defects.
→ Defects corrected & failure rate drops to a steady level for some time.
→ with time failure rate rises again.
  ↳ Environmental factors :- Dust, temp, pollution etc.

Wearing out of Hardware

Software	Hardware
* Software is a logical unit. ↓ intangible in nature ↳ Developed by programmers.	* Hardware is a physical unit. ↓ tangible in nature. ↳ Manufactured in factories.
* There are no spare parts in Software.	* Spare parts for hardware exists.
S/w modules may be dependent on each other.	Independent of each other

* Problem statement ↳ It may not be complete & clear initially.	* Problem statement ↳ clearly specified at the beginning of the production process.
* Requirements ↳ may change with time as S/w is developed.	* Requirements ↳ Are fixed before H/W manufacturing begin.
* Multiple copies less costly than H/W.	* Multiple copies are created using assembly line etc. ↳ costlier than copying a S/w.



① System software :-

→ The softwares that provide a platform for other s/w to run.

→ they service other programs.
eg- Operating system, compiler etc.

② Application software :-

→ They serve a particular purpose.
→ They solve a particular business purpose we say these s/w act as business s/w. eg - payroll s/w.
→ Personal needs (PC software)
↳ to make user world easier.

③ Engineering / Scientific software :-

→ solve a scientific problem.
→ Complex numerical problems are solved.
→ Number crunching.
eg : Astronomical s/w, Genetic analysis.

④ Embedded Software :-

→ Provide limited features & functionality.
eg - Macrowave ovens, dashboard displays, plane cockpit control panel.

⑤ Artificial intelligence Software :-

→ Induce human like intelligence in machines.

→ Complex algorithms which are non-numeric.
→ eg. Robotics, Game playing s/w like chess.

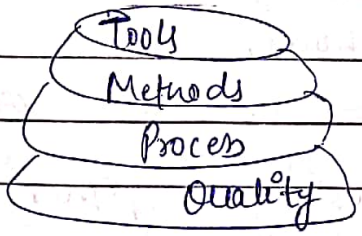
(6) Legacy software :-

→ very old and traditional software.
→ changed from time to time.
→ Do not have a good quality.

(7) web/ mobile Applications :- → they run on mobile device
↓ eg - games, wallpaper app.
These s/w that run on browser.
eg - online editing s/w.

Real word s/w → eg - weather forecast etc.
↳ control, monitor, analyze real world event in real time.

Software engineering : A layered technology



* software engineering comprises of a process, a set of methods for managing and developing the

S/w and a collection of Tools.

* The bedrock that supports software engineering is Quality focus. (Imp. layer)

Quality → Functional → degree to which correct software has been developed.
→ Non-functional (structural)
eg: Maintainability, robustness etc.
→ eg. Six sigma technique, TQM (Total quality management)

Process :- A framework that must be established for
→ effective delivery of S/W.
→ Timely development of software.
→ Management and control of S/w projects.
→ consists of several method.

Methods :- Provide technical "how-to" for building a software
→ each method consists of multiple tasks.
eg :- requirement analysis, testing, support.

Tools :- Provide automated / semi-automated support for process and methods.
CASE CAD → computer Aid software or design.
→ set of tools working together to deliver a functionality in the software.

Software Life Cycle Models

→ The process followed in the development of the S/w depends upon the Life Cycle Model chosen for development.

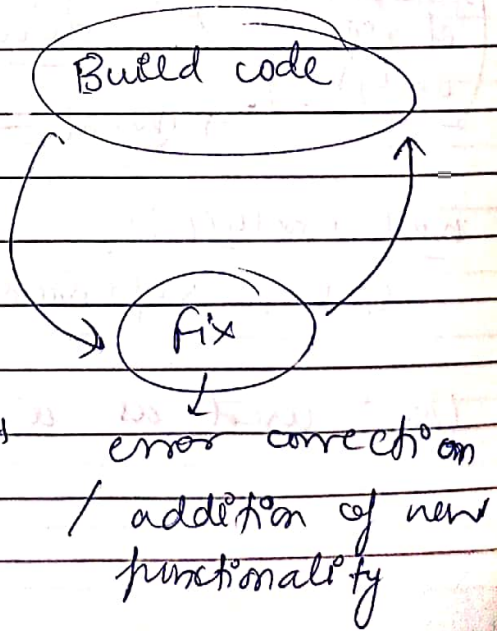
→ Life cycle model defines the major phases during and after the S/w development process.

Types of Life Cycle Models:-

1. Build and Fix Model
2. Waterfall model
3. Incremental Process models
   - (a) Iterative model
   - (b) Rapid Application development (RAD).
4. Evolutionary models
   - (a) Prototyping model
   - (b) Spiral model
5. Rational unified process model.

(1) Build and Fix model :-

→ A phase model consisting of writing code → fixing it →



→ Product constructed without any formal requirement specifications or design.

→ Directly developer start building the product which is fixed (changed / reworked) as many times as the customer desires.

Disadvantage :-

① Not suitable for large size S/W.
② Cost is very high as compared to a structured approach.
③ Code become unfixable very soon.
④ Maintainance is very difficult
   ↳ Lack of requirements & design documents.

② Waterfall model :- (Classic or Traditional model)

→ 5 phase model
→ Each phase executed sequentially without overlap.

Purpose of req. analysis :- To understand customer requirements & document them properly.
→ what to do → functional & non-functional requirements.

work product :-
SRS : software requirement Specification Document.

Use : used as a contract b/w developer & costumer.

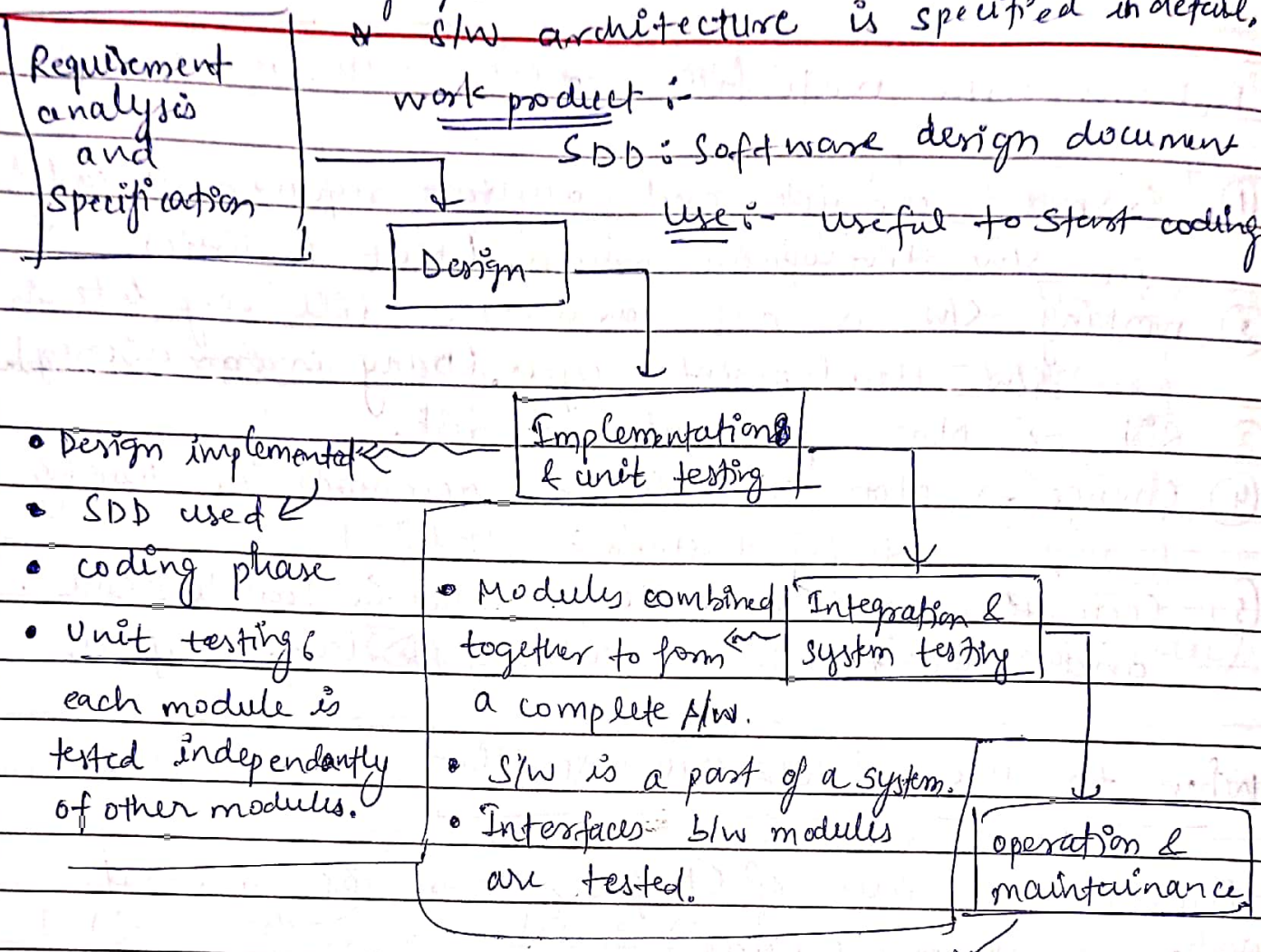
Design Phase → purpose : Transform requirements into a structure suitable for implementation in some programing language.

* S/w architecture is specified in detail.

work product :-

SDD : Software design document

Use :- useful to start coding.



purpose :- To preserve the value of s/w over time.

• After the deliver (deployment) of the s/w to customes.
• error correction, enhancement remove obsolute functionalities.

Problems with waterfall model :-

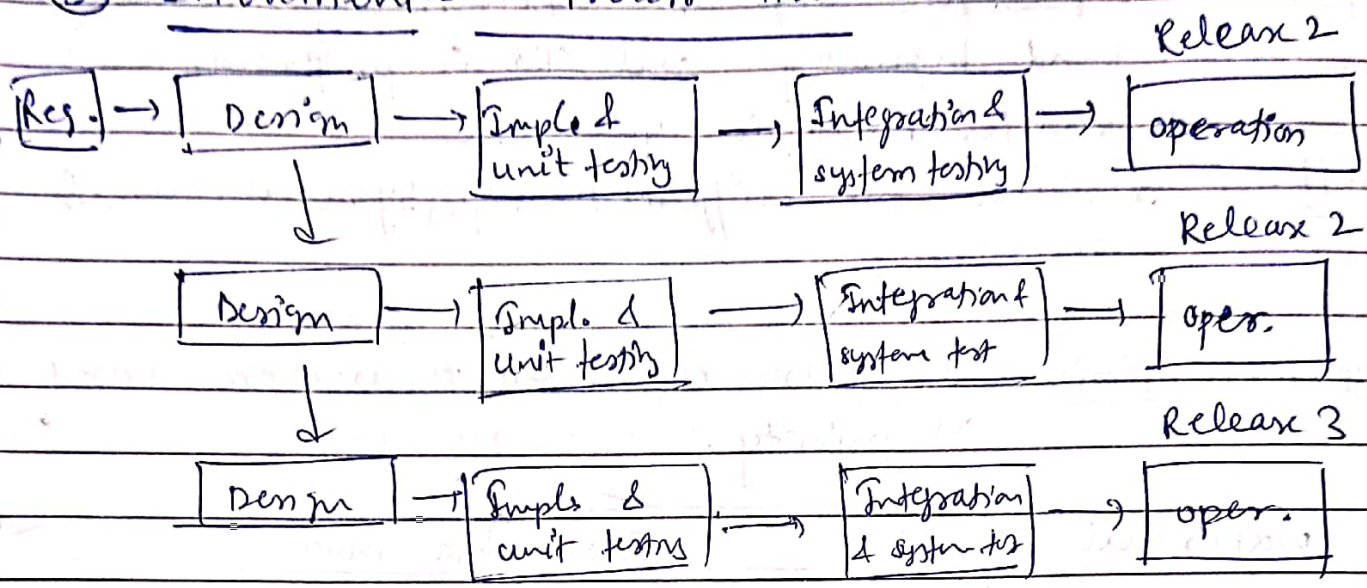
1. Expects complete and accurate requirement early in S/W development process. (Not Realistic).
2. working S/W is not available till very late in the S/W development cycle. (Delay in error discovery).
3. Risk → No assessment of risk.
4. Change → Not suitable for accomodating changing during development.
5. Sequential natare → Not realistic in today's world and this model is suitable for large projects.

When to use waterfall Model :-

Follows the idea of Define Before Design and Design Before code.

1. When the requirements are very clearly understood & they will not change during SDLC.
   stable
2. It can be used by an organisation :
   (a) that has an experience in developing a particular kind of software.
   (b) when it wants to build a new software based on an existing S/W.

(3) Incremental Process model:-



(Iterative Enhancement model)

Requirements :- are defined precisely.
↳ No confusion about requirements.

Delivery of functionality is done in multiples/phases depending upon the priorities of the requirements.

Every cycle:
↳ delivers a semi-complete product with limited functionality.
→ Last cycle: complete S/W.

(A) Iterative Enhancement Model:-

Phases: same phase as waterfall model
↳ they are present with less restriction.
↳ occur in same order but in the several cycles.

Requirements : Major req. are specified by the customer at beginning and SRS is prepared.

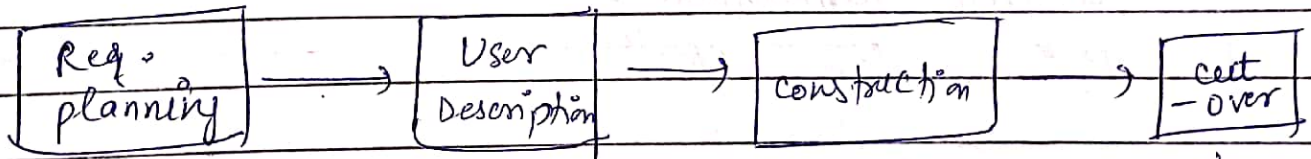
→ Reason for multiple cycle:

Priorities :- decided for different req. by customer & developer.

Implementation :- of these requirements is done based on priority.

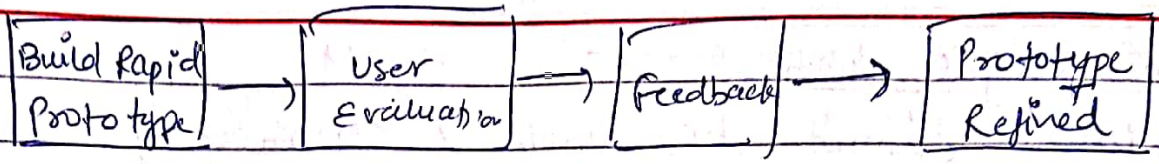
waterfall model	Incremental model
① One final product ↳ at the very end of SLDC ↳ with all functionalities and features.	① Complete S/w is divided into releases ↳ Limited functions.
② Long wait for the S/w.	② First release is available within week/months.

(B) Rapid Application Development Model :



with Active User Participation

→ Developed by IBM.



Requirement planning :-

- Using group elicitation techniques like brain-storming
- User communication req. for good understanding.

as long as requirements are finalized and done in requirement planning.

User Description :-

↳ A joint team of customer & developer understand & reviews the gathered requirements.

↳ Automated tools may also be used.

Construction :-

↳ Design → Coding → Testing ↳ code generators + screen generators.

↳ Product is released.

Cut-over :-

↳ Install the S/W + Acceptance testing + Training of user.

Major features :-

① Rapid prototype : Quick initial views about product.

② Powerful development tools → development time ↓.
e.g. CASE Tools

③ User involvement :- Acceptability of product ↑.

(Req. → Requirements)

Not useful when:-
→ User cannot be involved continously.
→ Tools and Reusable components cannot be used.
→ High skilled & specialized developers.
→ System cannot be modularized.

④ Evolutionary Models :-

Phases : Same as waterfall model but they are applied in cyclic manner.

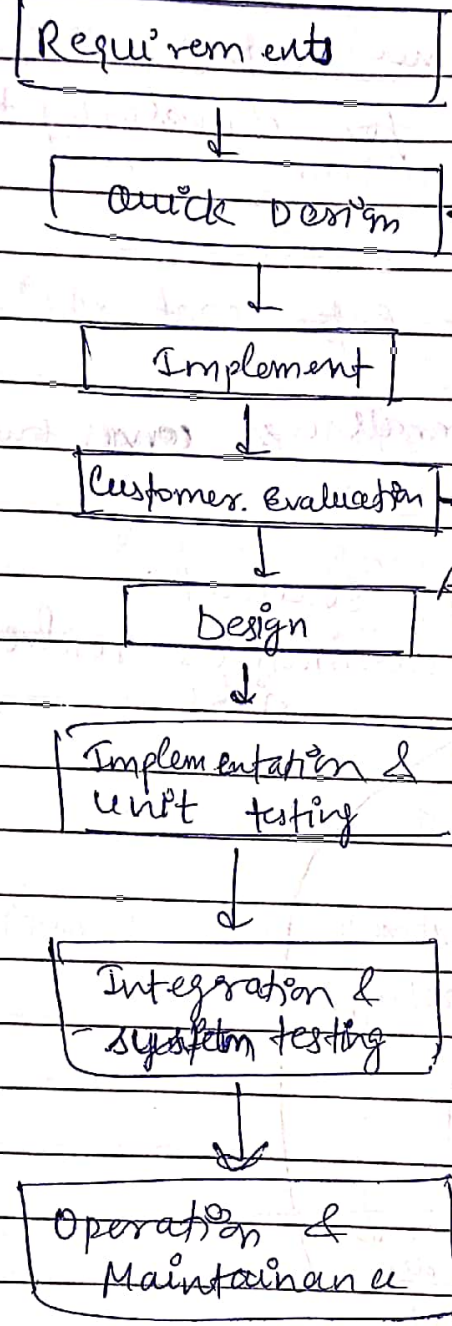
Difference as compared to Iterative Models

Iterative	Evolutionary
① A Usable product is delivered at end of each cycle.	① No usable product at end of each cycle.
② Req. implemented priority wise.	② Req. implemented category wise

Useful when :- → when a minimal version of the system is not required.
→ complex projects with unstable req.
→ New Technology
→ Req. are not clear

continous customer support.

(A) Prototyping Model :-



→ first develop a working prototype of the s/w instead of s/w

- limited functionality
+
Not very good quality
+
Low performance etc.

- used to
  - Refine requirements
  - prepare fine SRS.

final system developed using waterfall model

Is prototype our final product?

- No its not our final product.
- prototype is discarded after requirements are finalised.

Benifits of developing a prototype :-

↳ Helps us to unravel the customer req.

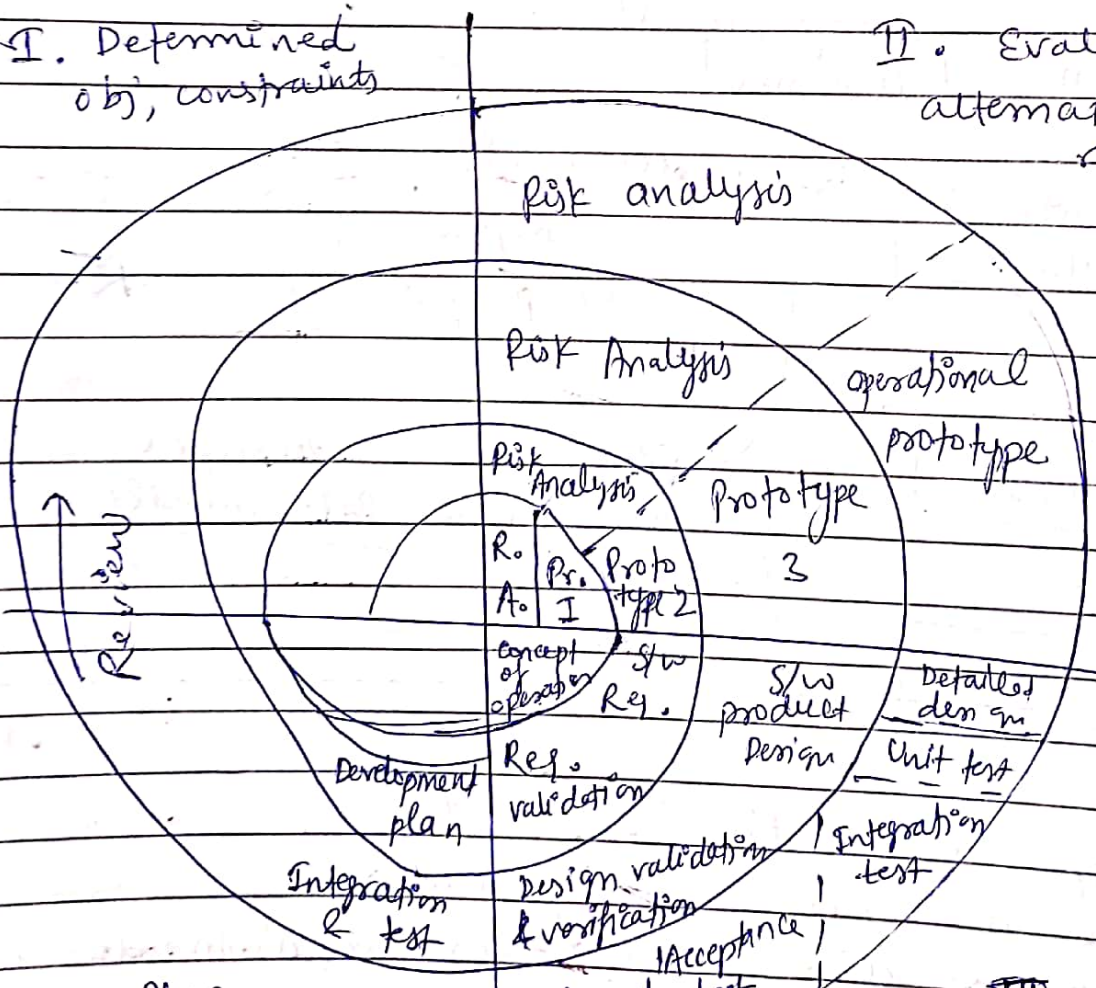
↳ Helps us to gather experience for developing the final system.

Does Prototype Development incur extra cost

↓

cost increase but overall cost comes down.

(B) SPIRAL MODEL :-



Developed by : Barry Boehm

Main feature : Handling uncertainty / Risk

Phases : Multiple phases having 4 activities.

Activities :-

① Planning
   ↳ Determine objectives + Alternatives
   ↳ constraints
② Risk Analysis, & resolve
   ↳ Identify risks, classify into levels.
   ↳ Alternatives & plan ahead
③ development and Testing
④ Assessment : Customer evaluation.

Phase 1 : Planning → Risk analysis → Prototype is build
Customer Evaluation & feedback ←

Phase 2 : Prototype refined → Req. documented and validated by customer
final prototype ←

Phase 3 : Risk are now known + Traditional approach for development.

Focus Areas :- (Plan for next cycle beforehand)
- Identify problem → classify into different risk levels
eliminate them before they affect S/W.

- Continous validation (review) by concerned people (designer or developers) to check the work product quality.

Advantages :-

① Acomodate good features of other SDLC models.

② S/w quality maintained during development by :
   - ↳ continous Risk Analysis
   - ↳ Reviews conducted.

Disadvantage :- Expertise required in Risk handling + carrying out spiral model.

SOFTWARE REQUIREMENT ENGINEERING (RE)

(understanding the req. of customers) + (Documenting the Req.)

What is it?

→ A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfil the system's purpose.
↳ (what to do) and (it does not tell us how to do).

Work Product:- RE produces one large document written in natural language, containing a description of what system will do without describing How it will do it.

↳ Software Requirement Specification & DOC (SRS)

Input to RE: problem statement prepared by customer.

(overview of the existing system) + (New or additional functionality to be added.)

Importance:- without well written requirements:-
* Develops will not know what to do.
* Customer may not be consistent about requirements.
* It becomes difficult to validate / accept the S/W.

4 Steps in Requirement Engineering :-

① **Requirement Elicitation :-** Gathering of requirements

→ Requirements are identified with the help of customer (existing system)

② **Requirement Analysis :** Requirements are analysed to :

- Fix inconsistancies, contradictions, Ommissions

③ **Requirement Documentation :** End product of first 2 steps. Leads to

* Preparation of SRS
* Becomes foundation for design of S/W.

④ **Requirement Review :** To improve the quality of SRS

or

Requirement verification

Software Requirement Elicitation :- (Requirement gathering)

Requirement Elicitation is an activity that helps us to understand what problem has to be solved & what customer expect from the S/w.

Foundation: Effective communication b/w customer - developer.

Method: Developer questions customer
↳ respond → cross-questioning

Hurdles :-
↳ Misunderstandings / conflict.
↳ communication gap (Omission of requirements).

Reason for conflicts between concerned people

(i) Developer is efficient in the knowledge of his own development domain while customer is effecient only in his domain (different from developer's domain).
(ii) Lack of proper communication skills.

Requirement Elicitation method :-

① Interviews :
Purpose :- To understand each other.

- Requirement engineer act as mediators b/w customer and development team.

* Correct Approach :-
  - open - minded, co-operative, understanding, flexible.

Types of Interviews

Structured	Open-Ended
① There is a set agenda ↳ Ques. may be prepared.	① No pre-set agenda ↳ no pre-defined list of ques. prepared.

* Question should be short and simple, clear.
* Both parties should be open for discussion to proceed in any direction.

Types of Stakeholders to interview

① Entry level personnel
- ↳ Do not have much domain knowledge.
- ↳ They have new ideas / perspectives.

② Mid level Stakeholders → (Project mangers/lead)
- ↳ Experienced people with good domain knowledge.
- ↳ They know the criticality of project.

③ Managers and Higher Management.
- ↳ useful insight about project.

(4) Users of S/w

Most imp. → because they will be using the S/w max. no. of times.

(II) Brainstorming sessions :-

↳ Group discussion technique
↳ Promote creative thinking & new ideas.
↳ Platform to express and share views, expectation & difficulties in implementation.

↳ Facilitator
↳ ego clashes / conflicts (to avoid them)
↳ encourage people to participate as much as possible.

• work product → Document

→ All ideas are documented to be visible to each participant (using white boards, projectors).
→ Detailed report, containing each idea in simple language, is prepared & reviewed by facilitator.
→ At the end, a document is prepared with list of requirements & their priority.

(III.) Facilitated application specification technique

(IV). Quality function deployment.

Desirable characteristics of a SRS :

1. Consistent
2. Correct
3. Complete
4. Understandable
5. Basis for design & testing
6. Acts as a contract
7. Modifiable
8. Verifiable
9. Traceable
10. Unambiguous

Types of Requirements :- (SRS also contains CONSTRAINTS)

(i) Functional & Non-functional requirements
(ii) User and system requirements → (Complexity, language diff.)
(iii) Interface specification

Stakeholder :- refers to anyone who have some direct or indirect influence on the system requirements
- ↳ end-user who will interact with system
- ↳ anyone (except user) who will be affected by the s/w that is developed.

Functional Requirements :- (Product features)

↳ Describe what the system has to do.
↳ what are the expectations from the s/w.
↳ what the s/w should not do.

eg :- features of a game.

Non-functional Requirements :-

↳ Mostly Quality requirements.

↳ Highlight how well the S/W performs its function.

eg- For users :- High performance ; reliability, usability.

For developers :-

↳ Maintainability, Testability, Portability.

User requirements :-

* Written for users who are NOT experts of S/W field.
* Highlight the overview of system without design description.
* Specifies → function + non-function req.
  - function → External Behaviour
  - non-function → constraints → quality

* what to avoid?
  - ↳ Complex language
  - ↳ Design details
  - ↳ Technical terms and values.

System requirements :-

↳ Derived from user req. or expanded form of user requirements.

↳ used as input to designers so that they can prepare SDD (Software design document).

Both user & system req. are a part of SRS.

Interface Specification :-

↳ Application Programming Interface (API) are specified in SRS.

↳ what kind of interfaces customer desires.

feasibility studies :-

↳ Determines if the project is workable or not.

↳ work product : feasibility report.

↳ use → it helps the management / project team / customer to decide if the S/W should be built.

What factor are considered ?

(I) current practices V/s proposed system

(II) Amount of resources needed.

(III) Major risks that can occur.

(IV) cost and schedule

(V) Technical skills required.

Benifits :-

(1) 10 % failure V/S 80% failure

better

(2) More accurate estimates can be prepared.

Characteristics of a good SRS (Detailed) :-

① **Consistency**
- ↳ No conflicts b/w the requirements.
- ↳ every req. must be specified using a standard terminology.

② **Correct**
- ↳ what is stated is exactly what is desired
- ↳ Expected functionality matches the req. present in SRS.

③ **Unambiguous**
- ↳ Every stated req. has only 1 unique meaning.
- ↳ Words with multiple meanings?
  - ↳ these words should be specificied with meaning.
- ↳ Software requirement specification language can be used.

Adv :- Language processor exists which tell diff. kind of errors in SR

Disadv :- Understanding of this language is not for everyone.

④ **Complete**
- ↳ includes all functional + non-functional req. + constraints
- ↳ Specifies expected output from all kinds (valid/invalid) input from the user.

(5) Traceable

↳ origin of each req. should be clear.

↳ important because feature referencing may be required for development / maintainance.

(6) Verifiable (SR verifiable iff each req. verifiable)

↳ iff there is a process to check if the S/W meets each of its req.

↳ "Good", "fast". } Ambiguous req. can never be verified.

(7) Modifiable (keep all cross-references).

↳ easy to make changes in SRS retaining its structure + Correctly Modify.

(8) Ranked for stability / Importance

↳ Each req. has ranking for its stability as well as for importance.

Software Size Estimation

① Line of Code :-

What is LOC?
↳ Declarations, Actual code including logic and computation.

What is NOT a LOC?

- Blank lines
  ↳ Included to increase / improve readability of code
- Comments
  ↳ Included to help in code understanding as well as during maintainance.

→ Not a LOC because
(i) Do not contribute to any kind of functionality.
(ii) Misused by developers to give a false notion about productivity.

Advantage of using LOC for size estimation :-

(i) very easy to count and calculate from the developer code.

Disadvantage of using LOC for size estimation :-

(i) LOC is language + technology dependent
(ii) What constitutes an LOC → varies from organisation to organisation.

Eg → program 1 →

```
for ( int i=1 ; i<10 ; i++)
    cout << i
```
} ② lines

```
for ( int i=1 ; i<10 ; i++)
{
    cout << i;
}
```
} ④ lines

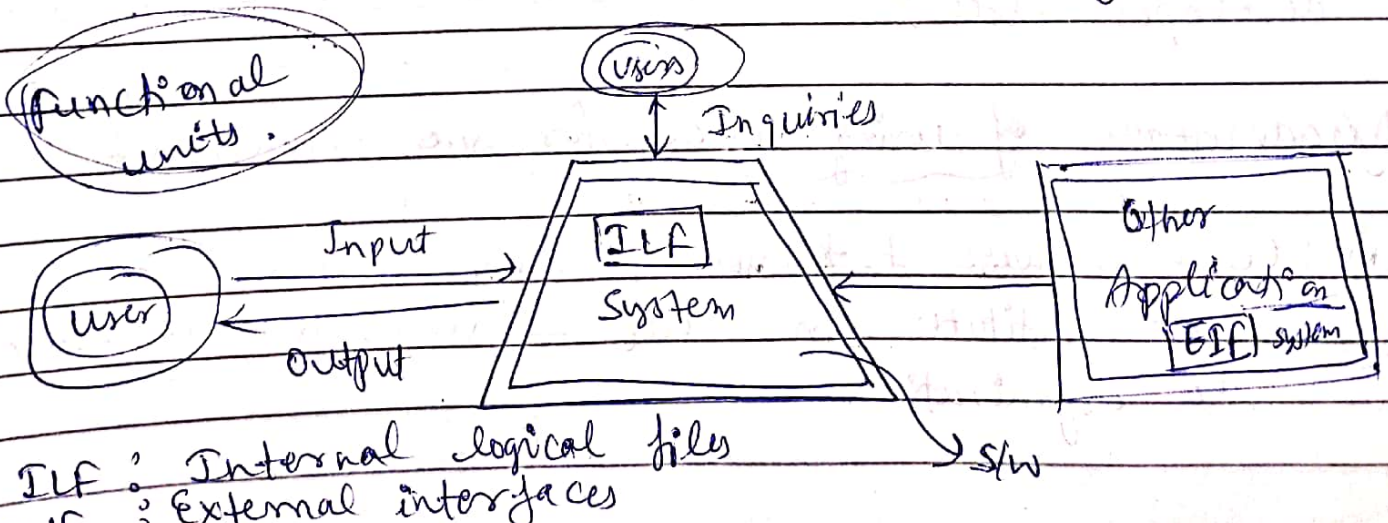
→ Both same

```
int a; // Declaration
```
code comment → Ambiguity

```
// Declaration
int a;
```
→ comment
→ code

② Function Points :-

* It measures functionality from user's point of view.
  ↳ what the user receives from S/W + what the user request from S/W.
* Focuses on what functionality is being delivered.



ILF : Internal logical files
EIF : External interfaces

A system has 5 functional units :-

(i) Internal Logical Files (ILF) :-
↳ The control info or logically related data that is present within the system.

(ii) External Interface files (EIF) :
↳ ~~The~~ control data or other logical data i.e., refenced by the system but present in another system.

(iii) External Inputs (EI) :
↳ data / control info that comes from outside our system.

(iv) External Outputs (EO) :
↳ data that goes out of system after generation.

(v) External Enquiries (EQ) → combination of I/P & O/P resulting in data retrieval.

Data function types
- ILF
- EIF

Transaction function types
- EI
- EO
- EQ

Counting function Points :-

Step-1 :→ Each function point is ranked according to complexity.

- Low
- Average
- High

There exists pre-defined weights for each f.p. in each category.

Table A

Functional units ↓i	j → weighting factors Low	Average	High
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

(Functional units)

Step-2 :→ Calculate Unadjusted function Point by multiplying each f.p. by its corresponding weight factor.

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} (z_{ij} * w_{ij})$$

z_{ij} : count of no. of functional units of category "i" classified as complexity "j".

w_{ij} : weight from Table A

How to Assign weights or rank function points?
↳ Dependant on the organization.
↳ Based on post projects.

Step-3:- Calculate Final function Points

$$\text{Final F.P.} = \text{UFP} \times \text{CAF}$$

CAF → complexity adjustment factor

Calculated using 14 aspects of procesing complexity

- 14 questions answered on a scale of 0 to 5.

0 → No influence
1 → Incidental
2 → Moderate
3 → Average
4 → Significant
5 → Essential

$$CAF = [0.65 + 0.01 \times \Sigma f_i]$$

[f_i = i varies from 1 to 14]

Qwy → Given the following values, compute f.p when all complexity adjustment factors (→ CAF) and weighting factors are average.

User I/p = 50
User O/p = 40
User enquiries = 35
User files = 6
External interface = 4

Sol =>

$$\sum f_i = 14 \times 3 = 42 \quad (\text{Avg})$$

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} Z_{ij} \, W_{ij}$$

$$UFP = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7$$
$$= 200 + 200 + 140 + 60 + 28$$
$$= 628$$

$$CAf = 0.65 + 0.01 \times \sum f_i$$
$$CAf = 0.65 + 0.01 \times 42$$
$$CAf = 0.65 + 0.42$$
$$CAf = 1.07$$

$$F.P = UFP \times CAf$$

$$F.P = 628 \times 1.07$$

$$\boxed{F.P = 671.96}$$

Advantages of function Point Approach :

* Size of S/W delivered is measured independent of
  - → Language
  - → technology

* F.P. are directly estimated from requirements, before design & coding.

↳ we get an estimate of S/W size even before major design or coding happens (early phases).

↳ Any change in req. can be easily reflected in F.P. count.

* Useful even for those user without technical expertise.

↳ F.P. is based on external structure of the S/W to developed.

COCOMO MODEL

Constructive Cost Model

* Developed by ~~&~~ Barry Bohem

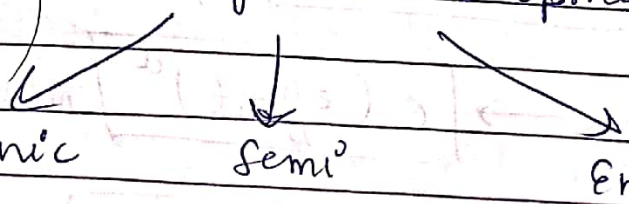
It is a hierarchy of S/W cost estimation model
- Basic model
- Intermediate model
- Detailed model

① Basic Model -

→ It estimates the software in a rough and quick manner.

→ Mostly useful for small - medium sized S/W.

→ 3 models of development



	Organic	Semi Detached	Embedded
• Size	2-50 KLOC	50-300 KLOC	300 or above KLO
• Team Size	Small size	Medium size	Large team
• Developer Experience	Experienced Developer needed	Average experienced people	very little previous experience.
• Environment	familiar Environment	Less familiar	Significant environment changes (Almost new)
• Innovation required	Little	Medium	Major
• Deadline	Not tight	Medium	Tight
eg :→	Payroll System	Utility system like compilers etc.	Air traffic monitoring system.

Basic Model Equations :-

① Effort → $a(KLOC)^b$ person-months

② Development Time → $c(effort)^d$ months

③ Average staff size → $\frac{Effort}{Dev.\ time}$ persons

(4) Productivity → $\frac{KLOC}{Effort}$ KLOC/P.M

Mode	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic

Ques: Suppose that a project was estimated to be 400 KLOC. Calculate effort & time for each of 3 models of development.

(i) Organic:- $a = 2.4$ $b = 1.05$

Effort = $2.4 \times (400)^{1.05} \simeq 1295$ PM

Dev. time = $2.5 (1295)^{0.38}$

$\simeq 38$ M

(ii) Semi detached

Effort = $3.0 (400)^{1.12} \simeq 2462$ PM

Dev. time = $2.5 (2462)^{0.35} \simeq 38.4$ M

(iii) Embedded

Effort = $3.6 (400)^{1.20} \simeq 4772$ PM

Dev. time = $2.5 (4772)^{0.32} \simeq 38$ M

COCOMO : The Intermediate Model

* Difference b/w Basic & Intermediate Model
  - ↳ Basic model was quick but inaccurate and phase insensitive.
  - ↳ Intermediate model includes a set of 15 additional predictors (COST DRIVERS).
  - ↳ It also takes development environment into account during cost estimation.

* Use of Cost Drivers
  - ↳ Cost drivers adjust NOMINAL cost of project to actual project environment.
    - ↳ Increase the accuracy of estimation.

* 15 cost drivers

I. Product Attributes
  - (a) Required software Reliability (RELY)
  - (b) Database Size (DATA)
  - (c) Product complexity (CPLX)

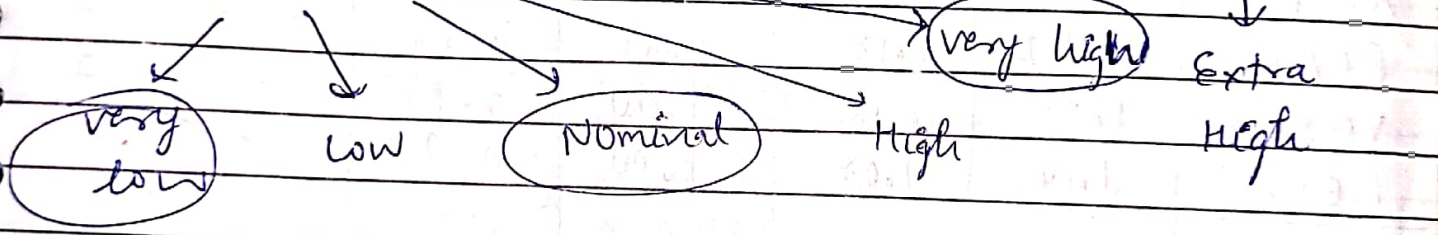
II. Comput Attributes
  - (a) Execution time constraint (TIME)
  - (b) Main Storage constraint (STOR)
  - (c) Virtual Machine constraint (VIRT)
  - (d) Computer turnaround time (TURN)

III. Personal Attributes
  - (a) Analyst Capability (ACAP)

(b) Application experience (AEXP)
(c) Programer Capability (PCAP)
(d) Virtual Machine Experience (VEXP)
(e) Programming Language Experience (LEXP)

IV. Project Attributes
(a) Modern Programming Practices (MODP)
(b) Use of software tools (TOOL)
(c) Required development schedule (SCED)

* Each cost driver is rated for the given project environment



To what extent the Cost Driver applies to the project.

Effort Adjust Factor (EAF) :-

↳ Calculated by multiplying all the values that have been obtained after categorising each cost driver.

Equations for COCOMO :-

$$\text{Effort} = a_i (KLOC)^{b_i} * EAf \quad \text{(PM)} \rightarrow \text{Person months}$$

$$\text{Development time} = c_i (\text{Effort})^{d_i} \quad \text{(M)} \rightarrow \text{months}$$

Cost Driver	very low	Low	Nominal	High	very High	Extra High
Prod. Attr.						
RELY	0.75	0.88	1.00	1.15	1.40	–
DATA	–	0.94	1.00	1.08	1.16	–
CPLX	0.75	0.85	1.00	1.15	1.30	1.65
Comp. Attr.						
TIME	–	–	1.00	1.11	1.30	1.66
STOR	–	–	1.00	1.06	1.21	1.56
VIRT	–	0.87	1.00	1.15	1.30	–
TURN	–	0.87	1.00	1.07	1.15	–
Pers. Attr.						
ACAP	1.46	1.19	1.00	0.86	0.71	–
AEXP	1.29	1.13	1.00	0.91	0.82	–
PCAP	1.42	1.17	1.00	0.86	0.70	–
VEXP	1.21	1.10	1.00	0.90	–	–
LEXP	1.14	1.07	1.00	0.95	–	–
Project Attr						
MODP	1.24	1.10	1.00	0.98	0.82	–
TOOL	1.24	1.10	1.00	0.91	0.83	–
SCED	1.25	1.08	1.00	1.04	1.10	–

Mode	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

COCOMO Intermediate Model - Numerical

Ques → A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring 2 pools of developers : very highly capable (with app.) with very little experience in programming language or developers of low quality but lot of programming language experience. which is better choice in terms of 2 pools ?

Sol →

$$E = a(KLOC)^b \times EAf$$

$$E = 2.8(400)^{1.20} \times EAf$$

Case I → $EAf = 0.82 \times 1.14$

$= 0.934$

$E = 2.8(400)^{1.20} \times 0.934$

$$\boxed{E = 3470 \text{ PM}}$$

Dev. time $= 2.5(3470)^{0.32}$

$$\boxed{\text{Dev. time} = 33.9 \text{ M.}}$$

Case II → $EAf = 1.29 \times 0.95$

$= 1.22$

$$\boxed{E = 3412 \times 1.22 \simeq 4528 \text{ PM}}$$

Dev. time $= 2.5(4258)^{0.32}$

$$\boxed{\text{Dev. time} \simeq 38.9 \text{ M}}$$

COCOMO: Detailed Development Mode

Detailed Mode is phase Sensitive,

it calculates the effect of cost drivers on each phase of SDLC.

* It uses phase - sensitive effort multipliers for each cost driver to determine the amount of effort required to complete each phase of SDLC.

* It establishes Module - Subsystem - System Heirarchy.
  ↳ The rating of cost driver is done at that level only where (C or D) cost driver is most susceptible to variable.

* Adjustment factor (A)

$$A = 0.4(DD) + 0.3C + 0.3I$$

DD - Design documentation
C - Code
I - Integration & testing

$$\text{Size equivalent} = \frac{(S \times A)}{100}$$

Mode and code size	plan & Req.	system design	Detail design	Code & Test	Integration & testing
Life cycle phase value of μ_p					
Organic small S=2	0.06	0.16	0.26	0.42	0.16
Org. medium S=32	0.06	0.16	0.24	0.38	0.22
S. dtt. medium S=32	0.07	0.17	0.25	0.33	0.25
S. det. large S=128	0.07	0.17	0.24	0.31	0.28
Embed. large S=128	0.08	0.18	0.25	0.26	0.31
Embed. XL S=320	0.08	0.18	0.24	0.24	0.34
Life cycle phase value of T_p					
Organic small S=2	0.10	0.19	0.24	0.39	0.18
org. medium S=32	0.12	0.19	0.21	0.34	0.26
S. det. med S=32	0.20	0.26	0.21	0.27	0.26
S. det. large S=128	0.22	0.27	0.19	0.25	0.29
Embed. large S=128	0.36	0.36	0.18	0.18	0.28
Embed. XL S=320	0.40	0.31	0.16	0.16	0.30

$$\text{Effort} = \mu_p E$$

$$\text{Development time} = T_p D$$

Numerical :-

Ques → Consider a project with the following main components (1) Screen edit (2) CLI (3) File I/p & O/p (4) Cursor movement (5) Screen movement. The sizes for these are estimated to be 4K, 2K, 1K, 3K KLOC. Using cocomo, determine,

1. Overall cost and schedule estimates (assume values for cost drivers, with atleast 3 being different from 1.0).
2. Cost and schedule estimates from different phases.

Sol → (I) Effort = $a_i (KLOC)^{b_i} \times EAF$

Effort = $3.2 (12)^{1.05} \times (1.15 \times 1.07 \times 0.86 \times 1.15)$

Effort = $3.2 (12)^{1.05} \times 1.216 \simeq 52.9$ PM

Sftw Reliability : High
Lang. Experience : Low
Product Complexity : High
Analyst Capability : High

Development time = $C_i (E)^{d_i}$

$= 2.5 (52.9)^{0.38}$

D = 11.29 months

(II) Effort$_d$ = $\mu_p \times E$

Development time = $T_p . D$

Effort (PM)

Plan & Req = 0.06×52.9

Syst Design = 0.16×52.9

Detailed design = 0.26×52.9

Code & test = 0.42×52.9

Integration & testing = 0.16×52.9

Development time (M)

Plan & Req. = 0.10×11.29

Syst Design = 0.19×11.29

Detailed & Design = 0.24×11.29

Code & test = 0.39×11.29

& ts = 0.18×11.29

COCOMO - II Model

* Developed by Barry Boehm.
* Revised version of original COCOMO.
* Includes 3 stages
  (1) Application composition Estimation
  (2) Early Design estimation
  (3) Post - Architecture Estimation

Application composition Estimation model

- focuses on those applications which can be quickly developed using interoperable components.
  eg - GUI, Database Manager, control packages for specific domains.

* can also be used prototyping phase of an application.

* size is estimated using objects points :
  (i) Screen (ii) Report (iii) 3GL composition

Steps for Effort Estimation :-

① Assess the object counts
Estimate the no. of screens, reports & 3GL components

② Classify complexity levels of each object

- Classify each object into (simple, Medium, difficult).
- Screens on the basis of no. of view + sources.
  Reports on the basis of no. of section + sources.

Screens

No. of views contained	Number & Sources of Data		
	Total < 4 (< 2 server, < 3 client)	Total < 8 (2-3 server, 3-5 client)	Total 8+ (>3 server, >5 client)
< 3	SIMPLE	SIMPLE	MEDIUM
3-7	SIMPLE	MEDIUM	DIFFICULT
> 8	MEDIUM	DIFFICULT	DIFFICULT

Reports

No. of sections	Number & sources of Data Tables		
	Total < 4 (< 2 server, < 3 clients)	Total < 8 (2-3 server, 3-5 clients)	Total 8+ (>3 servers, >5 clients)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4+	Medium	Difficult	Difficult

③ Assign complexity (weight) to each object depending on the category of the object.

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Component	–	–	10

④ Determine object points :-

↓

Add all these complexity weights to get (Total) object points.

(5) Compute new object points :

↓

By considering the resource components that would be used by us in developing the software.

$$NOP = \frac{(\text{Object points}) * (100 - \%\ \text{reuse})}{100}$$

(6) Calculate Productivity Rate (PROD) :

* calculated depending on the past project data.

$$PROD = \frac{NOP}{\text{Person-months}}$$

Developer experience & capability, ICASE maturity + Capability	PROD (NOP/PM)
very low	4
low	7
Nominal	13
High	25
very high	50

(7) Compute effort in Person-Months :

$$\text{Effort} = \frac{NOP}{PROD} \text{ person-months}$$

MODULE COUPLING : A Design Issue

What is coupling?

★ Coupling is the measure of degree of interdependence b/w modules.

High coupling → Strongly inter-related / dependent modules
Low coupling → Independent modules

★ Low coupling is desired because high coupling leads to more errors + it makes isolation (debugging) of errors very difficult.

How modules become coupled / dependent? → global
↳ when module share data / exchange data or they make calls to each other.

How to control coupling?
↳ control the amount of information exchanged b/w modules.

- Pass only data NOT control information.

Types of coupling :-

- Data coupling → BEST (Most Desirable)
- Stamp coupling
- Control coupling
- External coupling
- Communication coupling
- Content coupling → WORST (least Desirable)

[{ Did she conduct exam for communication and calculus } { short trick for memorising }]

① Data Coupling :
- Communication b/w modules occur by only passing the necessary data. (No control info).
- Data passed using parameters.

② Stamp coupling :
- It occurs when complete data structure is passed from 1 module to another.

Drawback : The entire data structure is not required by receiving module, only a part of data structure is needed.

③ Control Coupling :
- Communication b/w modules occurs by passing the control info or a module controls the flow of another module.

eg : flags, data that is set by 1 module & used by another module.

④ External Coupling :
- Dependency of a module on another module which is present in a S/W / H/W external to the system.

eg :- Extern tools.

⑤ Common coupling :

- When 2 modules have common shared data/ global data (accessed by both).
- shared data makes error isolation difficult OR evaluation of change to shared data is difficult.

⑥ Content coupling :

- It occurs when control is passed from one module to middle of another OR a module changes data of another module.
  (A module modifies another module).
  (A modules relies on the internal working of another module.)

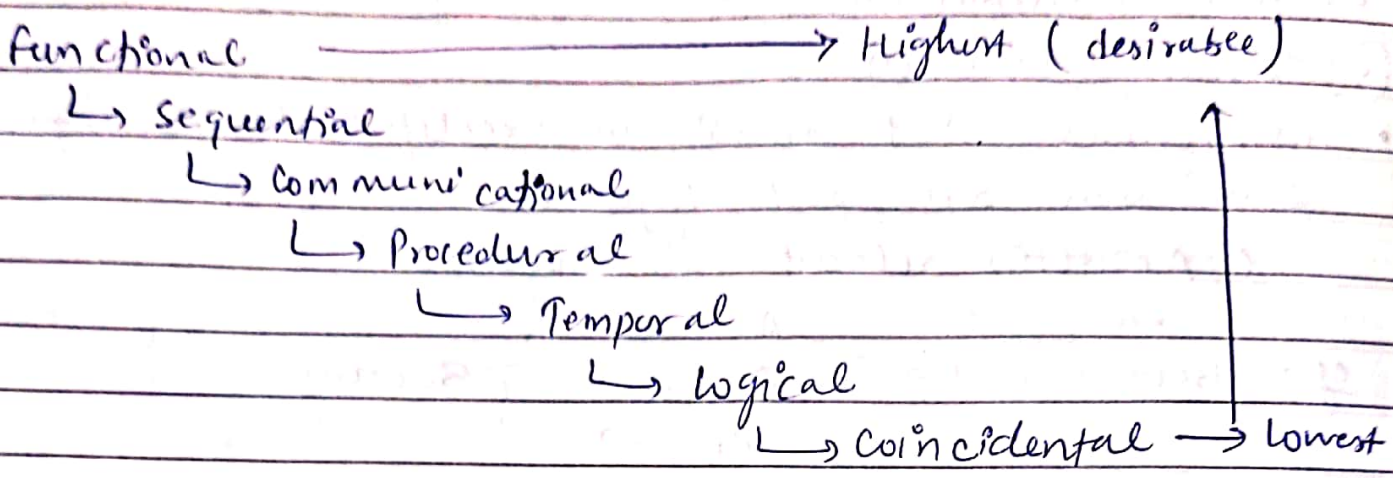
MODULE COHESION: A Design Issue

Cohesion : the degree to which elements of a module are functionally related to each other.
{cohesion is a glue that holds a module together}

A strong cohesive module implements functionality focusing on a single feature of the solution or S/w.

- High cohesion is desired.
- ⁂ [Higher cohesion ⇒ lower coupling]

Types of cohesion :-



① **Functional cohesion :**

If 2 operations present within a module perform the same functional task or they are part of the same project.

↳ Each element does some related task.

② **Sequential Cohesion :**

In a module if 2 operations are such, that :-

X's output → Y's input } X, Y ∈ same module

↳ Communicational convenience

③ **Communicational Cohesion :**

Elements inside a module that operate a same I/p data or contribute to same data.

↳ each elements affects each other.

④ **Procedural Cohesion :**

Modules whose instructions do different tasks, but to ensure a particular order in which tasks are

performed, they are put into same module.
↳ Poor maintainance.

(5) Temporal cohesion :

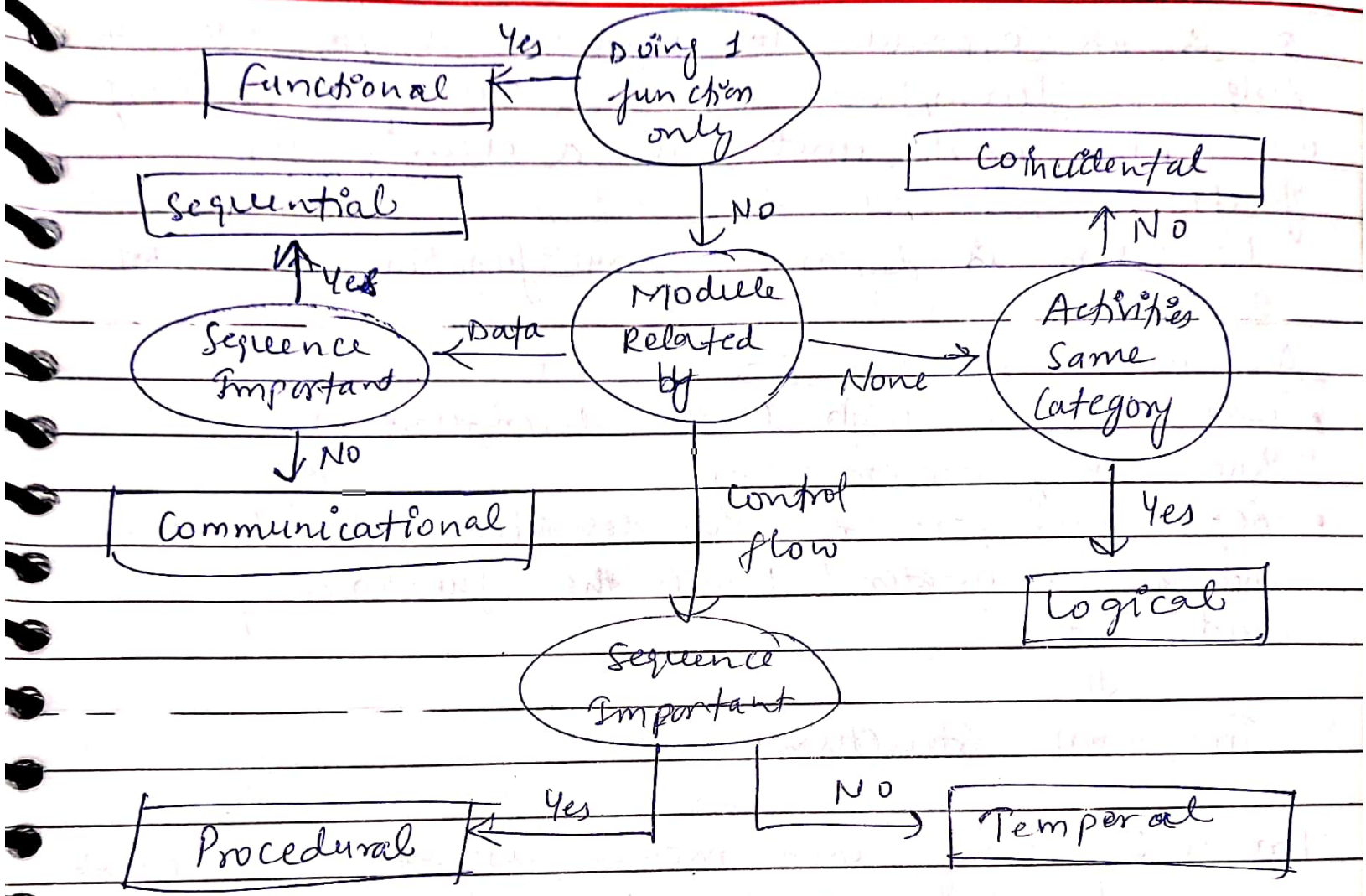
- Instructions that must be executed during same time span are put together.
  (Activities related in time).

eg: Boot set up → A1 → B1 → C1 } Booting

(6) Logical Cohesion :
↳ when modules have logically similar instruction / elements.

(7) Coincidental Cohesion :
↳ Instructions are not related to each other.

{ First Semester Of College Provided Tough & long curriculum. }



Function Oriented Design

It is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.

↳ System is designed from functional viewpoint.

A Generic Procedure :-

- Start with a high level description of what the S/W / program does.
- Refine each part of the description one by one by specifying in greater detail the functionality of each part.

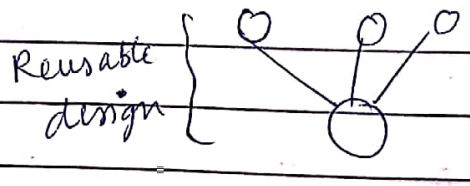
⇓

Top Down Structure

Problem : Mostly each module is used by atmost 1 other module → parent.

Solution : Designing of reusable modules.

modules use several other modules to do their required functions.



For a function-oriented design, design can be represented mathematically or graphically by using :-

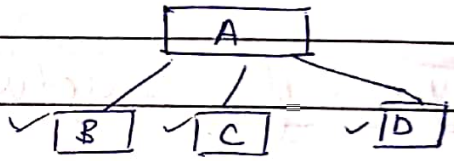
1. DFD (Data flow diagram)
2. Data dictionary
3. Structured charts
4. Pseudo code

Structured charts

- Hierarchical representation of the system.
- Partitions the system into BLACK BOXES

  ↓

  functionality is known to user but inner details are not known.
- Inputs are given to Black boxes and appropriate outputs generated.
- Using Black boxes reduces complexity of design.
- Modules at top-level call modules at lower level.
- Components are read from Top to Bottom and Left to right.

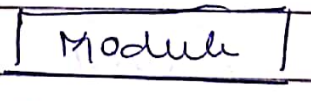


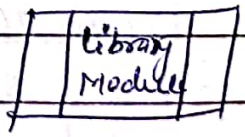
- When a module calls another, it views the called module as black box, passing required parameters and receiving results.

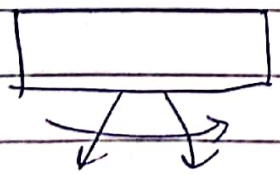
Structure Chart Notation

o→ Data/Parameter

●→ Control info

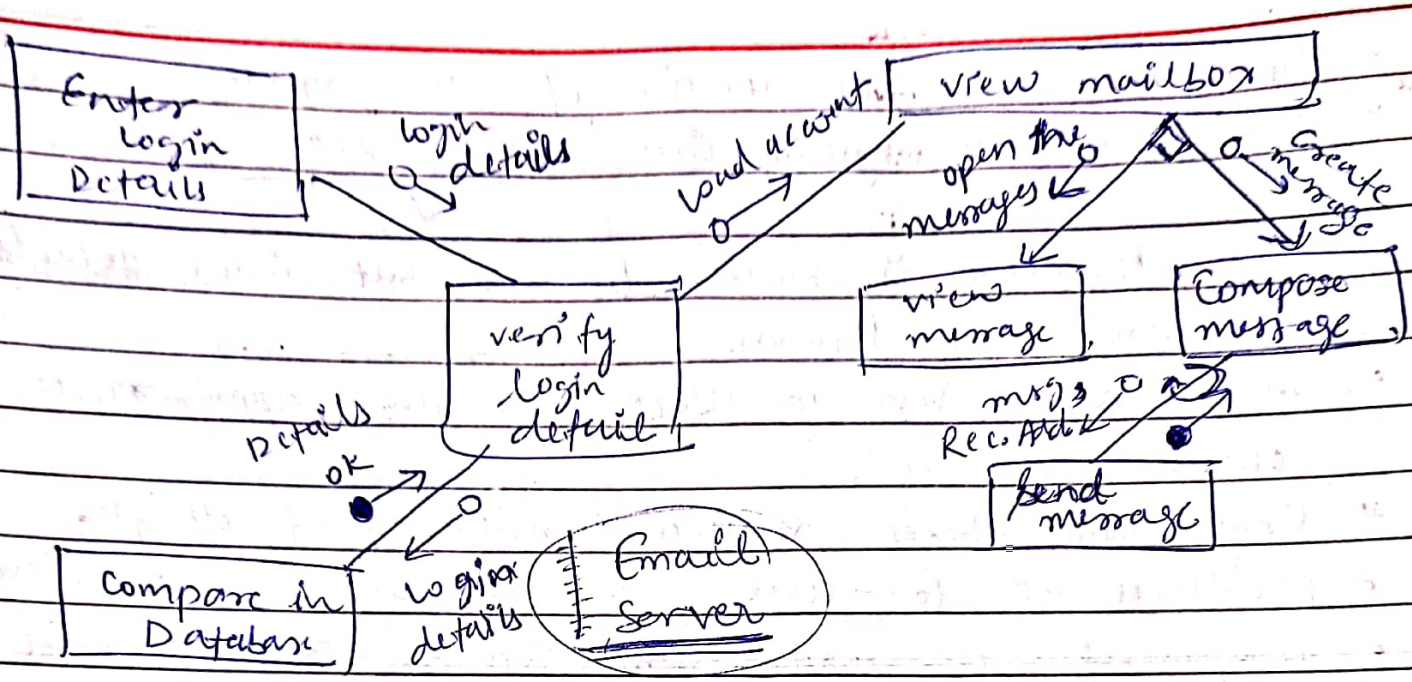






Repetitive call of module

Conditional call



Types of structure charts :-

① Transform - centred structures :-

These types of structure charts are designed for system that recieve an input which is transformed by a sequence of operations, with each operation being carried out by one module.

② Transaction - centred structures :-

These structures describe a system that process a no. of different types of transaction.

Pseudocode

* useful in both preliminary and detailed design phase.
* system description using short English like phrases describing the function.

- Keywords and Indentation: describes the flow of control.
- English phrases & Processing actions performed.

Advantage:

↳ used as a replacement for flow chart.
↳ decrease amount of documentation required.

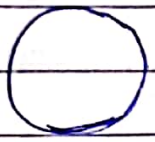
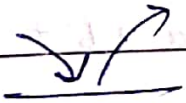
```
void sum(int a, int b)
{
    int c;
    c = a + b;
    cout << c;
}
```

```
Begin:
  Procedure: Sum
    Precondition: integers a & b
    declare variable c
    Assign sum of a & b to c
    print c
End.
```

Data Flow Diagrams

- A DFD shows the flow of data through the system and is also known as used for modelling the requirements.
- Also known as Bubble Chart or Data Flow Graph.

Symbols used in DFD :-

	Process	Depicts a process that transforms data inputs to data outputs.
	Data flow	Shows flow of data into or out of a process or data store.
	Source or sink	An external entity that acts as a source of system I/p or sink of system O/Ps.
	Data Store	Data repository is a collection of data items.

Some important Points:

- Unique names are important
- DFDs depict flow of data and not order of event like a flowchart.
- Decision paths (diamond nodes) represent logical expression are not specified.

Levelling in a DFD:

* DFDs can be drawn to represent the system at diff. level of abstraction.
* Higher level DFDs are partitioned / refined into lower levels → having more information & functional details.

Level - 0 DFD: Context Diagram or Fundamental system model.

Level-0 - DFD represent the entire system as a single bubble with input and output data indicated by incoming & outgoing arrows.

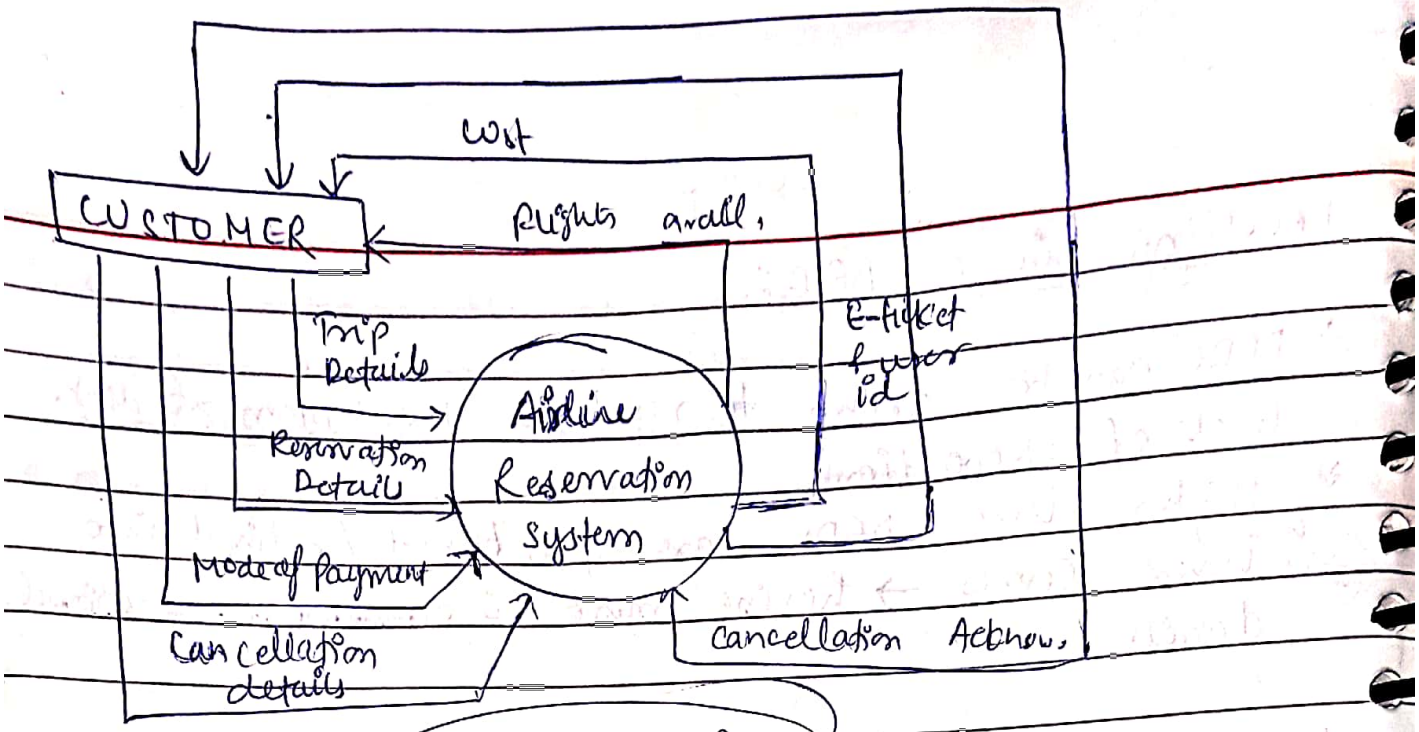
⇓

Decompose this DFD into multiple bubbles.

⇓

Each bubble is then decomposed into more detailed DFDs.

Preserve the no. of inputs & outputs b/w different levels of DFD.

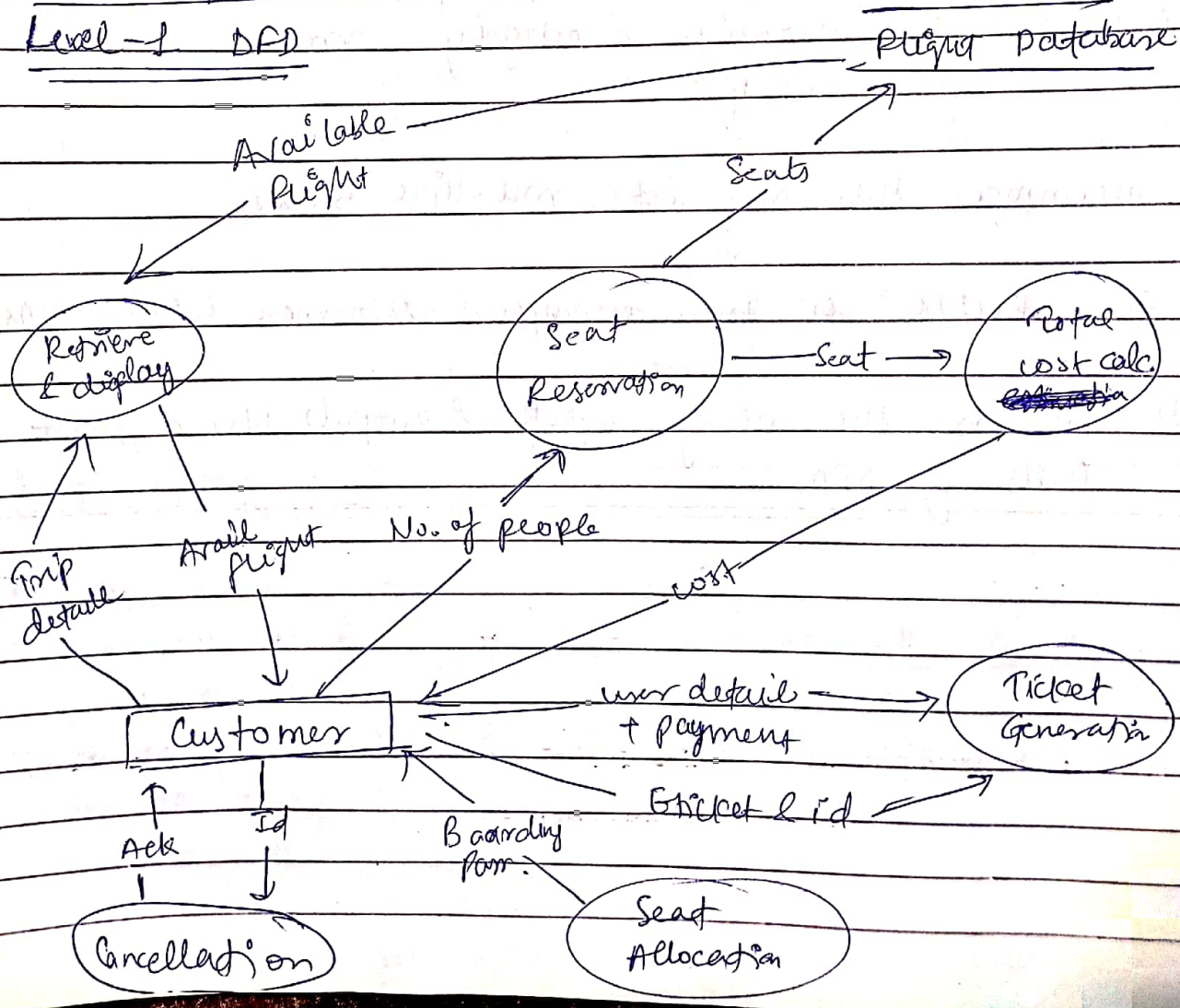


Level - 0 - DFD

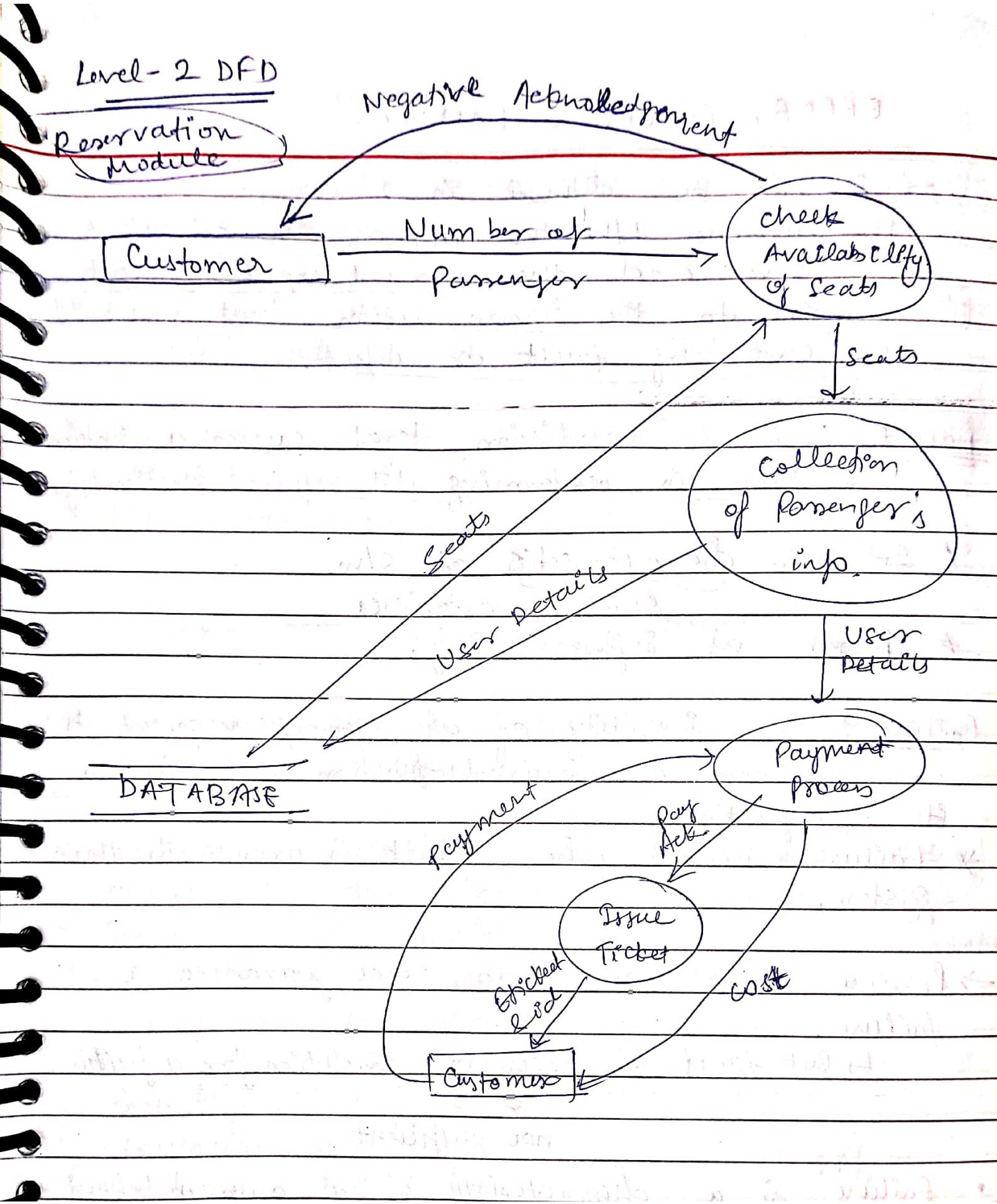
Input

Output

Level - 1 DFD



Level - 2 DFD



ERROR, FAULTS & FAILURES

Error :- can be defined in 2 ways:
(i) The difference b/w the actual observed output of a S/W and the correct expected output.
(ii) It refers to the human action that result in S/W containing fault or defect.

Fault : is a condition that causes a system to fail in performing its required functionality.

★ It is a characteristics of S/W.
Static characteristics
★ Known as Software Bug.

Failure : The inability of a system/component to perform a required function according to its specification.
★ Failure occurs when fault is present in the system.

→ Presence of a fault does not guarantee a failure.
↳ But fault is necessary condition for a failure.
↓
not sufficient

- failure is a characteristic of a program behaviour
↳ occurs only when a fault is executed.
- failure is a dynamic attribute.

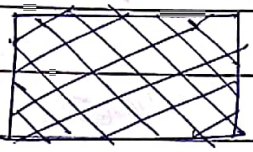
Functional Testing

* Functional testing allows to test the functionality of the program.

↓

Note: It only includes observation of the output for certain input values.

* No code Analysis, No examination of internal structure of code.
* Black Box Testing → contents of the box (S/W) are hidden from user / tester.



* The functionality is understood completely in terms of I/Ps and O/Ps.

* Does not include test cases for maintainance, robustness & other non-functional requirements.
* Internal logic remains unknown.
* It is verified that the system meets its functional specifications / requirements.

Drawback

1. Boundary value Analysis
2. Robustness testing
3. Worst case testing
4. Equivalence class testing
5. Decision table based testing
6. Cause-effect graphing technique.
7. Special value testing.

Different techniques

Boundary Value Analysis

Based on the fact that input values near the boundary have higher chances of error.

What are Boundary values ?

- values lying on the boundary
- values just above boundary (lower side)
- values just below the boundary (upper boundary)

eg : x and y.

eg → x → 1 to 100

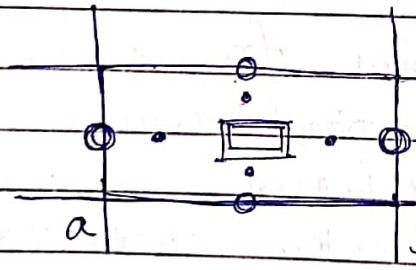
{ 1, 100
  2, 99 }

$a \leq x \leq b$

$c \leq y \leq d$

a → Lower Boundary

b → Upper Boundary



For each variable

- min. value
- just above min.
- Nominal value
- ~~Just below Max. value~~
- Max. value

Single Fault Assumption :→ Holding the values of all but 1 variable at their nominal value and letting that variable assume extreme value.

eg → (x, y) ⇒ (100-300)

→ (100, 200), (200, 100)

→ (101, 200), (200, 101)

→ (299, 200), (200, 299)

→ (300, 200), (200, 300)

For n variables → (Max. of) (4n+1) test cases

∴ 4n+1 Test Cases

Numerical :-

Ques :- Consider a program for determining the previous data. Input : Day, Month, Year with valid ranges as:

1 ≤ Months ≤ 12
1 ≤ Day ≤ 31
1900 ≤ Year ≤ 2000

} Possible o/p : Previous Date or Invalid date

Design : Boundary value Test Cases.

Test Case no.	Month	Day	Year	Output
1.	6	15	1900	14 Jun 1900
2.	6	15	1901	14 Jun 1901
3.	6	15	1960	14 Jun 1960
4.	6	15	1999	14 Jun 1999
5.	6	15	2000	14 Jun 2000
6.	6	1	1960	30 May 1960
7.	6	2	1960	1 Jun 1960
8.	6	30	1960	29 Jun 1960
9.	6	31	1960	Invalid input
10.	6			

10.	1	15	1960	14 Jan 1960
11.	2	15	1960	14 Feb 1960
12.	11	15	1960	14 Nov 1960
13.	12	15	1960	14 Dec 1960

$\Rightarrow$ $4n+1 = 13$ $\Rightarrow$ $4(3)+1 = 13$ test case.

$\downarrow$ $n=3$

Ques → Consider a program to classify a triangle input: Triple of positive integers (x, y, z) and the values range b/w 0 and 100. i.e., $[0, 100]$. Output: Equilateral or isoceles or scalar or Not a triangle. Design boundary value cases.

[Sum of 2 side of Δ > third side] related to not a triangle.

Test case no.	x	y	z	Output
1.	50	50	1	Isosceles
2.	50	50	2	Isosceles
3.	50	50	50	Equilateral
4.	50	50	99	Isosceles
5.	50	50	100	Not a Δ
6.	50	1	50	Isoceles
7.	50	2	50	Isoceles
8.	50	99	50	Isoceles
9.	50	100	50	Not a Δ

10.	1	50	50	Isosceles
11.	2	50	50	Isosceles
12.	99	50	50	Isosceles
13.	100	50	50	Not a Δ

⇒ $4n+1 = 4\times3+1 = 13$ test cases

Equivalence class testing

→ The input and output domain is partitioned into mutually exclusive parts called Equivalence class.
→ Any one sample from a class is representative of entire class.

Steps to make Test class :-

① Identify equivalence classes by taking each input & output condition and partitioning it into valid and invalid classes.
② Generate test cases using equivalence classes.

Example-1 → If a variable lies b/w 1 and 100 then equivalence class will be

all -ve no., 0	$1 \le x \le 100$	all +ves >100
Invalid	valid	Invalid

Example-2 :- Previous Date Program with inputs : day, month, year.

O_1 : All valid I/ps. → Previous date
O_2 : Invalid date

$1 \le m \le 12$, $1 \le d \le 31$, $1900 \le year \le 2025$

I_1 : $1 \le m \le 12$
I_2 : $m < 1$
I_3 : $m > 12$
I_4 : $1 \le d \le 31$
I_5 : $d < 1$
I_6 : $d > 31$
I_7 : $1900 \le Y \le 2025$
I_8 : $Y > 2025$
I_9 : $Y < 1900$

Month	Day	Year	Output
6	15	1962	O_1
2	31	1962	O_2
6	15	1962	O_1
13	15	1962	O_2
6	15	1960	O_1
6	-1	1960	O_2
6	32	1960	O_2
6	15	1962	O_1
6	15	2666	O_2
6	15	1890	O_2

Example : The Triangle Problem ($1 \leq$ Side ≤ 100)

O_1 : Equilateral Δ → 50, 50, 50
O_2 : Isosceles Δ → 99, 50, 50
O_3 : Scalene Δ → 99, 100, 50
O_4 : Not a Δ → 50, 50, 100

I_1 : $x < 1$
I_2 : $x > 100$
I_3 : $1 \leq x \leq 100$
I_4 : $y < 1$
I_5 : $y > 100$
I_6 : $1 \leq y \leq 100$
I_7 : $z < 1$
I_8 : $z > 100$
I_9 : $1 \leq z \leq 100$
I_{10} : $x = y = z$

I_{11} : $x = y$, $x \neq z$
I_{12} : $x = z$, $x \neq y$
I_{13} : $y = z$, $x \neq y$
I_{14} : $x \neq y$, $y \neq z$, $z \neq x$
I_{15} : $x = y + z$
I_{16} : $x > y + z$
I_{17} : $y = x + z$
I_{18} : $y > x + z$
I_{19} : $z = x + y$
I_{20} : $z > x + y$

Test Case	x	y	z	Output
1	0	50	50	I/p invalid
2	101	50	50	"
3	50	50	50	Equilateral
4	50	0	50	Invalid I/p
5	50	101	50	"
6	50	50	50	Equilateral
7	50	50	0	Invalid I/p
8	50	50	101	"
9	50	50	50	Equilateral
10	60	60	60	Equilateral

11	50	50	60	Isosceles
12	50	60	50	Isoceles
13	60	50	50	Isoceles
14	100	99	50	Scalane
15	100	50	50	Not a Δ
16	100	50	25	Δ
17	50	100	50	Δ
18	50	100	25	Δ
19	50	50	100	Δ
20	25	50	100	Δ

DECISION TABLE BASED TESTING

This testing technique is useful when a variety of actions exists and we need to take a particular action depending upon the existing condition.

Every decision table consist of 4 parts:

1. Condition Stub
2. Action Stub
3. Condition Entries
4. Action Entries

Conditions ⇒ Inputs condition

Actions ⇒ Outputs

* If all entries are binary then the decision table is called Limited Entry Decision Table.
* If conditions have several values then the table is called Extended entry decision Table.

X → valid

Condition Stub		ENTRIES							
	c_1	TRUE				FALSE			
	c_2	TRUE		FALSE		TRUE		FALSE	
	c_3	T	F	T	F	T	F	T	F
Action Stub	a_1	X	X			X			X
	a_2	X		X			X		X
	a_3		X			X		X	
	a_4				X		X		

If C_1 is true & C_2 is true & C_3 is true
↳ a_1 & a_2 is performed.

If C_1 is true & C_2 is true & C_3 is false
↳ a_1 & a_3 is performed.

The triangle problem

C_1: x,y,z are sides of Δ	N	←			Y				→
C_2: x = y	–	←	Y		→	←	N		→
C_3: x = z	–	Y	Y	N	N	Y	Y	N	N
C_4: y = z	–	Y	N	Y	N	Y	N	Y	N
a_1: Not a Δ	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

c1 : x < y+z ?	F	T	F	T	T	T	T	T	T	T	T
c2 : y < x+z ?	-	F	T	F	T	T	T	T	T	T	T
c3 : z < x+y ?	-	-	F	T	F	T	T	T	T	F	F
c4 : x = y ?	-	-	-	T	T	T	T	F	F	F	F
c5 : x = z ?	-	-	-	T	F	F	F	T	F	F	F
c6 : y = z ?	-	-	-	T	F	T	F	F	F	T	F
a1 : Not a Δ	X	X	X								
a2 : Scalene											X
a3 : Isosceles							X		X	X	
a4 : Equilateral				X							
a5 : Impossible					X	X		X			

Test case	x	y	z	Expected output
1	4	1	2	Not a Δ
2	1	4	2	11
3	1	2	4	11
4	5	5	5	Equilateral
5	-	-	-	Impossible
6	-	-	-	11
7	2	2	3	Isosceles
8	-	-	-	Impossible
9	2	3	2	Isosceles
10	3	2	2	11
11	3	4	5	Scalene

Cause effect Graphing Method

It is a technique to systematically select a high-yield set of test case.

Benifit : It points out the incompleteness and ambiguities in specifications.

How to Derive Test Case?

Step-1 : Identify the cause (input) and effects (output) from the specification and assign unique no. to each of them.

Cause : Any distinct i/p condition or equivalance class of i/p condition.

Effect : An output condition or system transformation (the effect any i/p has on the state of system).

Step-2 : Draw a boolean graph linking cause and effects. → (Cause effect graph).

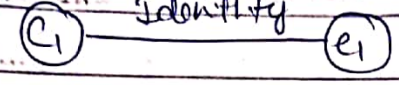
Step-3 : Specify constraints on graph describing combination of cause and/or effect that are impossible.

Step-4 : Convert the graph into limited entry decision table by tracing conditions in graph.

↳ Each column in Decision table is converted into a test case.

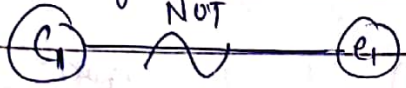
Cause - Effect graph Notations :-

(1) Identity function :



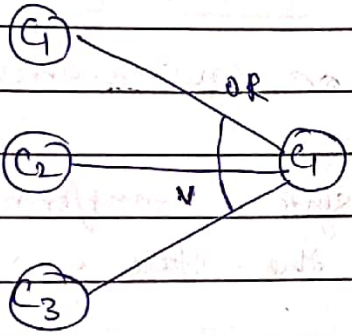
if c_1 is 1, e_1 is also 1
else e_1 is 0.

(2) NOT function



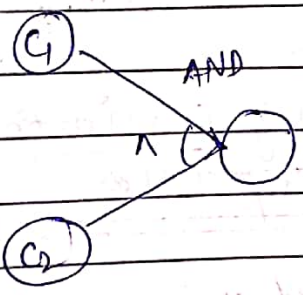
if c_1 is 1 then e_1 is 0
else e_1 is 1.

(3) OR function



if c_1 or c_2 or c_3 is 1
then e_1 is 1,
else e_1 is 0.

(4) AND function



if c_1 and c_2 are 1
then e_1 is 1 else e_1 is 0.

Ques - The character in a column 1 must be A or B. The character in the column 2 must be a digit. If these 2 hold then file update is made. If the character in column 1 is incorrect, message x is issued. If character in column 2 is not a digit, message y is issued.

Sol → Causes are

c_1 : character in column 1 is A

c_2 : character in column 1 is B

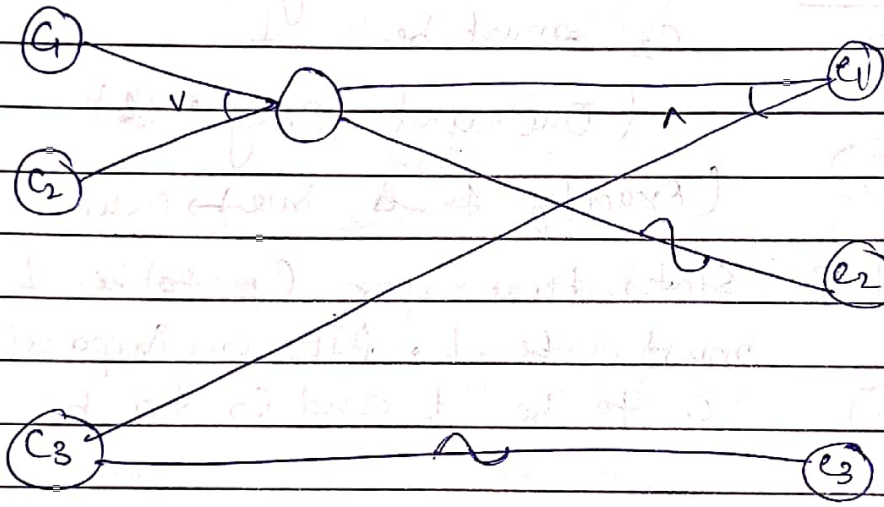
c_3 : character in colum 2 is a digit

Effects are

e_1 : update made $\longrightarrow (c_1 \vee c_2) \wedge c_3$

e_2 : message x is issued $\longrightarrow \bar{c}_1 \wedge \bar{c}_2 \approx \overline{(c_1 \vee c_2)}$

e_3 : message y is issued $\longrightarrow \bar{c}_3$

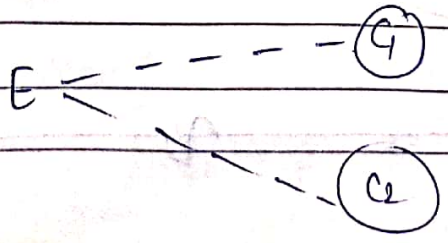


Cause effect Graph

Caux Effect Graph (Types of constraints) :-

To represent impossible combination of cause, few notations are :-

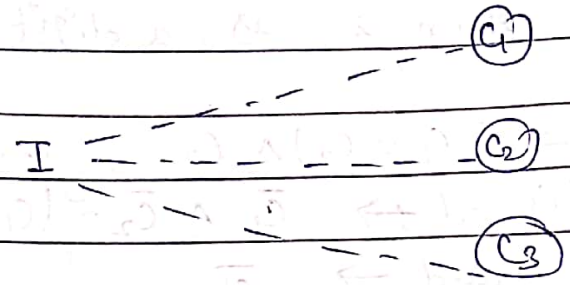
- E constraint : It must always be true that at most one of c_1 or c_2 can be 1 (c_1 & c_2 cannot be 1 simultaenously).



Exclusive

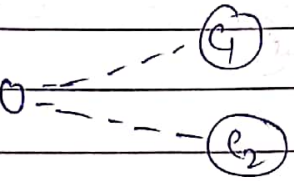
(causes: zero or 1 is true)

- I - constraint : Atleast one of C_1, C_2 and C_3 must always be 1 (C_1, C_2 & C_3 cannot be all 0 simultaneously).



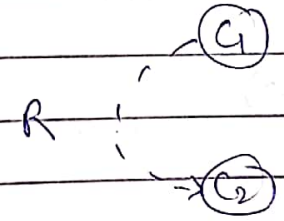
Inclusive

- O - constraint : one and only one out C_1 & C_2 must be 1 (One and only one)



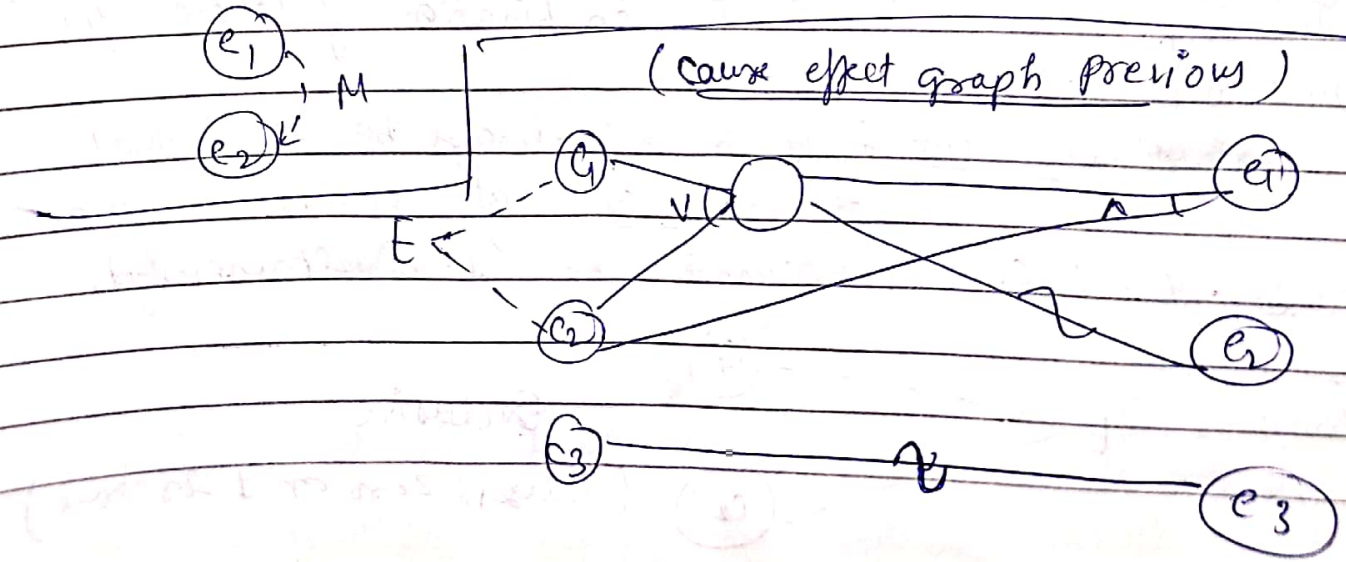
(Exactly 1 is true → causes

- R - constraint : States that for C_1 to be 1, C_2 must be 1. (It is impossible for C_1 to be 1 and C_2 to be 0)

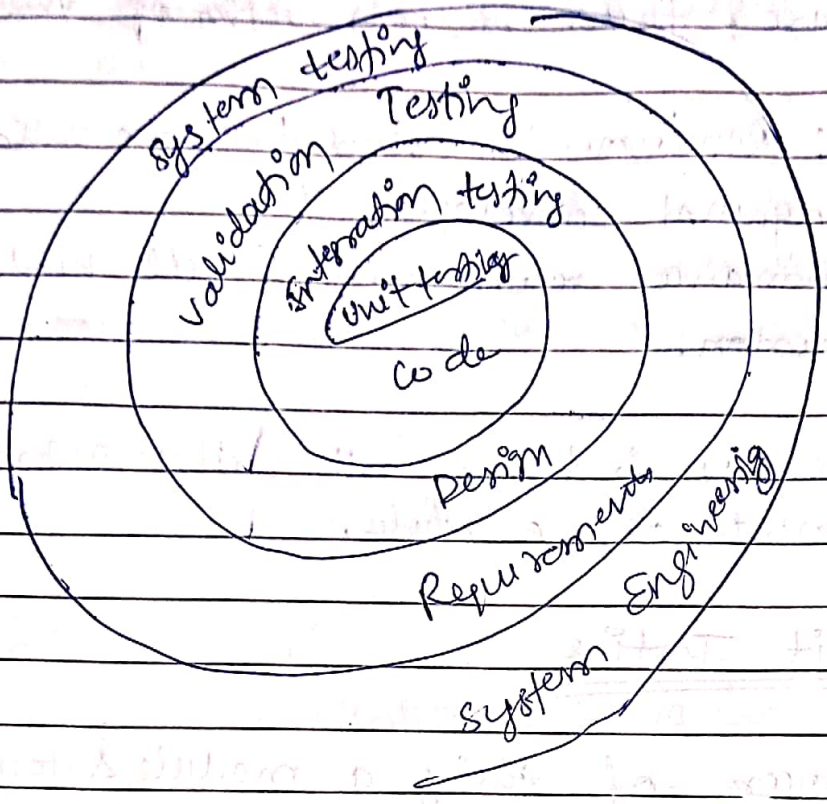


- M - constraint : States that the effect e_1 is 1, effect e_2 is forced to be 0.

(Mask)



Software Testing Strategy



* The software is developed by moving inwards in the spiral
* Testing is conducted by moving outwards in the spiral.

System Engineering: defines the role of the S/W and leads to requirement analysis.

↳ performance constraints, validation criteria, functionality of S/W.

Unit Testing: focuses on testing each module/component independantly based on implementation

↳ unit's internal code/structure, path testing, ensures complete coverage of unit.

Integration testing: Component are integrated together step by step to form a complete software.

↳ S/W architecture, focus on construction of S/W.

Validation Testing: Requirements established are validated against developed S/W.

↳ behaviour, performance requirements & all kind of validation criterion.

System testing: The S/W is tested with other system elements as a whole.

Unit Testing

* It is the process of taking a module & testing it in isolation from the rest of the S/W and comparing with the actual results with the results defined in the specification & design of module.

Why to test each unit independently?

(1) Fault Isolation & debugging becomes easier during unit testing.

(2) We can take an exhaustive approach in case of unit testing.

What to Test During Unit Testing?

→ Module Interfaces tested to check the correct flow of info. in and outside of module.

→ Local Data Structure examined to ensure that

the intermediate results / data is maintained or stored correctly.

→ Independant / Basis paths tested to ensure all statements within the module are executed atleast once during testing.

→ Boundary Conditions
  ↳ output / computer at boundary value is correct.

→ Internal logic
  ↳ logic, precedence, comparison of data types.

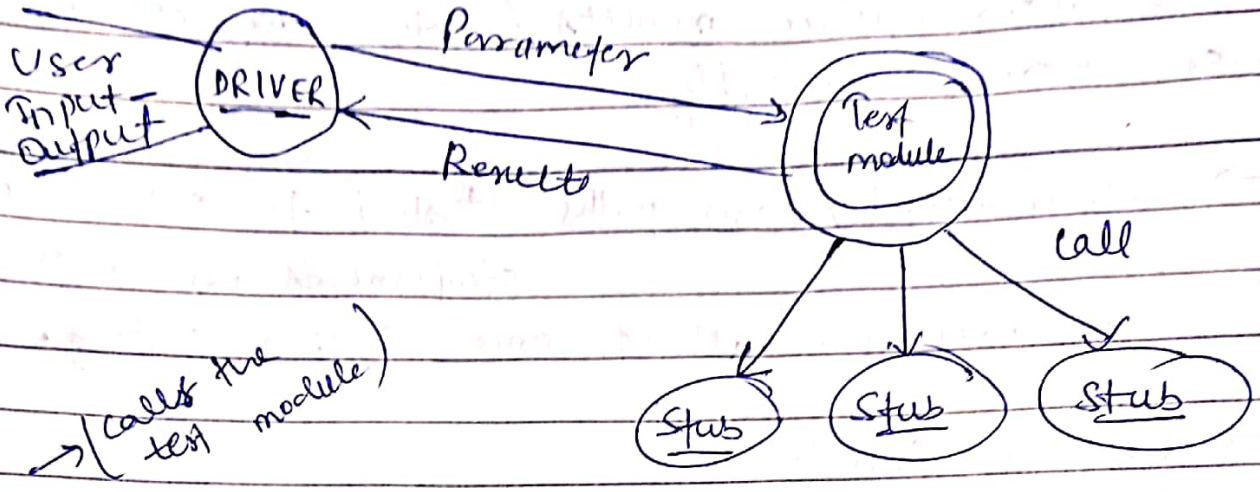
→ Error handling paths

Problems with unit Testing:

A component is not a stand-alone unit.
So,

- How to test it w/o anything to call it?
- How to test it w/o any module to be called by it.

Drivers and Stubs → used to overcome these problems.



DRIVER :- Main program that accepts test case data, passes data to the component to be tested and prints relevant result.
→ (called by test module)

STUB :- Subordinate Modules that are called by the module to be tested.

- It is a dummy sub-program that does minimal data manipulation, provides verification of entry and returns the control to module under testing.

Scaffolding :- The overhead code.
↳ Drivers and Stubs.
↳ A Test Harness.

- used to automatically generate scaffolding
- It allows us to test our module in an isolated environment by simulating the rest of S/W functionalities (taking care of I/P, O/P, parameters etc.).

Integration Testing

Determining the correctness of the interface is the focus of Integration Testing.

Why Integration Tests Required?

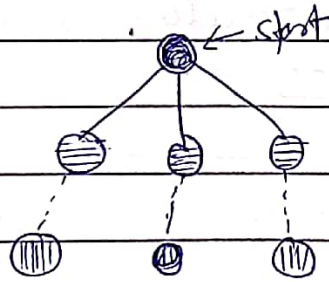
- Data may be lost during interfacing.
- Global data may also cause problems.
- Subfunctions may not work properly when combined.

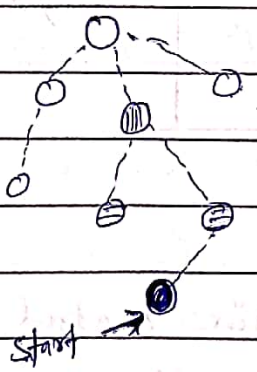
Method to Integrate and Tests

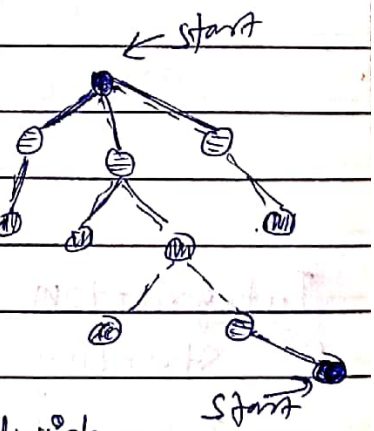
* Unit tested components are taken one by one and integrated incrementally.
  ↳ debugging & fault isolation becomes easier.

Types of Integration :-

① Top-down ② Bottom-Up ③ Sandwich







Top-down Integration	Bottom Up Integration	Sandwich Integration
• Calling module always available • No driver reqd.	• Drivers are reqd. • Stubs are not reqd.	

→ Testing after adding every module.

* In integration Testing, each time a new module is integrated (added), the subsystem (modules integrated till now) changes. New i/o may occur, new data flow paths and control logic may cause problems that previously worked fine.

Top-Down Integration

* Move down in central hierarchy.
* Start with main modul & integrate the sub-modules

Depth-First	Breadth-First
Integrate the units on a major control path. ↓	Combine all the components directly subordinate to a given module. ↓ =

Integration steps :-

1. Starting with the main control module, stubs are substituted for all components subordinate to it.
2. Depending upon type of integration stubs are replaced one at a time with actual components.
3. Test are conducted on each component integrated.

(4) on completion, another stub replaced with actual component.

(5) Regression testing ensures no new errors.

Problems :-

↳ Stubs lead to extra effort in form of overhead code.

↳ Soln. is Bottom-up integration

Bottom-up integration :-

* Begin testing the modules at lowest levels.
* Drivers required but No stubs needed.

↓

the procesing at lower levels is always available.

Step-1 : low-level components are combined into CLUSTERS / BUILDS that perform specific S/w function.

Step-2 : Driver cordinates the test case I/p - O/p.

Step-3 : The cluster is tested.

Step-4 : Driver is removed and the cluster are combined with new module / other clusters moving up in the hierarchy.

Regression Testing :-

(Reason)

The additional of new module as a part of integration testing may cause problems with the functions that previously worked flawlessly

What is it?

The Re-execution of some subset of test cases that have already been conducted to ensure that changes / additions do not create new side-effects.

How is it Done?

Regression testing may be Manual / Automated.

Captalse / Playback tools

- These tools enables the s/w engineers to capture the test cases and their results for subsequent playback and comparision.

What tests are conducted?

- examine the functionality of all modules
  - ↳ focus on functionality that is most likely to get affected by the newly integrated module
- Test only those s/w components that have changed.

Validation Testing

* focuses on user-variable actions and user recognizable output from the system.
* Validation succeeds when the s/w functions in a manner that can be expected by the customer.
* Validation Criterian
  - ↳ A section in the SRS document
  - ↳ describes the desired functionality that for basis of validation testing.

How is Validation Achieved?
Through a series of tests that clearly demonstrate the existence of functionality reqd. by user in s/w.

What does Validation Testing Ensure?
- functionality is achieved
- Correct behaviour achieved
- Performance constraint met.
- documents are correct.
- Non-functional requirements like usability etc.

* A deficiency list may be created in case something is missing / correct.

System Testing

- It incorporates the S/w with other system elements.
- System testing ensures that all system elements have integrated well with the S/w and the entire system functions properly.

↳ ensures full proof testing of entire system.

↓

S/w + other components

Types of System Testing :-

Recovery Testing :-

Ensures that the system must recover from faults and resume processing with little or no downtime.

Aim – To develop a fault tolerant system

Any processing fault should not bring the entire system.

- S/w must recover from faults and resume its normal processing within a specified time period.

How it is done?

↳ Force the system to fail in diff. ways and then we verify that the recovery of the system is done properly.

★ (Mean time to repair) → Manual Testing

Security testing :-

verifies that the protection mechanism built into the system will actually protect it from attacks.

Responsibility of tester

- acts as an intruder
- Ensures that the cost of attack is higher than the info. achieved.

Stress Testing :-

It executes the system in a manner that demands resources in abnormal quantity.

[eg. (10) per minute]

- excessive memory requirements
- database requests
- Simultaenous log-in.

Sensitivity testing :-

* A variation of stress testing.
* It attempts to uncover data combinations within the valid input domain, which may cause improper functioning.

Performance testing :-

- Aimed at testing the run-time performance of the S/W.

- Ensure that performance is monitored continuously beginning from unit testing phase itself.

Deployment testing :-

- Also known as Configuration testing.
- It tests the S/w in each environment in which it is to operate.
- examines all installation procedures, documentation required.

Acceptance Testing :-

* conduct when S/w is developed for a specific customer and not for a large public / audience.

Purpose : Allows customer to validate all requirements.

Process : Customer tests the S/w → Feedback
(Neg) ↑ Modification done ← Final S/w delivery

Alpha and Beta Testing

* conducted when the product is developed for the anonymous customer
  ↳ ~~Acceptance testing is not possible.~~

* carried out by the customer.

Alpha Testing	Beta Testing
(1) Done by customer at developer site.	(1) Done by customer at users/customers site.
(2) Conducted in a controlled environment	(2) Conducted in real time environment, not under the developers control.
(3) Developer is present.	(3) Developer is not present.
(4) Carried out before the release of the product to customer.	(4) Carried out after release of product to customer.
(5) Errors/failures are recorded.	(5) Failures are reported.
(6) White + Black Box Testing	(6) Black Box Testing

Verification	Validation
• Are we building the system right? ↳ It is the process of evaluating a system or component to determine if the products of a development phase satisfy the conditions imposed at start of that phase.	• Are we building the right software / system? ↳ It is the process of evaluating a system/component during or at the end of development process to determine whether it satisfies the specified requirements.
• Applied during development phases.	• Towards the end of development process.
• Manual Testing: we look & review the docs	• Program execution: requirements are validated.
• Activities involved ↳ Reviews, meetings, inspection	• Activities involved ↳ Black box & white box testing
• Carried out by Internal Quality Assesment Team	• Carried out by testing team.
• Done prior to Validation	• Done after verification

* Both verification & validation are complementary activities. If we are able to find more errors before execution (verification) validation becomes easier.
(Minimise error in early phases of development.)

Software Maintenance

Software maintenance includes:
→ error correction
→ enhancement of capabilities
→ deletion of obsolete capabilities
→ optimization

Any work done to change the S/W after it is in operation is considered to be maintenance work.

Purpose:- It preserves the value of S/W over time.

4 categories of Maintenance:

(1) Corrective Maintenance:

↳ Initiated as a result of defects in S/W. any kind of coding, design, logic error.

→ Appropriate actions are taken to restore the correct functionality of the S/W system.

Patching: Emergency fixes (mainly due to pressure from management).
↳ gives rise to unforeseen future-errors due to lack of proper impact analysis.

(2) Adaptive Maintenance:

↳ Modifying the S/W to match changes in ever-changing environment.

any outside factors: Platform changes, govt. policies, business rules.

(3) Perfective Maintenance :-
↳ Improving the procesing efficiency OR performance or restructuring the S/W to improve changeability

- Expansion in Requirements ⟹ Enhancing existing system functionality.
- Better, Faster, Cleaner.

(4) Preventive Maintenance :-
↳ Making the program easier to understand.
↳ helps us in future maintainance work.
eg :- optimization, documentation updation.

RELIABILITY

Reliability of a S/W specifies probability of failure free operation for a given time duration.
(presence of faults)

★ It is a dynamic system characteristic
↳ a function of no. of software failures

It is an execution event where the S/W behaves unexpectedly.

Not all S/W faults have equal probability of manifestation / execution.
↳ Removing S/W fault from those S/W parts which are rarely used makes little improvement in Reliability.

Operational Profile of S/W : The way in which S/W is used by the users.
- The kind of i/p that are supplied to the S/W.

Failures of a S/W depends on 2 factors:
(i) No. of faults being evaluated in S/W.
(ii) Operational profile of execution of S/W.

Types of time:

① Execution time: The actual CPU time that the S/W takes during its execution.

② Calendar time: Normal / Regular time that we use on daily basis.

③ Clock time: Actual time that is elapsed while the S/W is executing
⇓
includes the time that the S/W spends while waiting in system.

Reliability Metrics :-

① Probability of failure on demand:

- It is a measure of likelihood that the system will behave in an unexpected way when some demand is made on it.

eg: Safety-critical System.

② Rate of occurance of failure (ROCOF):

A measure of frequency of occurance with which unexpected behaviour is likely to be observed.

eg:- ROCOF = $\frac{2}{100}$ → S/w will fail 2 times out of 100 operational unit times.

③ Mean time to failure (MTTF):

- A measure of time interval b/w observed failures.
- Useful when system is stable and no changes are being made to it.
  ↳ Indication of how long the system will be operational before failure occurance.

④ Availability: Measure of how likely the system is to be available to you.

eg:- Availability = $\frac{10}{100}$

**

$$MTBF = MTTF + MTTR$$

MTBF: Mean time b/w failure
MTTF: Mean time to failure
MTTR: Mean time to repair

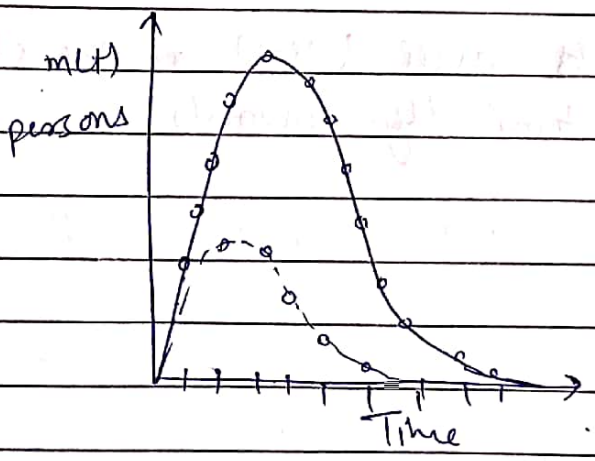
*

$$\text{Availability} = \left(\frac{MTTf}{MTTf + MTTR}\right) \times 100\%$$

Putnam Resource Allocation Model

- Based on Rayleigh curve for approximating Hardware development projects
  ↓
  Norden of IBM

- Putnam observed that Rayleigh curve was a close approximation for S/w project and S/w subsystem development.



Rayleigh Manpower loading curve

——— Overall curve
- - - - Design & Coding

- It measures Manpower in terms of person per unit time as a function of time.
  ↳ Person – year / year

The Rayleigh curve is given by differential equation

$$m(t) = \frac{dy}{dt} = 2Kat\,e^{-at^2} \quad \text{——— (1)}$$

dy/dt = Manpower utilization rate per unit time
t = elapsed time
a = parameter affecting shape of curve
K = area under curve in interval $[0, \infty]$

Integrating (1) in interval $[0, t]$ we get

$$y(t) = K[1 - e^{-at^2}] \quad \text{— (2)}$$

$y(t)$ ↓ Cumulative Manpower used upto time t.

$y(0) = 0$ $\quad$ $y(\infty) = K$

The cumulative manpower is null (zero) at the start of the project & grows monotonically towards the total effort K.

Importance of 'a' :-

Dimension of a is $1/\text{time}^2$

It plays an important role in determination of peak manpower.

↳ Larger value of 'a', earlier will be the occurance of peak. & steeper is person - profile.

$$\frac{d^2y}{dt^2} = 2Kae^{-at^2}[1 - 2at^2] = 0$$

$$t_d^2 = \frac{1}{2a} \Rightarrow a = \frac{1}{t_d^2} \quad \text{— (3)}$$

t_d : time where max. effort rate occurs

→ t_d represents the total project development time.

put t_d in place of t in eq – ②

Estimate of development time

$$E = y(t) = K\left(1 - e^{-\frac{t_d^2}{2t_d^2}}\right)$$

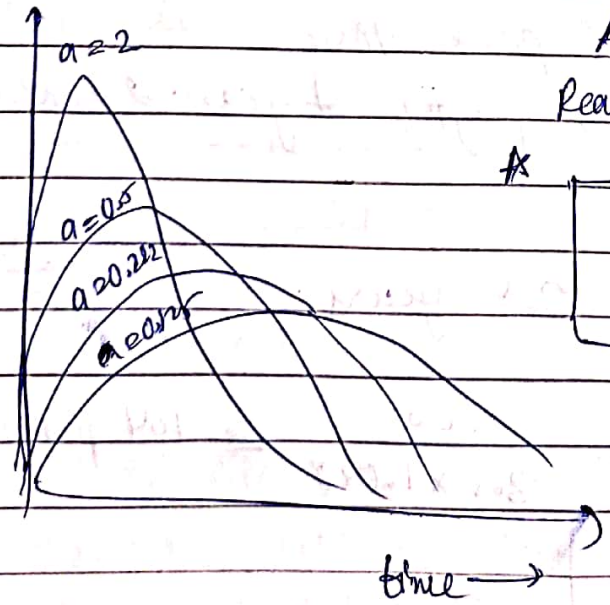
$$E = K(1 - e^{-0.5}) = 0.393K$$

$$\boxed{E = 0.393K} \quad \text{—④}$$

Put a - value from eq – ③ in eq – ①

$$m(t) = \frac{2K}{2t_d^2} t\, e^{-\frac{t^2}{2t_d^2}} = \boxed{\frac{K}{t_d^2} t\, e^{-t^2/2t_d^2}} \quad \text{—⑤}$$

people involved in project at the peak time.



At time $t = t_d$

Peak Manpower is given by

$$\boxed{m(t_d) = m_0 = \frac{K}{t_d\sqrt{e}}} \quad \text{—⑥}$$

K → total project cost or effort (person – years)

t_d → delivery time in years

Numerical on Putnam Model

Ques → A software project is planned to cost 95 PY in a period of 1 year 9 months. Calculate peak manning and average rate of S/w team build up.

Sol → Software project cost, $K = 95$ PY

Peak development time $= 1 + \frac{9}{12} = 1.75$ years $= t_d$

Peak manpower : $m_0 = \frac{K}{t_d \sqrt{e}} = \frac{95}{1.75 \sqrt{e}} \rightarrow 1.648$

$$m_0 = 33 \text{ persons}$$

* Average rate of S/w team build up $= \frac{m_0}{t_d} = \frac{33}{1.75}$

$= 18.8$ person/year

Ques → Consider a large scale project for which manpower requirement is $K = 600$ PY and development time is 3 year and 6 months.

(a) Calculate peak manning & peak time.

(b) What is the manpower cost after 1 year 2 months.

Sol → (a) $K = 600$ PY

$t_d = 3 + \frac{6}{12} = 3.5$ years

$m_0 = \frac{K}{t_d \sqrt{e}} = \frac{600}{3.5 \times 1.648} \simeq 104$ persons

$$m_0 \simeq 104 \text{ persons}$$

(b) $y(t) = K[1 - e^{-a \cdot t^2}]$

$t = 1 + \frac{2}{12}$ year $= 1.17$ years

$a = \frac{1}{2t_d^2} = \frac{1}{2 \times (3.5)^2} = 0.041$

$y(1.17) = 600[1 - e^{-(0.041 \times (1.17)^2)}]$

$y(1.17) = 32.6$ PY

Reverse Engineering

It is the process followed so as to find unknown and difficult information about a software.

Why is it important?
It is important in case when software lacks proper documentation, is highly unstructured or its structure has degraded through maintenance works.

Purpose: Recovering the information from existing code/ any intermediate documents.
↳ Program understanding at any level is included in R.E.

Uses of Reverse Engineering:
- Program understanding.
- Redocumentation or document generation

- Recovery of design detail
- Identify Reusable components.
- Business rules implied on S/W.
- Understanding high level system description.
- Identify components that require restructuring.

What is not in scope Reverse engineering ?

* Re - design
* Re - structuring
* Enhancement of system functionality.

Major tasks :-

(1) Mapping the program to the (problem it represents in) application domain.

(2) Mapping b/w concrete and abstract levels

High level abstraction	Detailed design	Concrete implementation

(3) Rediscover high level structure.
↳ understanding implementation detail

(4) Find missing links b/w program syntax & semantics.

(5) Extract Re - usable components.
↳ to find no. of components that can be reused from this code.